VOLUME FOUR

Building and Shipping Software

from *How Computers Work*

KedByte Technologies Private Limited

KEDBYTE

TECHNOLOGIES PRIVATE LIMITED

HOW COMPUTERS WORK

From a Grain of Sand to Artificial Intelligence

VOLUME FOUR

Building and Shipping Software

The web and how it is built, APIs and where they came from, version control down to the object database, automation, the cloud and the mandate that created it, and keeping all of it safe.

WRITTEN BY

Shikhar Singh

Founder & Chairman

KedByte Technologies Private Limited

Part G • Chapters 35 to 44 • First Edition

HOW COMPUTERS WORK

Volume Four — Building and Shipping Software

Part G, chapters 35 to 44

Author	Shikhar Singh
Designation	Founder & Chairman, KedByte Technologies Private Limited
Published by	KedByte Technologies Private Limited
Imprint	KedByte Press
Edition	First Edition
Volume	4 of 5
Complete work	Nine parts, fifty-three chapters

About this volume. This is Volume 4 of five. It contains Part G of *How Computers Work*, chapters 35 to 44. 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 G — Building and Shipping Software 1

35 Web and App Development 2

35.0 What this chapter gives you 2
35.1 Frontend, backend and the line between them 2
35.2 HTML: the structure of a page 5
35.3 CSS: presentation, and the cascade that confuses everyone 8
35.4 JavaScript: the language and its event loop 12
35.5 The DOM and how a browser turns HTML into pixels 15
35.6 The browser as a platform 18
35.7 Frontend frameworks, explained rather than listed 21
35.8 Build tooling: why frontend code needs a build step 24
35.9 The backend: what a server program actually does 27
35.10 Databases 30
35.11 ORMs and data access 34
35.12 Authentication and sessions 36
35.13 Application architecture 41
35.14 Mobile apps 46
35.15 Testing, performance and accessibility for the web 50
35.16 A realistic learning path 55
35.98 Common wrong ideas 60
35.99 Chapter summary in 20 lines 61

36 APIs — What They Are and Where They Came From 62

36.0 What this chapter gives you 62
36.1 What an API actually is 62
36.2 The history, properly 65
36.3 Local APIs: signatures, libraries and versions 69
36.4 System call APIs 71
36.5 Remote APIs: what the network changes 74
36.6 HTTP APIs in practice 77
36.7 REST, properly 80
36.8 The alternatives, compared honestly 82
36.9 Authentication and authorization for APIs 85
36.10 Making an API reliable in the real world 88
36.11 Webhooks: the inversion where the server calls you 94
36.12 Designing an API people can actually use 99
36.13 Consuming an API well 104
36.14 APIs as a business 109
36.98 Common wrong ideas 113
36.99 Chapter summary in 20 lines 114

37 Git — Why It Works With No Network 116

37.0 What this chapter gives you 116
37.1 What version control is for 116
37.2 Centralized versus distributed 119
37.3 Why Git was made 122
37.4 What is actually inside a .git directory 125

37.5 What “the whole history” really means 128
37.6 What “origin” actually is 131
37.7 Remote-tracking refs and staleness 134
37.8 Which git operations need the network 137
37.9 The reader’s outage, worked through 141
37.10 The trade-offs of the distributed model 144
37.11 Git is not GitHub 146
37.12 Getting started properly 149
37.98 Common wrong ideas 152
37.99 Chapter summary in 20 lines 152

38 What a Commit Actually Is 154

38.0 What this chapter gives you 154
38.1 Content-addressed storage, the central idea 154
38.2 Hashing, and the honest truth about SHA-1 157
38.3 The four object types 160
38.4 How the four types compose 163
38.5 Why a commit points at its parent 167
38.6 What a branch really is 170
38.7 HEAD 173
38.8 The rest of the refs 176
38.9 The reflog 178
38.10 The commands that reveal the machinery 182
38.11 How git stores things efficiently 185
38.12 Reading the reader’s own situation 188
38.13 Why this design is good, and what it costs 191
38.98 Common wrong ideas 194
38.99 Chapter summary in 20 lines 194

39 The Index, Push, Fetch, Merge and Rebase 196

39.0 What this chapter gives you 196
39.1 The three states and three places 196
39.2 What the index actually is 199
39.3 What `git add` really does 202
39.4 What `git commit` really does 205
39.5 Undoing things, laid out clearly 208
39.6 What actually crosses the wire on a push 211
39.7 The reader’s own failure, analysed properly 215
39.8 Fetch versus pull 218
39.9 Merge 221
39.10 Conflicts at the file level 224
39.11 Rebase 227
39.12 The reader’s rebase, explained 230
39.13 When rebasing is safe and when it destroys work 232
39.14 Merge versus rebase as a team policy 234
39.15 Cherry-pick, revert, bisect and the rest 236
39.98 Common wrong ideas 239
39.99 Chapter summary in 20 lines 239

40 Your Own Git Server, and What GitHub Adds 241

40.0 What this chapter gives you 241
40.1 A git server is just another repository 241
40.2 Bare repositories 244
40.3 Doing it end to end 247
40.4 Remotes in depth 251
40.5 Hooks: server-side and client-side 256
40.6 What GitHub actually is 261
40.7 What GitHub adds that plain git does not have 263
40.8 The exact boundary, using the reader’s own outage as evidence 268
40.9 The self-hosted alternatives 271

40.10 Migrating and mirroring 275
40.11 Working with others, practically 279
40.12 Monorepos and large repositories 283
40.13 Backing up a git repository properly 288
40.98 Common wrong ideas 292
40.99 Chapter summary in 20 lines 293

41 CI/CD and What a "Run" Really Is 294

41.0 What this chapter gives you 294
41.1 What continuous integration means and the problem it solves 294
41.2 Continuous delivery and continuous deployment 297
41.3 What a runner actually is 299
41.4 The workflow file 301
41.5 Triggers: what starts a run 305
41.6 The most important idea: a run belongs to a commit, not a branch 310
41.7 Why verifying by run ID against a head SHA is the stronger claim 313
41.8 Statuses, checks, and the two APIs behind them 317
41.9 Merge queues 321
41.10 `mergeStateStatus`, value by value 325
41.11 What a good pipeline actually does, in order 328
41.12 Deployment strategies 331
41.13 Secrets in CI 334
41.14 Reading a failed run like an engineer 338
41.15 The wider landscape 342
41.98 Common wrong ideas 345
41.99 Chapter summary in 20 lines 345

42 SSH, HTTPS, Keys, Tokens and Scopes 347

42.0 What this chapter gives you 347
42.1 The two transports git uses, and why there are two 347
42.2 What SSH actually is 351
42.3 Public key cryptography for SSH 354
42.4 `authorized_keys` and how it works 358
42.5 `known_hosts` and host key verification 362
42.6 The SSH agent 366
42.7 Why SSH on port 22 can be blocked when 443 is not 369
42.8 HTTPS authentication for git 373
42.9 Personal access tokens 377
42.10 Scopes, properly 380
42.11 OAuth, tokens, SSH keys and GitHub Apps 384
42.12 Practical hardening for a developer's own machine 387
42.13 A diagnosis table for git authentication failures 391
42.98 Common wrong ideas 394
42.99 Chapter summary in 20 lines 395

43 Cloud Computing, the Bezos Mandate, AWS and Azure 396

43.0 What this chapter gives you 396
43.1 What the cloud actually is 396
43.2 Inside a real datacentre 399
43.3 Regions, availability zones and edge locations 402
43.4 Virtualization, the technology that made cloud possible 405
43.5 Containers 408
43.6 Kubernetes and orchestration 412
43.7 The Bezos API mandate 415
43.8 How AWS actually started 418
43.9 The AWS services you actually need to know 421
43.10 Azure, Google Cloud and the rest 424
43.11 The service models: IaaS, PaaS, SaaS and FaaS 426
43.12 Serverless, properly 428
43.13 Storage in the cloud 432

43.14 Cloud networking 434
43.15 Identity and access 437
43.16 What cloud costs and how bills explode 439
43.17 DevOps, infrastructure as code and the modern workflow 442
43.18 Microservices versus monolith, decided honestly 446
43.98 Common wrong ideas 449
43.99 Chapter summary in 20 lines 450

44 Security 451

44.0 What this chapter gives you 451
44.1 What security actually means 451
44.2 Hashing 454
44.3 Password storage done properly 457
44.4 Symmetric encryption 461
44.5 Asymmetric encryption, signatures and the quantum question 464
44.6 Authentication and identity 467
44.7 Authorization and access control 470
44.8 The web application vulnerability classes, defensively 473
44.9 Memory safety 476
44.10 Network attacks conceptually 478
44.11 Malware and how infection actually happens 481
44.12 Social engineering 484
44.13 Supply chain security 486
44.14 Defence in depth for a real system 488
44.15 Detection and response 491
44.16 Privacy and the law, briefly 493
44.17 Security for the individual developer 495
44.18 Famous breaches and what each teaches 497
44.98 Common wrong ideas 499
44.99 Chapter summary in 20 lines 500

PART G

Building and Shipping Software

The web, APIs, version control, automation, the cloud and keeping it all safe.

35

CHAPTER

KB

Web and App Development

35.0 What this chapter gives you

1. You will be able to draw the line between frontend and backend, say exactly what runs on the user's device and what runs on a server you control, and state the security rule that follows from it.
2. You will be able to say what HTML, CSS and JavaScript each are, why they are three separate things, and why only one of them is a programming language.
3. You will be able to explain the CSS cascade, specificity with the real calculation, the box model, flexbox and grid, and stop guessing at layout.
4. You will be able to describe the JavaScript event loop with the call stack, the task queue and the micro-task queue, and predict the output order of a piece of asynchronous code by hand.
5. You will be able to trace an HTML file all the way to lit pixels: parse, DOM, CSSOM, render tree, layout, paint, composite, and say what reflow costs.
6. You will be able to say what a frontend framework actually solves, how a virtual DOM and reconciliation work, and when you do not need one at all.
7. You will be able to write a minimal web server in two languages, explain routing and middleware, and compare Node, Python, Java, Go, Ruby, PHP and .NET on honest grounds rather than fashion.
8. You will be able to design a small relational schema, write SQL with joins and grouping, explain how a B-tree index turns a scan into a lookup, read an EXPLAIN plan, and define ACID one letter at a time.
9. You will be able to store a password correctly, choose between sessions and tokens, describe the OAuth 2.0 authorization code flow step by step, and separate authentication from authorization.
10. You will be able to replace the "180-day roadmap" with an order of learning that matches how long these things really take.

35.1 Frontend, backend and the line between them

35.1.1 PLAIN – in simple words

1. A web application lives in two places at once.
2. Part of it runs on the user's own device, inside the browser. That part is called the **frontend**.
3. Part of it runs on a computer you own or rent, far away. That part is called the **backend**.
4. The frontend is what the user sees and touches: text, pictures, buttons, boxes to type in.
5. The backend is what the user never sees: the stored data, the rules about who may do what, the money, the emails, the search.
6. The two talk to each other over the network, using the request and reply pattern from the networking chapters.
7. The device asks, the server answers. The device asks again, the server answers again. That is the whole shape of it.
8. This is called **client-server**. The client is the asking side. The server is the answering side.

9. Here is the single most important rule in this whole chapter. Everything on the user's device can be changed by the user.
10. The user can edit the page, change the numbers you sent, delete your checks, and send you anything at all.
11. So the frontend can be polite, helpful and fast. It can never be trusted.
12. Every rule that matters must be checked again on the server, where the user cannot reach it.

35.1.2 PLAIN - a picture in your head

1. Think of a bank branch.
2. The counter, the forms, the pen on a chain and the queue barrier are the frontend. They are in the public part of the building.
3. The vault, the ledgers and the staff-only room are the backend.
4. The form asks you to write your account number in the right shape. That is helpful. It stops honest people making honest mistakes.
5. But nobody thinks the form is security. A person can write anything on a form. They can bring their own form. They can shout their request instead.
6. So the clerk checks the request again, against the real ledger, behind the glass, before any money moves.
7. That second check is the backend check. The form was the frontend check.
8. Both are useful. Only one of them is protection.

Where this comparison breaks:

9. In a bank you can see the customer. On the web you cannot. The "customer" may be a program sending ten thousand requests a second from another country.
10. In a bank the forms live in the branch. On the web, you hand the customer a complete copy of your entire form-processing machine and let them run it at home. They can take it apart first.
11. And a bank branch serves one person at a time at each counter. A server handles thousands of half-finished conversations at once.

35.1.3 PLAIN - a worked example

1. A shop page shows a jacket at 4,000 rupees. The page has a discount box.
2. The frontend code checks the discount code, works out 10 percent off, and shows 3,600.
3. The user then presses Buy. The browser sends a message to the server.
4. A careless developer sends the price in that message:

```
{ "item": "jacket-01", "price": 3600, "code": "SAVE10" }
```

5. The user opens the browser's developer tools, changes one number, and sends this instead:

```
{ "item": "jacket-01", "price": 1, "code": "SAVE10" }
```

6. If the server saves that order, the shop has just sold a jacket for one rupee. Nothing was hacked. The user simply typed a different number.
7. The correct message contains no price at all:

```
{ "item": "jacket-01", "code": "SAVE10", "qty": 1 }
```

8. The server looks up the real price of `jacket-01` in its own database, checks whether `SAVE10` is real, unexpired and allowed for this user, and computes the total itself.
9. The frontend showed 3,600 so the user was not surprised. The server decided 3,600 because it is the only side that can decide.
10. This exact mistake has cost real companies real money many times.

35.1.4 PLAIN - what is really happening inside

1. The browser is a program on the user's machine. Chapter 18 called such a program a process: code plus memory, run by the operating system.
2. When you visit a site, the browser makes a network connection to a server, as in Chapters 28 and 32, and asks for a document.
3. The server sends back text. Usually HTML, which describes the content.
4. The browser reads that text and builds an internal model of the page in its own memory. Then it draws that model on the screen.
5. The HTML normally refers to more files: stylesheets, scripts, images, fonts. The browser fetches each of those with more requests.
6. Once a script is running, it can make further requests on its own, without loading a new page. That is how modern pages update in place.
7. Every one of those requests is a separate arrival at the server, and the server has no memory of the previous one unless you build that memory.
8. That last point is what "HTTP is stateless" means, and it is why sessions and tokens exist at all, which is section 35.12.
9. The split exists for three reasons. Speed, because a round trip to a server costs tens or hundreds of milliseconds and a local reaction costs none.
10. Control, because your data and your rules must live where users cannot edit them.
11. Scale, because one server can serve many clients only if the clients do their own drawing.

35.1.5 TECHNICAL - the engineer's version

1. The client-server model is an architectural style, not a protocol. HTTP is one instance of it. It was covered from the wire upward in Chapters 23 to 33.
2. HTTP/1.1 is defined by RFC 9110 to RFC 9112 (June 2022), which replaced the older RFC 7230 series. HTTP/2 is RFC 9113. HTTP/3 over QUIC is RFC 9114.
3. HTTP is stateless by design: each request carries everything needed to process it. State is reconstructed from cookies, headers or tokens.
4. The trust boundary sits at the network edge of your server. Everything on the far side of it is **untrusted input**, including your own JavaScript.
5. The formal rule is the end-to-end argument from Saltzer, Reed and Clark (1981, 1984): a check is only meaningful at the point that owns the outcome.

Concern	Frontend can	Server must
Field format	show hints early	validate again
Price, totals	display only	compute, authoritative
Permissions	hide buttons	enforce on every call
Rate limits	debounce clicks	count and reject
Secrets	never hold any	hold all of them

6. Client-side validation is a usability feature with a measurable benefit: it removes a network round trip, typically 30 to 300 ms depending on distance.
7. Anything shipped to the browser is public. API keys in a bundle, "hidden" admin routes and commented-out code are all readable. Minification is not obfuscation and obfuscation is not security.
8. Tools that make this concrete: browser DevTools Network and Sources panels, `curl -v`, Burp Suite and mitmproxy for intercepting and editing requests.
9. Server-side rendering and client-side rendering shift where HTML is produced, but they do not move the trust boundary at all. That is fixed.

35.1.6 WORDS - remember these

1. Frontend — the part running on the user's device — client-side code executing in the browser's JavaScript engine and rendering engine.
2. Backend — the part running on your machine — server-side processes you control, holding data, secrets and authorization logic.
3. Client-server — one side asks, the other answers — an architectural style where a requesting process contacts a listening service process.
4. Stateless — the server forgets between requests — no server-held context is implied by the protocol; state must be carried or looked up.
5. Trust boundary — the line where you stop believing input — the point where data crosses from an environment you do not control into one you do.

35.2 HTML: the structure of a page

35.2.1 PLAIN - in simple words

1. HTML stands for HyperText Markup Language.
2. A **markup language** is a way of adding labels to text to say what each part of the text is.
3. It does not say what the text should look like. It says what the text means.
4. You mark up a piece of text by wrapping it in a tag. A tag is a word inside angle brackets.
5. `<p>Hello</p>` says: the word Hello is a paragraph.
6. Most tags come in pairs. An opening tag, then the content, then a closing tag with a slash.
7. The opening tag, the content and the closing tag together are called an **element**.
8. Tags can carry extra information called **attributes**, written as `name="value"` inside the opening tag.
9. Elements go inside other elements. That is **nesting**, and it makes the page a tree, like folders inside folders.
10. HTML is not a programming language. It cannot add two numbers, make a decision, or repeat something. It only describes structure.
11. That is not an insult. Describing structure well is the whole job, and most accessibility and search problems come from doing it badly.

35.2.2 PLAIN - a picture in your head

1. Think of a printed book being prepared before typesetting.
2. An editor goes through the manuscript with a pencil and writes in the margin: this line is a chapter title, this block is a quotation, this is a footnote, this is a list.
3. The editor does not choose the font. The editor does not choose the size. Those come later, from the designer.
4. The editor's marks are the markup. They record what each piece of text *is*.
5. A different designer can take the same marked-up manuscript and produce a paperback, a large-print edition, or an audiobook script.
6. Because the marks say "chapter title", the audiobook narrator knows to pause and change tone. The word "chapter title" carries meaning that "big bold text" does not.

Where this comparison breaks:

7. The editor's marks are read by a human who can use judgement. HTML is read by a machine that will do exactly and only what the tags say.
8. A book is finished once. An HTML page is re-laid-out every time the window size changes, on screens the author never saw.
9. And HTML is more forgiving than any editor would be. Browsers repair broken markup silently, which hides mistakes rather than reporting them.

35.2.3 PLAIN - a worked example

1. Here is a complete, valid page. Everything in it earns its place.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport"
        content="width=device-width, initial-scale=1">
  <title>Bookshop - Contact</title>
  <link rel="stylesheet" href="/style.css">
</head>
<body>
  <header>
    <h1>Ambleside Bookshop</h1>
    <nav aria-label="Main">
      <a href="/">Home</a>
      <a href="/contact" aria-current="page">Contact</a>
    </nav>
  </header>

  <main>
    <h2>Send us a message</h2>
    <form action="/contact" method="post">
      <label for="name">Your name</label>
      <input id="name" name="name" type="text" required>

      <label for="email">Email</label>
      <input id="email" name="email" type="email" required>

      <label for="msg">Message</label>
      <textarea id="msg" name="msg" rows="5"></textarea>

      <button type="submit">Send</button>
    </form>
  </main>

  <footer>
    <p>Open 9am to 6pm, closed Sunday.</p>
  </footer>
</body>
</html>
```

2. Line by line, the parts that people get wrong.
3. `<!DOCTYPE html>` is not a tag. It is a switch. Without it, browsers fall into "quirks mode" and copy 1990s bugs on purpose.
4. `lang="en"` tells screen readers which language to pronounce, and tells search engines who this page is for.
5. `<meta charset="utf-8">` says the bytes are UTF-8, from Chapter 7. Leave it out and names with accents may turn into rubbish.
6. The viewport meta tag tells a phone not to pretend it is a 980-pixel desktop. Without it, your responsive CSS does nothing on mobile.
7. `<label for="name">` is joined to `<input id="name">` by the matching value. Now clicking the label focuses the box, and a screen reader announces the right words.
8. `name="name"` is what the server receives. `id` is for the page. They are different jobs that beginners merge.
9. `type="email"` makes phone keyboards show the at sign, and lets the browser check the shape for free.

35.2.4 PLAIN - what is really happening inside

1. The browser receives the file as a stream of bytes, not as a page.
2. It decodes those bytes into characters using the declared encoding, then splits the characters into tokens: start tag, end tag, text, comment.
3. It builds a tree from those tokens. Each element becomes a node with a parent and a list of children. That tree is the DOM, which is section 35.5.
4. **Semantic elements** are tags whose names describe the role of the content: `header`, `nav`, `main`, `article`, `section`, `aside`, `footer`, `h1` to `h6`, `figure`, `time`.
5. They matter for two audiences you cannot see.
6. Screen readers build a list of headings and landmarks from them, so a blind user can jump straight to the main content instead of hearing the menu on every page.
7. Search engines use them to work out what the page is about and which part is the real content.
8. A `<div>` says nothing. It is a plain box. A page built only from divs looks identical and is much harder to use without sight.
9. Forms are the one place plain HTML can send data. A form with `method="post"` packages the named fields and sends them to the server on its own, with no JavaScript at all.
10. The browser gives you validation, keyboard handling, autofill and mobile keyboards for free, but only if you use real form elements.

35.2.5 TECHNICAL - the engineer's version

1. HTML was created by Tim Berners-Lee at CERN; the first public description circulated in 1991. HTML 2.0 was standardized as RFC 1866 in November 1995.
2. HTML 4.01 became a W3C Recommendation in December 1999. XHTML 1.0 followed in 2000 and pushed strict XML syntax, which the web rejected.
3. WHATWG formed in 2004 in response. HTML5 became a W3C Recommendation on 28 October 2014. Since a 2019 agreement, the WHATWG HTML Living Standard is the single normative specification. There are no numbered versions now.
4. The parsing algorithm is fully specified, including error recovery. Two conforming browsers must build the same DOM from the same broken input. This is a standard, not a convention, and it is unusual in being so.
5. Elements are void (`img`, `br`, `input`, `meta`), raw text (`script`, `style`), or normal. Void elements have no closing tag and no children.
6. Attribute values should be quoted. Unquoted values are legal in HTML5 but break on spaces. Quoting is a convention with a good reason.
7. ARIA (Accessible Rich Internet Applications, WAI-ARIA 1.2, W3C Recommendation June 2023) adds roles and states for widgets HTML lacks.
8. The first rule of ARIA, stated in the specification's own authoring practices, is not to use ARIA when a native element exists. A real `<button>` beats `<div role="button" tabindex="0">` on every measure.

Input type	Added in	Mobile keyboard	Free validation
text	HTML 2.0	standard	none
email	HTML5	at sign	shape check
number	HTML5	digits	min, max, step
date	HTML5	date picker	range
tel	HTML5	phone pad	none

9. Tools: the W3C Nu HTML Checker (`validator.w3.org/nu`), `axe` DevTools, and the accessibility tree view in Chrome and Firefox DevTools.

35.2.6 WORDS - remember these

1. Markup — labels added to text — a syntax that annotates content with structural and semantic roles rather than presentation.
2. Element — a tag plus its content — a node in the document tree with a tag name, attribute set and child node list.
3. Attribute — extra detail on a tag — a name-value pair on an element, exposed in the DOM as content attributes and IDL properties.
4. Semantic HTML — tags that say what content is — markup chosen so the accessibility tree and machine readers infer correct roles.
5. Void element — a tag with no closing half — an element defined as having no end tag and no permitted children, such as `img` or `input`.
6. Doctype — the switch that turns on modern rules — a legacy string that selects no-quirks mode in the HTML parser.

35.3 CSS: presentation, and the cascade that confuses everyone

35.3.1 PLAIN - in simple words

1. CSS stands for Cascading Style Sheets.
2. HTML says what things are. CSS says what they look like.
3. Keeping those two apart is the idea called **separation of content and presentation**.
4. The benefit is real. Change one stylesheet and every page changes. Give the same HTML a different stylesheet and you get a different design.
5. A CSS rule has two halves. A **selector** picking which elements to affect, and a block of **declarations** saying what to do to them.
6. A declaration is a property and a value: `color: black;`.
7. Many rules can hit the same element. When two rules disagree, the browser has to pick one. The picking procedure is called the **cascade**.
8. The cascade is where almost everyone gets stuck, so we will do it properly and give you the real calculation.
9. Every element is drawn as a rectangle. Understanding that rectangle, the box model, removes most layout confusion.
10. Two layout systems, flexbox and grid, do nearly all modern layout. Learn those two well and you can stop guessing.

35.3.2 PLAIN - a picture in your head

1. Imagine a school with a stack of rulebooks about uniform.
2. The national rule says shoes must be black. The school rule says shoes must be brown. The class teacher's note says shoes must be blue.
3. A pupil arrives. Which colour wins?
4. You need an agreed order for settling arguments, or the pupil cannot dress.
5. The school's order is: a note about *this named pupil* beats a note about *this class*, which beats a note about *all pupils*.
6. If two notes are equally specific, the one written most recently wins.
7. And a rule marked "no exceptions, headmaster" beats everything, which is why the headmaster should use it almost never.
8. That is the cascade. Origin, then importance, then specificity, then order.

Where this comparison breaks:

9. Rules about pupils apply to whole pupils. CSS settles each property separately. Colour can come from one rule and size from another, in the same element, at the same time.

10. And some CSS properties are inherited by children automatically, like text colour, while others are not, like border. No school rule works that way.

35.3.3 PLAIN – a worked example

1. Here is a real box, measured.

```
.card {
  width: 300px;
  padding: 20px;
  border: 2px solid black;
  margin: 16px;
  box-sizing: content-box;
}
```

2. With `content-box`, which is the original CSS behaviour, `width` means the content area only.
3. Total horizontal space the box occupies on screen:

Part	Left	Right	Total
content	-	-	300
padding	20	20	40
border	2	2	4
margin	16	16	32

4. Drawn width is $300 + 40 + 4 = 344$ px. Space consumed in the flow, including margin, is 376 px.
5. That surprise is why almost every project now writes:

```
*, *::before, *::after { box-sizing: border-box; }
```

6. With `border-box`, `width: 300px` means the border edge is 300 px, so the content area shrinks to $300 - 40 - 4 = 256$ px. What you type is what you measure.
7. Now specificity, calculated properly. Specificity is a triple: (A, B, C).

Selector	A ids	B classes	C elements
`p`	0	0	1
`.note`	0	1	0
`p.note`	0	1	1
`#main p.note`	1	1	1
`:is(#a, p)`	1	0	0

8. Compare left to right. (1,0,0) beats (0,9,9). A single id beats nine classes. The columns are not digits of one number, they are separate ranks.
9. Attribute selectors and pseudo-classes count in column B. Pseudo-elements such as `::before` count in column C. The universal selector * counts zero.
10. `:where(...)` always contributes zero, which makes it the correct tool for library defaults you want authors to override easily.

35.3.4 PLAIN – what is really happening inside

1. The browser collects every declaration that could apply to one element and one property, then sorts them by these steps in order.
2. Step one, origin and importance. Order from weakest to strongest: browser defaults, then user settings, then your stylesheet, then your `!important` rules, then user `!important`, then browser `!important`.

3. Note the flip. Among `!important` rules the order reverses, so a user's accessibility override can beat an author's. That is deliberate.
4. Step two, since 2023, cascade layers declared with `@layer`. Later layers beat earlier ones, and this is decided before specificity.
5. Step three, specificity, using the (A, B, C) triple above.
6. Step four, source order. If everything else ties, the declaration that appears later in the stylesheet wins.
7. Inline `style="..."` is not a selector. It sits above all normal author rules and is only beaten by `!important`.
8. If no rule sets a property, the element may still get a value by **inheritance** from its parent. Text properties inherit. Box properties do not.
9. If nothing inherits either, the property takes its **initial value** from the specification.
10. The honest version: people say "the most specific rule wins". That is only step three of six, and it is not even the first tiebreaker. The full order is origin, importance, layer, specificity, then source order.

35.3.5 TECHNICAL - the engineer's version

1. CSS was proposed by Hakon Wium Lie on 10 October 1994 at CERN, with Bert Bos. CSS Level 1 became a W3C Recommendation on 17 December 1996. CSS 2.1 became a Recommendation in June 2011.
2. There is no "CSS3" specification. Since Level 2 the language is split into independently versioned modules: Selectors Level 4, Grid Level 2, and so on. Saying "CSS3" is a marketing habit, not a standard.
3. Display types: `block`, `inline`, `inline-block`, `flex`, `grid`, `none`, and the two-value syntax `display: inline flex`. `none` removes the element from layout entirely; `visibility: hidden` keeps its space.
4. Flexbox is one-dimensional: items are distributed along a main axis. It reached W3C Candidate Recommendation status in 2012 and shipped broadly from 2013.

```
.toolbar {
  display: flex;
  justify-content: space-between;
  align-items: center;
  gap: 12px;
  flex-wrap: wrap;
}
.toolbar .spacer { flex: 1 1 auto; }
```

5. Grid is two-dimensional: rows and columns declared together. It shipped in Chrome 57 and Firefox 52 in March 2017, which is the date modern CSS layout really begins.

```
.page {
  display: grid;
  grid-template-columns: 240px minmax(0, 1fr);
  grid-template-rows: auto 1fr auto;
  gap: 16px;
  min-height: 100vh;
}
```

6. The `fr` unit means one share of the leftover space. `minmax(0, 1fr)` is the standard fix for grid children that refuse to shrink below their content.
7. Positioning: `static` (default), `relative` (offset from its own place, keeps its space), `absolute` (removed from flow, positioned against the nearest positioned ancestor), `fixed` (against the viewport), `sticky` (relative until a scroll threshold, then fixed).

Unit	Relative to	Typical use
px	device-independent pixel	borders, hairlines
em	own font-size	spacing inside a component

Unit	Relative to	Typical use
rem	root font-size (16 px)	type scale, layout
%	parent's same dimension	fluid widths
vh / vw	1% of viewport	full-screen sections
ch	width of "0" glyph	text measure limits

8. One CSS px is defined as 1/96 inch at a nominal viewing distance, not one hardware pixel. On a 3x phone screen, one CSS px covers nine device pixels.
9. Media queries adapt to the environment:

```
@media (min-width: 48rem) { .page { --cols: 2; } }
@media (prefers-reduced-motion: reduce) {
  * { animation-duration: 0.01ms !important; }
}
```

10. Mobile-first means writing the small-screen rules first and using `min-width` queries to add complexity. It is a convention, and a good one.
11. Custom properties, usually called CSS variables, are real cascading values, live at runtime, and are readable by JavaScript. They shipped in 2016.

```
:root { --gap: 12px; --ink: #111; }
.card { padding: var(--gap); color: var(--ink); }
```

12. Container queries let a component respond to its own container rather than the viewport. They reached cross-browser support in February 2023.

```
.sidebar { container-type: inline-size; }
@container (min-width: 30rem) {
  .card { display: grid; grid-template-columns: 1fr 2fr; }
}
```

13. Native CSS nesting reached cross-browser support during 2023 and 2024 and removes one of the main historical reasons to use Sass.
14. `:has()`, the long-requested parent selector, shipped across browsers by December 2023. `.card:has(img)` styles a card because of what it contains.
15. Tools: DevTools Styles pane shows the winning declaration and strikes out losers; the Layout pane overlays grid and flex lines; `@supports` queries let you ship progressive enhancement safely.

35.3.6 WORDS - remember these

1. Cascade — the argument-settling procedure — the ordered algorithm of origin, importance, layer, specificity and source order.
2. Specificity — how targeted a selector is — the (id, class, type) triple compared left to right, never summed.
3. Box model — every element is a rectangle — content, padding, border and margin boxes, with `box-sizing` choosing what `width` measures.
4. Flexbox — line things up along one direction — CSS Flexible Box Layout, one-dimensional distribution along a main and cross axis.
5. Grid — a real table-free grid — CSS Grid Layout, two-dimensional placement with explicit and implicit tracks.
6. rem — a size tied to the page's base text — the computed font-size of the root element, 16 px unless changed.
7. Container query — react to my own box, not the window — `@container` rules evaluated against a queried ancestor's size.

35.4 JavaScript: the language and its event loop

35.4.1 PLAIN - in simple words

1. JavaScript is the programming language that runs inside web browsers.
2. Unlike HTML and CSS, it really is a programming language. It can decide, repeat, calculate, store and react.
3. It was written in 1995 at a company called Netscape, by one person, Brendan Eich, in about ten days.
4. Ten days is not enough time to design a language carefully. Many of its odd corners come straight from that deadline.
5. It was never removed, because the web could not break old pages. So the world's most used language is one that was rushed and then frozen.
6. Since 1997 the language has had a written standard, and it now gets a new edition every June, so it improves without breaking what came before.
7. The single hardest idea in JavaScript is **asynchrony**: starting something slow, not waiting for it, and dealing with the answer when it arrives.
8. Asynchrony exists because JavaScript in a browser has only one worker. If that worker waits, the whole page freezes: no scrolling, no clicking.
9. So instead of waiting, it writes a note saying what to do when the answer comes, and gets on with other work.
10. The machine that manages those notes is called the **event loop**, and it is the thing you must understand to write correct JavaScript.

35.4.2 PLAIN - a picture in your head

1. Picture a single cook in a small kitchen with a strict rule: finish the task in your hands before starting the next one.
2. An order arrives: boil rice, twenty minutes. The cook does not stand and watch the pot. That would block every other order.
3. The cook puts the rice on, sets a timer, writes "when the timer rings, drain the rice" on a slip, and puts the slip in a tray.
4. Then the cook takes the next order and starts chopping.
5. When the cook finishes a task and has nothing in hand, they look in the tray, take the oldest slip, and do it.
6. That is the event loop. One cook, no waiting, a tray of pending slips.
7. There is a second, smaller tray, right next to the cook, checked before the big one and emptied completely each time.
8. Anything urgent, like "the sauce reduced, add salt now", goes in the small tray and is done before the next big order.
9. The small tray is the microtask queue. The big tray is the task queue.

Where this comparison breaks:

10. A real cook can glance at the pot while chopping. JavaScript truly cannot. If a function loops for three seconds, nothing else happens for three seconds, including the timer that was due one second ago.
11. And a timer set for 100 ms does not mean the slip is done at 100 ms. It means the slip is put in the tray at 100 ms. If the cook is busy, it waits.

35.4.3 PLAIN - a worked example

1. Predict the output of this. Most people get it wrong the first time.

```
console.log("1");

setTimeout(() => console.log("2"), 0);
```

```
Promise.resolve().then(() => console.log("3"));

queueMicrotask(() => console.log("4"));

console.log("5");
```

2. The answer is 1, 5, 3, 4, 2.
3. Here is why, step by step.
4. The whole script is itself one task. It runs to the end before anything else is considered. So "1" prints, then "5" prints.
5. Along the way, `setTimeout` handed a callback to the timer system with a delay of zero. When zero milliseconds pass, that callback is placed in the **task queue**.
6. `Promise.resolve().then(...)` placed its callback in the **microtask queue** immediately, because the promise was already settled.
7. `queueMicrotask` placed its callback in the microtask queue too, after the promise one.
8. The script ends. The call stack is now empty.
9. Before touching the task queue, the engine drains the microtask queue completely. So "3" prints, then "4".
10. Only now does it take one item from the task queue. "2" prints.

Moment	Stack	Microtasks	Tasks
script running	script	3, 4	(timer pending)
script ends	empty	3, 4	2
drain micro	3 then 4	empty	2
next task	2	empty	empty

11. The rule to memorize: microtasks run to exhaustion after every task, before the next task, and before the browser gets a chance to paint.
12. That gives you a real bug. A microtask that queues another microtask forever starves the page. The browser never repaints. Nothing on screen moves.

35.4.4 PLAIN – what is really happening inside

1. The **call stack** is a pile of function calls. Calling a function pushes a frame on. Returning pops it off. When the pile is empty, the engine is idle.
2. Slow things are not done by JavaScript at all. Network requests, timers, disk access and user input are handled by the browser's own code, in other threads, written in C++ or Rust.
3. When one of those finishes, the browser does not interrupt your code. It places your callback in a queue.
4. The event loop is a simple repeating check: is the stack empty? If yes, drain all microtasks, then take one task, run it, then possibly render.
5. Promises, `async/await`, `queueMicrotask` and MutationObserver use the microtask queue. Timers, network events, clicks and messages use task queues.
6. `await` is not magic. It splits your function in two at that point. The rest of the function becomes a microtask attached to the awaited promise.
7. Rendering happens between tasks, not during one. That is why a long loop freezes the screen even if you change styles inside it.
8. Three ways to write the same wait, in historical order.

```
// 1. Callbacks (1995 onward)
getUser(id, (err, user) => {
  if (err) return done(err);
  getOrders(user, (err, orders) => {
```

```
    if (err) return done(err);
    done(null, orders);
  });
});

// 2. Promises (standardized in ES2015)
getUser(id)
  .then(user => getOrders(user))
  .then(orders => done(null, orders))
  .catch(done);

// 3. async / await (ES2017)
try {
  const user = await getUser(id);
  const orders = await getOrders(user);
  return orders;
} catch (err) {
  handle(err);
}
```

9. All three do the same thing. The third is easiest to read and easiest to get right, because errors travel through ordinary `try` and `catch`.
10. The trap in version three: two sequential `await` calls that do not depend on each other waste time. Use `Promise.all` to run them together.

```
const [user, rates] = await Promise.all([
  getUser(id),
  getRates(),
]);
```

35.4.5 TECHNICAL – the engineer's version

1. Brendan Eich wrote the first prototype in May 1995 at Netscape Communications. It was called Mocha, renamed LiveScript in September 1995, then JavaScript in December 1995 as part of a marketing deal with Sun.
2. The name is the source of endless confusion. JavaScript and Java are unrelated languages.
3. Netscape submitted it to Ecma International; ECMA-262 1st edition was published in June 1997. The standard language is called **ECMAScript**; JavaScript is a trademarked implementation name.
4. ES5 (December 2009) added strict mode and JSON. ES2015, also called ES6, was the large modernization: `let`, `const`, arrow functions, classes, modules, promises, template literals, destructuring.
5. Since 2015 there is one edition each June. ECMAScript 2025 was the 16th edition (June 2025) and ECMAScript 2026 the 17th (June 2026).
6. Types: `number` (IEEE 754 double, so integers are exact only to 2^53 - 1), `bigint`, `string` (UTF-16 code units), `boolean`, `undefined`, `symbol`, `null`, and `object`.
7. Coercion is the automatic conversion between types, and it is the source of the famous oddities.

Expression	Result	Reason
`0.1 + 0.2`	0.30000000000000004	binary floats
`[] + {}`	"[object Object]"	both to string
`"5" - 2`	3	minus forces number
`"5" + 2`	"52"	plus prefers string
`NaN === NaN`	false	IEEE 754 rule
`typeof null`	"object"	1995 bug, frozen

8. `0.1 + 0.2` is not a JavaScript flaw. It is IEEE 754 binary floating point, from Chapter 7, and it happens in Python, Java and C too.
9. Use `===` not `==`. `==` applies the coercion table and produces surprises such as `"" == 0` being true.
10. **Closures**: a function keeps a live reference to the variables of the scope where it was defined, even after that scope returns.

```
function counter() {
  let n = 0;
  return () => ++n;
}
const next = counter();
next(); // 1
next(); // 2
```

11. JavaScript uses **prototypal inheritance**. Every object has a hidden link to another object; property lookup walks that chain. `class` syntax, added in ES2015, is sugar over prototypes, not a separate system.
12. Modules: ES modules (`import`, `export`) are the standard, defined in ES2015 and shipped in browsers from 2017. CommonJS (`require`, `module.exports`) is Node's older system and is still everywhere.
13. Engines: V8 (Chrome, Node, Edge, Deno), SpiderMonkey (Firefox), JavaScriptCore (Safari, Bun). All are JIT compilers with multiple tiers, described in Chapter 16.
14. Error handling rules that matter: never swallow an error silently; attach a `.catch` or `try/catch` to every promise chain; an unhandled rejection terminates a Node process by default since Node 15 (2020).
15. Tools: `node --prof`, Chrome DevTools Performance panel for long tasks, `performance.now()` for sub-millisecond timing.

35.4.6 WORDS - remember these

1. Event loop — the one-worker queue system — the algorithm that drains microtasks and dispatches one task per turn on a single thread.
2. Call stack — the pile of calls in progress — the LIFO structure of execution contexts; overflow raises RangeError.
3. Microtask — an urgent note done before anything else — a job queued to the microtask queue, drained fully after each task.
4. Promise — a receipt for a future value — an object in pending, fulfilled or rejected state, with `then` scheduling microtasks.
5. Closure — a function that remembers where it was born — a function object plus a reference to its defining lexical environment.
6. Prototype — the object I fall back to — the internal `[[Prototype]]` link traversed during property lookup.
7. ECMAScript — the written standard — ECMA-262, published annually each June since 2015; JavaScript is an implementation of it.

35.5 The DOM and how a browser turns HTML into pixels

35.5.1 PLAIN - in simple words

1. HTML arrives as text. Text cannot be clicked, moved or measured.
2. So the browser converts the text into a live tree of objects held in memory. That tree is the **DOM**, the Document Object Model.
3. Every tag becomes a node. Every node knows its parent and its children.
4. JavaScript does not edit your HTML file. It edits this tree. Change the tree and the screen follows.
5. Your CSS is turned into a second structure with the same idea, called the CSSOM.

6. The browser combines the two into a **render tree**: only the things that will actually be drawn, each with its final style.
7. Then it works out where every box goes, which is **layout**. Then it fills in the colours, which is **paint**. Then it stacks the layers, which is **composite**.
8. If you change something that affects size or position, the browser must redo layout. That is called **reflow** and it is the expensive one.
9. If you change only a colour, it can skip layout and just repaint. Cheaper.
10. If you change only a transform or opacity, it can often skip both and just re-stack existing layers. Cheapest of all.
11. Knowing which of the three you triggered is most of frontend performance.

35.5.2 PLAIN – a picture in your head

1. Think of a newspaper being made in a print room, before computers.
2. The reporters' typed pages are the HTML. Just text with notes on it.
3. The editor's style guide, saying headlines are 36 point and captions are italic, is the CSS.
4. The layout artist takes both and builds a paste-up board: which story goes in which column, how many lines it runs to, where the picture sits. That is layout.
5. Then the ink goes on: black type, grey photographs. That is paint.
6. Then the transparent overlay sheets are stacked in order. That is composite.
7. Now the editor says one caption should be bold. The word gets wider. The line rewraps. The column gets longer. The story below moves down. The whole board must be redone.
8. That is a reflow, and you can see why it costs.
9. But if the editor only says "print this headline in a darker grey", the shapes do not move. Only the ink changes. That is a repaint.

Where this comparison breaks:

10. A newspaper is laid out once. A browser may redo layout sixty times a second while you drag a window.
11. And in a print room the artist works on the whole board. Browsers do partial invalidation: they mark a subtree as dirty and try to redo only that. The cost still spreads upward more often than you would like.

35.5.3 PLAIN – a worked example

1. Here is the classic layout-thrashing bug, in ten lines.

```
const items = document.querySelectorAll(".row");

// BAD: read, write, read, write, ...
for (const el of items) {
  el.style.height = el.offsetHeight + 10 + "px";
}
```

2. `offsetHeight` is a **read** of a geometric value. `style.height` is a **write** that invalidates geometry.
3. The browser batches writes and flushes them lazily. But a read forces it to flush immediately, so the answer is correct. That forced flush is a reflow.
4. With 200 rows, that loop causes 200 forced synchronous layouts. On a mid phone that can be 200 ms of frozen page.
5. The fix is to separate all reads from all writes.

```
const items = document.querySelectorAll(".row");

// GOOD: read everything, then write everything
const heights = [...items].map(el => el.offsetHeight);
items.forEach((el, i) => {
```

```
  el.style.height = heights[i] + 10 + "px";
});
```

6. Now there is one layout pass, not 200. Same output, roughly one hundredth of the cost.
7. Properties that force a synchronous layout when read include `offsetTop`, `offsetHeight`, `scrollTop`, `clientWidth`, `getBoundingClientRect()` and `getComputedStyle()`.
8. Now events. Suppose a list has 500 rows and each needs a click handler.

```
// 500 listeners, 500 objects, slow to attach
rows.forEach(r => r.addEventListener("click", onClick));

// 1 listener, using bubbling. This is delegation.
list.addEventListener("click", (e) => {
  const row = e.target.closest(".row");
  if (row) onClick(row.dataset.id);
});
```

9. This works because a click on a row also fires on its parents, in order, outward. That travelling is **bubbling**, and using it deliberately is **event delegation**.
10. Delegation also handles rows added later, which the first version cannot.

35.5.4 PLAIN – what is really happening inside

1. The full sequence, in order, for a first page load.

```
bytes -> characters -> tokens -> nodes -> DOM tree
CSS bytes ------------------------------> CSSOM tree
DOM + CSSOM -> render tree -> layout -> paint -> composite
```

2. Parsing is incremental. The browser starts building the DOM before the whole file has arrived, which is why a slow page still shows its header early.
3. A plain `<script>` tag in the middle of the body stops parsing dead. The browser must fetch and run it before continuing, because the script might write into the document.
4. `defer` says: fetch it now in parallel, run it after parsing finishes, in order. `async` says: fetch now, run the moment it arrives, order not guaranteed. Use `defer` for application code.
5. CSS is **render-blocking**. The browser will not paint until it has the stylesheets, because painting with the wrong styles then correcting would flash. That flash has a name, FOUC.
6. Layout computes a box for every rendered node: x, y, width, height, in a single tree walk, honouring the box model and the layout mode of each parent.
7. Paint turns boxes into a list of drawing commands: fill this rectangle, draw this glyph run, clip here.
8. Composite runs those command lists into layers, often on the GPU from Chapter 22, and stacks them. Scrolling and CSS transforms usually only touch this stage, which is why they are smooth.
9. An event has three phases: capture, travelling down from the document to the target; target; then bubble, travelling back up. Most code only ever uses the bubble phase.
10. `stopPropagation()` halts travel. `preventDefault()` cancels the browser's built-in reaction, such as following a link. They are different and beginners confuse them constantly.

35.5.5 TECHNICAL – the engineer's version

1. DOM Level 1 became a W3C Recommendation in October 1998. The DOM is now a WHATWG Living Standard, and it is a language-independent API, not a JavaScript feature.
2. The **critical rendering path** is the minimum set of resources needed for first paint: the HTML, the render-blocking CSS, and any parser-blocking JavaScript.
3. Rendering engines in production: Blink (Chrome, Edge, Opera, forked from WebKit on 3 April 2013), WebKit (Safari), Gecko (Firefox).

4. Style resolution is per element per property; matching selectors right to left is the standard implementation trick that makes descendant selectors cheap enough.
5. Reflow cost is superlinear in tree depth and node count. Modern engines do partial layout, but a change to a flow-root ancestor still invalidates its whole subtree.

Change	Triggers	Relative cost
`width`, `top`, font-size	layout, paint, composite	highest
`background-color`, `color`	paint, composite	medium
`transform`, `opacity`	composite only	lowest
adding a class with all three	full pipeline	highest

6. Animate `transform` and `opacity`. Animating `left` and `width` forces layout on every frame and will not hold 60 fps on a mid-range phone.
7. A frame budget at 60 Hz is 16.7 ms; at 120 Hz it is 8.3 ms. Chapters 21 and 22 traced that budget from key press to photons.
8. `requestAnimationFrame` schedules a callback just before the next paint, which is the correct hook for animation. `setTimeout(fn, 16)` is not.
9. `content-visibility: auto` and `contain` let you tell the engine that a subtree cannot affect the outside, allowing it to skip work entirely.
10. The DOM API is genuinely slow relative to plain objects, mostly because each call crosses from the JavaScript engine into the C++ engine. Batch changes with `DocumentFragment` or by building a string once.
11. Tools: Performance panel with the Frames and Main tracks, the "Layout Shift Regions" and "Paint flashing" overlays in Rendering, and the `PerformanceObserver` API for `long-animation-frame` entries.

35.5.6 WORDS - remember these

1. DOM — a live tree of the page in memory — the Document Object Model, a language-neutral object representation defined by WHATWG.
2. CSSOM — the same idea for styles — the parsed stylesheet object model used during style resolution.
3. Render tree — only the things that get drawn — the tree of styled boxes, excluding `display: none` and including generated content.
4. Reflow — redo the positions — layout invalidation and recomputation of box geometry; also called relayout.
5. Repaint — redo the colours — rasterization of the display list without a geometry pass.
6. Composite — stack the finished layers — GPU-assisted assembly of rasterized layers into the final frame.
7. Bubbling — the event travels outward — phase 3 of DOM event dispatch, from target up to the root.
8. Delegation — one listener for many children — attaching a handler to a common ancestor and using `event.target`.

35.6 The browser as a platform

35.6.1 PLAIN - in simple words

1. A browser is not a document viewer any more. It is an operating system for web pages.
2. It gives your page a set of built-in services, the same way an operating system gives a program files, memory and the network.
3. It can store data on the user's disk in three different ways, for different sizes and purposes.
4. It can fetch more data from a server at any time, without loading a new page.
5. It can run your code on extra threads, so slow work does not freeze the screen.

6. It can keep a small program running in the background that answers requests even with no network, which is how a web app works offline.
7. It can hold a two-way conversation with a server, where the server speaks first, instead of only answering questions.
8. It can connect two users' browsers directly to each other, for calls and video.
9. It can draw arbitrary graphics and run code on the graphics chip.
10. And it does all this inside a **sandbox**: a locked room where the page can touch only what it has been given, and must ask permission for the rest.

35.6.2 PLAIN - a picture in your head

1. Think of a serviced office you rent by the hour.
2. You get a desk, a locked drawer, a phone and a window. That is the platform: storage, network, screen.
3. You cannot walk into another tenant's office. The doors are locked and the locks are not yours to change. That is the same-origin rule.
4. You cannot open the fire door to the roof without asking the building manager first, and the manager asks you in front of the tenant. That is the permission prompt for camera, microphone and location.
5. There are two small back rooms you can hire someone to work in, out of sight of your desk. One does heavy calculation. One sits by the post slot and answers deliveries even when you have gone home.
6. Those are web workers and service workers.

Where this comparison breaks:

7. An office tenant can carry things down the corridor by hand. A page cannot reach another origin's storage at all, by any route, including through the user's own copy of the browser.
8. And a real building manager can be persuaded. The sandbox is enforced by the browser's own code, and cannot be talked around by the page.

35.6.3 PLAIN - a worked example

1. A small task: fetch a list, cache it, and survive going offline.

```
// 1. Ask the server for JSON
const res = await fetch("/api/books", {
  headers: { "Accept": "application/json" },
});
if (!res.ok) throw new Error("HTTP " + res.status);
const books = await res.json();

// 2. Keep a copy the user can read while offline
localStorage.setItem("books", JSON.stringify(books));
```

2. Note `res.ok`. `fetch` does not reject on a 404 or a 500. It only rejects on a network failure. Forgetting this is one of the most common bugs.
3. Now the storage choices, with real limits.

Store	Holds	Typical limit	Sent to server
Cookie	small strings	4 KB each	yes, every request
localStorage	strings	about 5 MB	no
sessionStorage	strings, per tab	about 5 MB	no
IndexedDB	structured objects	large, quota-based	no
Cache Storage	whole responses	quota-based	no

4. localStorage is synchronous. A 2 MB read blocks the single thread, and the page freezes for that time. Do not use it for anything large.

5. IndexedDB is asynchronous and can hold hundreds of megabytes, subject to a quota that is usually a share of free disk space.
6. Cookies travel with every matching request automatically. That is why they suit session identifiers and suit nothing else.
7. A minimal service worker turns that fetch into an offline-capable one.

```
// sw.js
self.addEventListener("fetch", (event) => {
  event.respondWith(
    caches.match(event.request).then(hit =>
      hit || fetch(event.request)
    )
  );
});
```

8. Now the browser asks the service worker before it asks the network. If the worker has a copy, the page loads with the aeroplane mode switch on.

35.6.4 PLAIN – what is really happening inside

1. The **same-origin policy** is the core rule. An origin is the triple of scheme, host and port. `https://a.com` and `https://a.com:8443` are different origins, and so are `http` and `https` versions of one host.
2. Code from one origin cannot read the DOM, cookies or storage of another. It can still cause requests to another origin, which is why CSRF exists.
3. CORS is the controlled exception. The *server* adds a response header saying which origins may read its answers. The browser enforces it. The rule lives with the server, not the page.
4. A **web worker** is a separate thread with its own memory and no access to the DOM. You talk to it by sending messages that are copied, not shared.
5. Use one for parsing a large file, image processing, or cryptography, so the main thread stays free to draw at 60 frames per second.
6. A **service worker** is a special worker that sits between the page and the network, like a programmable proxy on the user's own machine.
7. It survives page closes, wakes for push messages, and must be served over HTTPS, because a hostile one would be permanent.
8. **WebSockets** solve a real problem. Plain HTTP is ask-and-answer, so a server with news cannot speak first.
9. The old workaround was polling: ask every few seconds, usually to be told nothing happened. Wasteful in requests, battery and latency.
10. A WebSocket starts life as an ordinary HTTP request with an Upgrade header, then the same TCP connection becomes a two-way message pipe that stays open.
11. **WebRTC** goes further and connects two browsers directly, so audio and video do not pass through your server. Your server only helps them find each other.

35.6.5 TECHNICAL – the engineer's version

1. Same-origin policy dates to Netscape Navigator 2.02 in 1995. CORS is defined in the WHATWG Fetch Standard. Preflight uses an `OPTIONS` request when the method or headers are non-simple.
2. `fetch()` shipped from 2015 and replaced `XMLHttpRequest`, which came from Microsoft Outlook Web Access around 1999 and reached IE 5 in 1999. The term AJAX was coined by Jesse James Garrett in February 2005.
3. WebSocket is RFC 6455, December 2011. Handshake is HTTP `Upgrade: websocket`; frames then carry a 2 to 14 byte header, versus roughly 500 to 800 bytes of HTTP headers per polled request.

Technique	Direction	Overhead	Best for
Polling	client asks	high, constant	rare updates
Long polling	client waits	medium	legacy fallback
SSE	server to client	low, text only	feeds, progress
WebSocket	both ways	lowest per message	chat, games
WebRTC data	peer to peer	lowest latency	calls, screen share

4. Server-Sent Events use the `text/event-stream` MIME type, reconnect automatically, and are part of the HTML Living Standard. They are one-way and much simpler to operate than WebSockets.
5. WebRTC was open-sourced by Google in May 2011 and became a W3C Recommendation on 26 January 2021. It needs STUN and usually TURN servers to traverse NAT, which Chapter 24 described. TURN relays cost real bandwidth.
6. Graphics: Canvas 2D is immediate-mode drawing. WebGL 1.0 shipped March 2011 and is based on OpenGL ES 2.0; WebGL 2.0 (2017) on OpenGL ES 3.0. WebGPU shipped in Chrome 113 in May 2023, in Safari 26 in 2025, and gives compute shaders and a modern binding model.
7. Storage quota is not fixed by specification. Chrome allows an origin up to about 60 percent of total disk in practice; Safari applies stricter eviction. This is an implementation detail and changes between versions.
8. Permissions requiring an explicit user prompt: camera, microphone, geolocation, notifications, clipboard read, MIDI, Bluetooth, USB, and persistent storage. Most also require a secure context (HTTPS or localhost).
9. Powerful features are gated three ways: secure context, transient user activation (a real click within a few seconds), and Permissions Policy headers set by the site.
10. Tools: DevTools Application panel shows every store, service worker state, and quota usage; `chrome://webrtc-internals` shows live peer connections.

35.6.6 WORDS - remember these

1. Sandbox — a locked room for the page — an isolated execution environment with mediated access to host resources.
2. Origin — the identity of a site — the tuple of scheme, host and port used for all isolation decisions.
3. CORS — the server's note saying who may read this — Cross-Origin Resource Sharing headers defined by the Fetch Standard.
4. Service worker — a proxy of your own on the user's machine — an event-driven worker intercepting fetches, enabling offline and push.
5. Web worker — a second thread with no screen access — a dedicated worker with an isolated global scope and message-passing only.
6. WebSocket — a phone line instead of letters — RFC 6455 full-duplex framed messaging over a single upgraded TCP connection.
7. IndexedDB — a real database in the browser — a transactional object store with indexes and an asynchronous API.

35.7 Frontend frameworks, explained rather than listed

35.7.1 PLAIN - in simple words

1. Here is the problem a framework exists to solve.
2. Your page shows information. That information changes. The page must change to match, everywhere it appears, every time.
3. Doing that by hand means writing instructions: find this element, change its text, add this row, remove that one, toggle this class.
4. That style is called **imperative**: you say the steps.

5. With ten pieces of information it is fine. With two hundred, in a page where any change can affect any other, it becomes impossible to keep straight.
6. A framework lets you write **declarative** code instead: you describe what the page should look like for the current data, and the framework works out the steps to get there.
7. The second big idea is the **component**: one self-contained piece of the page with its own markup, styling and behaviour, that you can use many times.
8. A page becomes a tree of components, the way a house is a set of rooms rather than one giant space.
9. That is all a framework is: a way to say "screen equals function of data", plus a way to cut a screen into reusable pieces.

35.7.2 PLAIN - a picture in your head

1. Imagine a departures board at a station, made of physical letter tiles.
2. The imperative way: a person with a ladder, told "change platform 4 from 12 to 9, and change the 14:05 to Delayed, and remove the 13:50 row".
3. Every instruction must be right, in the right order, or the board lies.
4. The declarative way: you hand over a printed sheet showing what the board should say now. A machine compares it with what the board says and changes only the tiles that differ.
5. You never think about tiles. You think about the correct board.
6. The comparing step is called **reconciliation**, and the sheet you hand over is the idea behind a **virtual DOM**.

Where this comparison breaks:

7. The station board has fixed rows. A web page has a shifting tree, so the comparison needs hints about identity. That is what the `key` attribute is for, and getting keys wrong causes real bugs.
8. And comparison is not free. Some frameworks skip it entirely by working out at build time exactly which tiles a given change touches.

35.7.3 PLAIN - a worked example

1. The same counter, three ways.

```
// Imperative: plain DOM
let n = 0;
btn.addEventListener("click", () => {
  n = n + 1;
  out.textContent = "Count: " + n;
});
```

```
// Declarative: React
function Counter() {
  const [n, setN] = useState(0);
  return (
    <button onClick={() => setN(n + 1)}>
      Count: {n}
    </button>
  );
}
```

```
<!-- Declarative: Svelte 5 -->
<script>
  let n = $state(0);
</script>
<button onclick={() => n++}>Count: {n}</button>
```

2. In the first, you own the update. In the other two, you own the description and the tool owns the update.

3. With one counter the first is shorter. With a form of forty fields that feed a summary, a chart and a save button, the first becomes a maze.
4. What React does when `setN` runs: it calls `Counter()` again, gets a new description, compares it with the old one, sees only the text differs, and changes that one text node. It does not rebuild the button.
5. What Svelte does: at build time the compiler already worked out that `n` appears in exactly one text position, so it emits code that assigns to that text node directly. There is no comparison at runtime.

35.7.4 PLAIN - what is really happening inside

1. **State** is the data that can change. **Props** are the data a parent hands to a child. A component re-renders when its own state or its props change.
2. A **virtual DOM** is a plain tree of ordinary JavaScript objects describing the wanted output. Cheap to build, cheap to throw away.
3. Reconciliation walks the old tree and the new tree together. Same type at the same position means update in place. Different type means destroy and rebuild. Lists are matched by `key`.
4. Using an array index as a key is the classic bug: delete the first row and every key shifts, so the framework updates the wrong rows and typed input jumps to the wrong place.
5. **State management** is the question of where shared data lives. Local state inside a component is easy. Data that four distant components need is not.
6. The escalation ladder is: local state, then lift state to a common parent, then a context or provider, then a dedicated store, then a server-state library that caches responses.
7. **Hydration** happens when HTML is rendered on the server for speed, then the same components run in the browser to attach event handlers to the existing markup. The page looks ready before it is interactive.
8. The honest version: hydration is a workaround, not a triumph. It ships the data twice, once as HTML and once as JSON, and it is why some server-rendered pages feel dead for a second. Islands, partial hydration and React Server Components are all attempts to send less.

35.7.5 TECHNICAL - the engineer's version

1. React was created at Facebook by Jordan Walke and open-sourced in May 2013 at JSConf US. Facebook renamed itself Meta in October 2021. React 19 is current, with 19.2 published in 2026.
2. JSX is a syntax extension compiled to function calls; it is not HTML and not part of the language. `class` becomes `className` because `class` is reserved.
3. React uses one-way data flow: data goes down through props, changes go up through callbacks. Hooks arrived in React 16.8 (February 2019) and replaced class components as the normal style.
4. The core hooks are `useState`, `useEffect`, `useMemo`, `useCallback`, `useRef` and `useContext`. Rules: only at the top level, only in components or other hooks, because hook identity is positional.
5. `useEffect` is for synchronizing with something outside React, not for deriving data. Overuse of effects is the most common source of React bugs.

Framework	First release	Approach	Trade-off
React	May 2013, Meta	virtual DOM, hooks	huge ecosystem, more rerenders
Angular	Sept 2016 (v2)	full framework, DI	batteries included, steep start
Vue	Feb 2014, Evan You	reactive proxies	gentle curve, smaller job market
Svelte	Nov 2016, R. Harris	compiler, runes	least runtime code, smaller ecosystem
Solid	2018, R. Carniato	fine-grained signals	very fast, small community

6. AngularJS (October 2010, Misko Hevery at Google) and Angular 2 onward (14 September 2016) are different frameworks with a shared name. Angular uses TypeScript, RxJS and dependency injection, and now ships version 22.
7. Vue 3 (September 2020) uses ES Proxy-based reactivity and a composition API. Svelte 5 (19 October 2024) introduced runes, replacing its label-based reactivity. Solid uses signals with no component rerenders at all.
8. Signals, which Solid popularized, have now been adopted by Angular, Vue, Svelte and Preact, and a TC39 signals proposal exists. Experts disagree on whether React should adopt them; React's team argues its compiler achieves the same result without changing the model.
9. Meta-frameworks add routing, server rendering and data loading: Next.js for React (version 16 in 2026), Nuxt for Vue, SvelteKit, Angular with its own SSR, Remix and Astro.
10. When you do not need a framework: content sites, marketing pages, blogs, documentation, forms with a handful of fields, and anything where the server can render the HTML and a hundred lines of plain JavaScript will do.
11. A framework costs 40 to 150 KB of JavaScript before your own code, plus a build step, plus upgrade work forever. Spend that only when interactivity genuinely earns it.

35.7.6 WORDS - remember these

1. Declarative — describe the result — express the target state and delegate the transition to the runtime.
2. Component — one reusable piece of screen — an encapsulated unit of template, state and behaviour with a props interface.
3. Virtual DOM — a cheap paper copy of the page — an in-memory tree diffed against the previous render to compute minimal DOM mutations.
4. Reconciliation — comparing old and new — the diffing algorithm, keyed by element type and `key` attribute.
5. Hydration — waking up server HTML — attaching client event listeners and state to markup already present in the document.
6. Signal — a value that knows who reads it — a fine-grained reactive primitive that updates only its dependents.

35.8 Build tooling: why frontend code needs a build step

35.8.1 PLAIN - in simple words

1. Twenty years ago you wrote a `.js` file and put it on a server. Done.
2. Today most projects run a program over the code first, and ship the result. That program is the **build step**.
3. There are four honest reasons for it.
4. One, your code is split across hundreds of small files, and loading hundreds of files is slow. So they are joined into a few. That is **bundling**.
5. Two, you write modern syntax, and some browsers do not understand all of it. So it is rewritten into older syntax. That is **transpiling**.
6. Three, some files are not JavaScript at all: JSX, TypeScript, Svelte components, Sass. Something must convert them.
7. Four, shipping fewer bytes makes pages faster, so names are shortened, spaces removed and unused code deleted.
8. Nothing here is magic. The build step is a translator and a packer.

35.8.2 PLAIN – a picture in your head

1. Think of moving house.
2. Your belongings are spread through many rooms in many small containers. That is your source code: many files, each doing one thing, easy for you to find.
3. You would not carry two hundred small boxes to the van one at a time. You pack them into a few large cartons. That is bundling.
4. Anything broken or never used gets thrown out rather than moved. That is tree shaking.
5. Air is squeezed out and clothes are vacuum-packed. That is minification.
6. And you keep an inventory sheet saying which carton each item is in, so that when something is missing at the far end you can trace it back. That is the source map.

Where this comparison breaks:

7. When you move house, you unpack. The browser never unpacks: it runs the packed form. The source map only exists for you, in the debugger.
8. And you can add a room after moving in. Bundles are fixed at build time, which is why code splitting has to be planned, not discovered.

35.8.3 PLAIN – a worked example

1. A tiny `package.json`, which is the project's identity card.

```
{
  "name": "bookshop-web",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "test": "vitest run"
  },
  "dependencies": { "react": "^19.2.0" },
  "devDependencies": { "vite": "^8.2.0" }
}
```

2. `dependencies` are needed to run. `devDependencies` are needed only to build and test, and never reach the browser.
3. `^19.2.0` means "19.2.0 or any later 19.x". That range is why two people can install the same project on the same day and get different code.
4. The **lock file**, `package-lock.json` or `pnpm-lock.yaml`, records the exact version and content hash of every package that was actually installed.
5. Commit the lock file. Without it your builds are not reproducible, and a working build today can break tomorrow with no change from you.
6. Tree shaking, shown concretely:

```
// utils.js exports 40 functions
import { formatDate } from "./utils.js";
```

7. Because ES modules declare their imports statically, the bundler can prove the other 39 are unreachable and drop them. CommonJS `require` is dynamic, so it usually cannot.
8. Code splitting turns one big file into several loaded on demand:

```
const Chart = lazy(() => import("./Chart.jsx"));
```

9. Now the chart's code downloads the first time a user opens the chart, not on every page load.

35.8.4 PLAIN – what is really happening inside

1. A bundler starts at an entry file, follows every import, and builds a graph of modules. Then it writes that graph out as one or more files.
2. A transpiler parses source into a syntax tree, rewrites nodes, and prints new source. Babel does this for JavaScript syntax.
3. Transpiling handles *syntax*. Missing *functions* need a **polyfill**, which is real code implementing the missing feature. These are different jobs.
4. TypeScript adds a type layer on top of JavaScript. It checks your code, then erases every type annotation. Nothing is checked at runtime.
5. That erasure is the key point people miss. TypeScript protects you while you write. It gives you nothing at all against bad data arriving from a network request. You still have to validate.
6. Minification renames local variables to single letters, removes whitespace and comments, and simplifies expressions. Typical saving is 30 to 60 percent before compression.
7. A source map is a separate file mapping every position in the output back to a line and column in your source, so the debugger can lie helpfully.
8. `node_modules` is where installed packages live, honestly: a very large folder containing your dependencies and all of their dependencies, often tens of thousands of files and hundreds of megabytes.
9. It is not a virus and it is not waste by design. It is the visible cost of a culture of very small packages. It should never be committed to git, and it is always rebuildable from `package.json` plus the lock file.

35.8.5 TECHNICAL – the engineer's version

1. npm launched in January 2010, written by Isaac Schlueter, and is now operated by GitHub. The registry holds well over three million packages.
2. Alternatives: Yarn (2016), pnpm (2017, which hard-links a global store and cuts disk use sharply), and Bun's built-in installer (2022).
3. Semantic versioning is MAJOR.MINOR.PATCH. `^1.2.3` allows minor and patch updates, `~1.2.3` allows patch only, and an exact pin allows neither. This is a convention, and packages break it regularly.

Tool	First release	Written in	Role
webpack	2012, T. Koppers	JavaScript	bundler, loaders
Rollup	2015, R. Harris	JavaScript	library bundler
Babel	2014, S. McKenzie	JavaScript	transpiler
esbuild	2020, E. Wallace	Go	very fast bundler
Vite	2020, Evan You	JS plus Rust	dev server, build
SWC	2019	Rust	transpiler, minifier

4. Vite's development server does not bundle. It serves native ES modules and transforms each file on request, so start-up is near instant regardless of project size. For production it bundles with Rollup.
5. Speed differences are large and measurable: esbuild and SWC are commonly 10 to 100 times faster than Babel and webpack on the same input, because they are compiled languages with parallelism rather than single-threaded JavaScript.
6. TypeScript was released by Microsoft on 1 October 2012, designed by Anders Hejlsberg. TypeScript 7, a native port of the compiler to Go, shipped in 2026 with roughly a tenfold type-checking speed-up.
7. Supply chain risk is real. Compromised packages have shipped malware to millions of installs. Mitigations: lock files, `npm audit`, `npm ci` in CI, `--ignore-scripts`, and provenance attestations.
8. Bundle budgets that hold up in practice: under 100 KB of compressed JavaScript for a content site, under 300 KB for a rich application. Measure compressed transfer size, not the raw file size.
9. Tools: `npx vite build --sourcemap`, `rollup-plugin-visualizer`, Chrome DevTools Coverage panel to find unused bytes, `npm ls <pkg>` to find who pulled a package in.

35.8.6 WORDS - remember these

1. Bundler — the packer — a tool that resolves a module graph and emits fewer, larger output files.
2. Transpile — rewrite new syntax as old — source-to-source compilation between language versions.
3. Polyfill — supply a missing function — runtime code implementing a standard API absent in the host.
4. Tree shaking — drop code nobody uses — dead-code elimination over a static ES module graph.
5. Source map — the inventory sheet for debugging — a JSON mapping from generated positions back to original source.
6. Lock file — the exact recipe of what was installed — a resolved dependency tree with versions and integrity hashes.
7. Code splitting — download it when needed — emitting separate chunks loaded by dynamic `import()`.

35.9 The backend: what a server program actually does

35.9.1 PLAIN - in simple words

1. A server program is an ordinary program with one unusual habit: it never finishes.
2. It opens a port, which is a numbered door on the machine, and waits.
3. When a request arrives, it reads it, decides what it means, does the work, writes an answer, and goes back to waiting.
4. Deciding what a request means is **routing**: matching the method and path, such as `GET /books/12`, to the piece of code that handles it.
5. The piece of code that handles it is called a **handler**, or in some traditions a **controller**.
6. Work that must happen on many requests, such as logging, checking who the user is, or parsing the body, is factored out into **middleware**: small steps a request passes through on the way in.
7. There are two shapes of backend. One builds finished HTML pages and sends them. The other sends only data, usually JSON, and lets the frontend draw.
8. The first is templating. The second is API-only. Both are fine, and mixing them is normal.

35.9.2 PLAIN - a picture in your head

1. Think of a hospital reception.
2. The door is the port. Anyone may walk in.
3. Before anyone sees a doctor they pass a fixed sequence: security check, then registration, then the file is pulled, then vitals are taken. Every patient, every time, in that order. That is middleware.
4. Reception then reads the complaint and sends the patient to the right department. That is routing.
5. The specialist who actually treats them is the handler.
6. The discharge letter handed back is the response.
7. If registration fails, the patient never reaches the specialist, and is sent away at that step. That is middleware returning early, which is exactly how authentication is enforced.

Where this comparison breaks:

8. A hospital treats patients one at a time per doctor. A server holds thousands of half-finished requests at once, interleaved.
9. And a hospital remembers you between visits. HTTP does not. Every arrival is a stranger until a cookie or token proves otherwise.

35.9.3 PLAIN - a worked example

1. The same tiny API, in Node with Express and in Python with FastAPI.

```
// server.js  -  Node with Express 5
import express from "express";
const app = express();
```

```
app.use(express.json());          // middleware
app.use((req, res, next) => {     // logging
  console.log(req.method, req.url);
  next();
});

app.get("/books/:id", async (req, res) => {
  const book = await db.findBook(req.params.id);
  if (!book) return res.status(404).json({ error: "no" });
  res.json(book);
});

app.listen(3000);
```

```
# main.py  -  Python with FastAPI
from fastapi import FastAPI, HTTPException

app = FastAPI()

@app.get("/books/{book_id}")
async def read_book(book_id: int):
    book = await db.find_book(book_id)
    if book is None:
        raise HTTPException(status_code=404)
    return book
```

2. The shapes are the same: declare a path, name the variable part, do the work, return data or an error code.
3. FastAPI reads the `book_id: int` annotation and rejects `/books/abc` with a 422 before your function runs. Express does not; you must check yourself.
4. The request lifecycle, drawn:

```
TCP accept
  -> parse HTTP request line and headers
  -> middleware 1 (parse body)
  -> middleware 2 (log)
  -> middleware 3 (authenticate) -- may stop here
  -> router match: GET /books/:id
  -> handler runs, queries database
  -> serialize response, set status and headers
  -> write bytes, maybe keep connection alive
```

35.9.4 PLAIN - what is really happening inside

1. Under the routing there is a socket, from Chapter 28. `listen` puts the process into the accepting state; the kernel queues incoming connections.
2. Two families of design handle concurrency.
3. **Thread or process per request**: each request gets its own worker, which is allowed to block. Simple to reason about. Costs memory per worker, roughly 0.5 to 8 MB of stack, so thousands of workers is expensive.
4. **Event loop**: one thread handles many connections by never blocking, using the same idea as the browser event loop in section 35.4. Cheap per connection, but one slow synchronous function stalls everyone.
5. Node.js uses the second. So do FastAPI with async handlers, Go's runtime with very cheap goroutines, and modern Java with virtual threads.
6. This is why a Node server is excellent for many connections doing little work each, such as an API or a chat gateway, and a poor choice for heavy number-crunching in the request path.

7. The way out for Node is not to fight it: move heavy work to a worker thread, a separate service, or a background job queue.
8. In every design, the database is usually the real bottleneck, not the language. A 40 ms query dwarfs a 0.4 ms difference in framework overhead.

35.9.5 TECHNICAL – the engineer's version

1. Node.js was released by Ryan Dahl in 2009 and presented at JSConf EU in November 2009. It pairs V8 with libuv for asynchronous input and output. Node 24 "Krypton" is the active LTS as of 2026; Node 26 became the current line in May 2026.
2. Node is single-threaded for JavaScript but not single-threaded overall: libuv keeps a thread pool, four by default, for file system work, DNS and some crypto.
3. Express began in 2010 by TJ Holowaychuk; Express 5.0 was released in 2024 after a very long wait. Fastify and Hono are faster modern alternatives. Deno (2018) and Bun (2022) are competing runtimes.

Stack	Since	Concurrency	Strong at
Node.js	2009	event loop	APIs, realtime, one language
Django	2005	threads, async	admin, ORM, batteries
FastAPI	Dec 2018	async, ASGI	typed APIs, auto docs
Spring Boot	Apr 2014	threads, virtual	large enterprise systems
Go	2009, 1.0 2012	goroutines	throughput, single binary
Rails	July 2004	threads	fast product iteration
Laravel	June 2011	processes	shared hosting, PHP shops
ASP.NET Core	June 2016	async tasks	Windows shops, strong tooling

4. Django was created at the Lawrence Journal-World by Adrian Holovaty and Simon Willison and released publicly in July 2005; Django 6.1 is current. FastAPI was released by Sebastian Ramirez in December 2018.
5. Ruby on Rails was extracted from Basecamp by David Heinemeier Hansson in July 2004 and popularized convention over configuration. Rails 8.1 is current in 2026.
6. Go was announced by Google on 10 November 2009 (Robert Griesemer, Rob Pike, Ken Thompson) with 1.0 in March 2012; Go 1.26 is current. A goroutine starts at about 2 KB of stack, versus about 1 MB for an operating system thread.
7. Spring Framework began in 2003 from Rod Johnson's book; Spring Boot arrived in April 2014 and Spring Boot 4.1 is current in 2026. Java 25, released 16 September 2025, is the newest long-term-support release.
8. PHP dates from 1995 (Rasmus Lerdorf); PHP 8.5 arrived November 2025. Laravel was released by Taylor Otwell in June 2011 and is now at version 13.
9. .NET Core 1.0 shipped June 2016 and unified with .NET 5 in November 2020; .NET 10, released 11 November 2025, is the current long-term-support version.
10. Honest guidance: pick the language your team already knows. Every framework in that table serves millions of users somewhere. The differences that matter are hiring, libraries and operational familiarity, not benchmarks.
11. Tools: `curl -i`, `httpie`, `ab` and `wrk` for load, `strace` or `dtruss` for syscalls, and `clinic.js` or `pprof` for profiling.

35.9.6 WORDS - remember these

1. Route — the address-to-code map — a match on HTTP method and path pattern dispatching to a handler.
2. Middleware — a step every request passes through — a composable function in the request pipeline that may short-circuit the response.
3. Handler — the code that answers — the terminal function producing a status, headers and body; also called a controller action.
4. Templating — the server builds the HTML — server-side rendering of markup from data plus a template language.
5. Event loop server — one worker, never waiting — non-blocking I/O multiplexed with epoll, kqueue or IOCP through libuv or similar.
6. Port — a numbered door on a machine — a 16-bit TCP or UDP endpoint identifier bound by a listening socket.

35.10 Databases

35.10.1 PLAIN - in simple words

1. Programs forget everything when they stop. A database is the place data goes so it survives, and so many users can share it safely.
2. You could use files. People do, and then discover the hard parts: two users writing at once, finding one record among ten million, and a crash halfway through an update.
3. A database solves exactly those three problems: concurrency, fast search, and never being left half-done.
4. The oldest and still best default is the **relational** database: data stored as tables of rows and columns, queried with a language called SQL.
5. A **table** is one kind of thing, such as books. A **row** is one of them. A **column** is one fact about it.
6. A **primary key** is the column that uniquely names a row. A **foreign key** is a column in one table holding the key of a row in another, which is how tables are joined together.
7. **NoSQL** is a loose family of databases that gave up some of those rules for speed, scale or a friendlier shape. Sometimes that is the right trade. Often it is not.
8. If you have no strong reason otherwise, start with PostgreSQL. This is not fashion; it is the advice most experienced engineers give.

35.10.2 PLAIN - a picture in your head

1. Think of a library's paper records before computers.
2. One drawer holds a card per book: title, author number, year. Another drawer holds a card per author: number, name, country.
3. The author's name is written once, in the author drawer. The book card only stores the author's number.
4. That is **normalization**, and the reason for it is simple: if an author's name is wrong, you fix one card, not four hundred.
5. Looking up a book by title is fast because the drawer is sorted and you can jump to the right divider. That sorted arrangement is an **index**.
6. Without an index you would read every card in the drawer. With one you touch about twenty. That gap is the whole reason indexes exist.
7. And when a book moves between branches, both the "removed from A" and "added to B" cards must change together, or the book vanishes. Doing both or neither is a **transaction**.

Where this comparison breaks:

8. Library cards do not lie to each other. Two clerks writing at the same moment can produce a state neither intended, which is why isolation levels exist and why they are subtle.

9. And a card drawer holds thousands. A real index handles billions with roughly the same number of steps, because it is a tree, not a list.

35.10.3 PLAIN – a worked example

1. Start with a badly designed table. One row per sale.

id	book	author	author_country	qty
1	Dune	F. Herbert	US	2
2	Emma	J. Austen	UK	1
3	Dune	F. Herbert	US	3

2. Three faults. The author's country is repeated; if it is wrong it is wrong in many places. You cannot record an author with no sales. And updating one row can leave the data disagreeing with itself.
3. Normalized into three tables:

```
CREATE TABLE authors (
  id      SERIAL PRIMARY KEY,
  name    TEXT NOT NULL,
  country CHAR(2) NOT NULL
);

CREATE TABLE books (
  id        SERIAL PRIMARY KEY,
  title     TEXT NOT NULL,
  author_id INT NOT NULL REFERENCES authors(id)
);

CREATE TABLE sales (
  id      SERIAL PRIMARY KEY,
  book_id INT NOT NULL REFERENCES books(id),
  qty     INT NOT NULL CHECK (qty > 0),
  sold_at TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

4. Now a real query: total copies sold per country, best first, ignoring countries with fewer than 10 sales.

```
SELECT a.country,
       SUM(s.qty) AS copies
FROM sales s
JOIN books   b ON b.id = s.book_id
JOIN authors a ON a.id = b.author_id
WHERE s.sold_at >= '2026-01-01'
GROUP BY a.country
HAVING SUM(s.qty) >= 10
ORDER BY copies DESC
LIMIT 5;
```

5. `JOIN` follows the foreign keys to stitch the three tables back together. `GROUP BY` collapses many rows into one per country. `WHERE` filters rows before grouping; `HAVING` filters groups after. That difference is examined constantly and understood rarely.
6. Now indexes, with real arithmetic. A `sales` table with 10 million rows.
7. Without an index, finding rows for one `book_id` reads all 10,000,000 rows. At perhaps 2 million rows per second in memory, that is about 5 seconds.
8. With a B-tree index, the rows are held in a sorted tree with a high branching factor. If each node holds about 200 keys, then 200^3 is 8 million and 200^4 is 1.6 billion.

9. So four levels cover the table. The lookup reads four pages instead of ten million. That is roughly a million times less work, and it is why the answer comes back in under a millisecond.

```
CREATE INDEX idx_sales_book ON sales (book_id, sold_at);
EXPLAIN ANALYZE
SELECT * FROM sales WHERE book_id = 42;
```

10. `EXPLAIN` prints the plan the database intends to use. `Seq Scan` means it is reading everything; `Index Scan` means it is using the tree. Adding `ANALYZE` actually runs it and reports real timings and row counts.
11. The single most useful debugging habit in backend work is comparing the planner's estimated rows with the actual rows. A large gap means the statistics are stale, and the plan is probably bad.

35.10.4 PLAIN - what is really happening inside

1. **ACID**, one letter at a time.
2. **A, Atomicity.** A transaction happens completely or not at all. Money leaving one account and arriving in another either both happen or neither does. A crash in between leaves no half-state.
3. **C, Consistency.** The database never ends a transaction in a state that breaks the rules you declared: foreign keys, uniqueness, checks.
4. **I, Isolation.** Concurrent transactions do not see each other's unfinished work. How strictly is a setting, and that is the interesting part.
5. **D, Durability.** Once the database says "committed", the data survives an immediate power cut, because it was written to a log on disk and flushed.
6. Isolation levels, weakest to strongest, and the anomaly each one stops:

Level	Prevents	Still allows
Read uncommitted	nothing	dirty reads
Read committed	dirty reads	non-repeatable reads
Repeatable read	non-repeatable reads	phantoms (in theory)
Serializable	all of the above	nothing, but retries

7. A **dirty read** is seeing another transaction's uncommitted change. A **non-repeatable read** is reading the same row twice and getting different values. A **phantom** is running the same query twice and getting a new row.
8. PostgreSQL defaults to read committed and never permits dirty reads at all. Its repeatable read also prevents phantoms, because it uses snapshots. MySQL's InnoDB defaults to repeatable read. These are implementation details worth checking for your engine.
9. **Connection pooling.** Opening a database connection is expensive: a TCP handshake, authentication, and on PostgreSQL a new operating system process, costing several megabytes.
10. So the application keeps a small set of open connections and lends them out. A pool of 10 to 30 usually serves far more concurrent users than that, because each request holds a connection for only milliseconds.
11. The classic production failure is one pool per instance times many instances, exhausting the server's connection limit. PgBouncer exists to sit in the middle and fix exactly that.

35.10.5 TECHNICAL - the engineer's version

1. The relational model comes from Edgar F. Codd's 1970 paper at IBM, "A Relational Model of Data for Large Shared Data Banks". SEQUEL was designed by Donald Chamberlin and Raymond Boyce in 1974 and renamed SQL.
2. SQL was standardized by ANSI in 1986 and ISO in 1987; the current edition is SQL:2023. Every engine adds extensions, so portable SQL is a discipline.

3. B-trees come from Rudolf Bayer and Edward McCreight in 1970, published 1972. Databases use B+ trees, where all values sit in the leaves and leaves are linked, making range scans cheap.
4. ACID as a concept is Jim Gray's, 1981; the acronym was coined by Theo Harder and Andreas Reuter in 1983.
5. PostgreSQL descends from POSTGRES, begun by Michael Stonebraker at Berkeley in 1986; Postgres95 added SQL, and the name PostgreSQL dates from 1996. PostgreSQL 18 was released on 25 September 2025.

Type	Example	Shape	Good at
Relational	PostgreSQL 18	tables, SQL	almost everything
Document	MongoDB 8.3	JSON documents	flexible schema
Key-value	Redis 8.10	key to value	cache, counters, queues
Wide-column	Cassandra 5	partitioned rows	huge write volume
Graph	Neo4j	nodes and edges	relationship traversal
Search	Elasticsearch	inverted index	full-text ranking
Time-series	TimescaleDB	time-partitioned	metrics, sensors

6. Origins: MongoDB from 10gen in 2009 (Dwight Merriman, Eliot Horowitz); Redis in 2009 by Salvatore Sanfilippo; Cassandra at Facebook in 2008, Apache in 2010; Neo4j in 2007; Elasticsearch in 2010 by Shay Banon.
7. Redis changed licence in March 2024, which produced the Valkey fork under the Linux Foundation, then adopted AGPLv3 with Redis 8 in May 2025. If you use a managed cache, check which one you are actually running.
8. **CAP theorem**, stated correctly. It was conjectured by Eric Brewer in a July 2000 keynote and proved by Seth Gilbert and Nancy Lynch in 2002.
9. It says: when a network **partition** occurs, a distributed system must choose between **consistency** (every read sees the latest write) and **availability** (every request gets a non-error answer).
10. The common misuse is to describe systems as "CP" or "AP" as a permanent personality, or to say you may "pick two of three". You do not choose partitions; the network chooses them for you. C versus A is a choice made only during a partition, often per operation.
11. Daniel Abadi's PACELC (2012) is the honest extension: else, when there is no partition, you still trade latency against consistency.
12. Five scenarios, and the honest answer for each.

Scenario	Choose	Why
Payments, orders	PostgreSQL	transactions decide correctness
Session cache	Redis	sub-ms reads, expiry built in
Product search	Elasticsearch	ranking, typo tolerance
Device telemetry	Timescale	time partitioning, retention
Social graph paths	Neo4j	multi-hop traversal

13. Note what is missing: no scenario says "MongoDB because we do not know the schema yet". PostgreSQL has a `jsonb` column type with indexing, so you can keep flexible documents inside a relational database and still have joins and transactions when you need them.
14. Tools: `psql`, `EXPLAIN (ANALYZE, BUFFERS)`, `pg_stat_statements` for the slowest queries, `pgbench` for load, and the slow query log in MySQL.

35.10.6 WORDS - remember these

1. Primary key — the unique name of a row — a column set with a uniqueness and not-null constraint identifying each tuple.
2. Foreign key — a pointer to another table's row — a referential integrity constraint enforced on insert, update and delete.
3. Normalization — store each fact once — decomposing relations to remove update anomalies, usually to third normal form.
4. Index — the sorted shortcut — an auxiliary B+ tree or hash structure mapping key values to row locations.
5. Transaction — all or nothing — a unit of work bounded by BEGIN and COMMIT with ACID guarantees.
6. Isolation level — how much of others' work you may see — the setting choosing which concurrency anomalies are permitted.
7. CAP theorem — under a network split, pick consistency or answers — during a partition a system cannot be both linearizable and available.
8. Connection pool — a reused set of open lines — a bounded cache of database connections shared across requests.

35.11 ORMs and data access

35.11.1 PLAIN - in simple words

1. Your program thinks in objects. The database thinks in tables. Something has to translate.
2. An **ORM**, an Object-Relational Mapper, is a library that does that translation for you.
3. You write `user.orders` in your language and it writes SQL, runs it, and turns the rows back into objects.
4. The gain is real: less repetitive code, fewer typos, and safety against SQL injection because values are sent separately from the query text.
5. The cost is also real: it is easy to write one innocent line that causes hundreds of database round trips.
6. An ORM usually brings **migrations** too: small, ordered, versioned scripts that change the shape of the database and are applied in the same order on every machine.

35.11.2 PLAIN - a picture in your head

1. An ORM is a translator at a meeting between two people who share no language.
2. For ordinary sentences the translation is faithful and saves everyone effort.
3. But if you ask an ambiguous question, the translator will produce something grammatical and wrong, and you will not notice because you cannot read the other language.
4. So a good engineer learns enough of the other language to check the translation, even when using the translator daily.

Where this comparison breaks:

5. A human translator gets tired. An ORM is perfectly consistent, which means a bad pattern is repeated identically ten thousand times a second.

35.11.3 PLAIN - a worked example

1. The **N+1 query problem**, the single most common performance bug in backend work.

```
orders = Order.objects.filter(status="paid")   # 1 query
for o in orders:
    print(o.customer.name)   # 1 query per order
```

2. With 200 orders that is $1 + 200 = 201$ round trips. At 2 ms each, that is about 0.4 seconds of pure waiting, for data one query could fetch.
3. The fix is to tell the ORM to fetch the related rows in advance:

```
orders = (Order.objects
          .filter(status="paid")
          .select_related("customer"))   # 1 query, joined
```

4. Now it is one query with a JOIN. Roughly 2 ms instead of 400 ms.
5. The same fix has different names in different tools: `select_related` and `prefetch_related` in Django, `include` in Prisma, `joinedload` in SQLAlchemy, `includes` in Rails, `with` in Laravel.
6. The way to catch it is to log queries per request in development. If one page view issues 300 queries, you have found it.

35.11.4 PLAIN - what is really happening inside

1. The ORM keeps a map from classes to tables and from attributes to columns, plus rules for relationships.
2. Accessing a relationship attribute that has not been loaded triggers a **lazy load**: a fresh query, issued right there in the loop. That is where N+1 comes from.
3. Many ORMs also keep an identity map and a unit of work, so the same row read twice gives the same object and changes are written in one flush at the end.
4. Migrations work by recording which scripts have run in a table inside the database itself. On deploy, the tool compares that list with the files on disk and applies the missing ones in order.
5. Never edit an applied migration. Write a new one. The applied list is the shared history and rewriting it desynchronizes environments.
6. Raw SQL is better when the query is the point: complex reporting, window functions, recursive queries, bulk updates, and anything where you need to control the plan.
7. Modern practice is to use the ORM for the ordinary 90 percent and drop to SQL for the 10 percent that matters, rather than choosing one forever.

35.11.5 TECHNICAL - the engineer's version

1. The term object-relational impedance mismatch describes the structural gap: inheritance, identity and associations have no direct relational equivalent.
2. Notable tools and dates: Hibernate for Java (2001, Gavin King), Rails ActiveRecord (2004), Django ORM (2005), SQLAlchemy (2006, Michael Bayer), Entity Framework (2008), Eloquent (2011), Prisma (2019, now version 7), Drizzle (2022).
3. Two design patterns, from Martin Fowler's 2002 book Patterns of Enterprise Application Architecture: Active Record, where a row and its behaviour are one object, and Data Mapper, where they are separate.
4. Query builders such as Knex, jOOQ and Drizzle sit between raw SQL and a full ORM: typed composition, no object graph, no lazy loading, so no N+1.
5. Parameterized queries prevent SQL injection because the driver sends the query text and the values on separate paths, so a value can never become syntax. String concatenation into SQL is the vulnerability.
6. Migration tools: Alembic, Django migrations, Flyway, Liquibase, Prisma Migrate, Rails Active Record migrations, EF Core migrations.
7. Zero-downtime schema change is a separate discipline: add a nullable column, backfill in batches, start writing both, switch reads, then drop the old column, across several deploys. Never rename in one step.
8. Tools: `django-debug-toolbar`, SQLAlchemy `echo=True`, Prisma's query event log, and `pg_stat_statements` to find the query issued most often rather than the slowest single one.

35.11.6 WORDS - remember these

1. ORM — a translator between objects and tables — a mapping layer generating SQL and materializing rows as domain objects.
2. N+1 — one query, then one more per result — repeated lazy loads inside an iteration, fixed by eager loading or a join.

3. Migration — a versioned change to the schema — an ordered, recorded script applied idempotently across environments.
4. Lazy loading — fetch it only when touched — deferred query execution on first access to an unloaded association.
5. Parameterized query — values sent apart from the text — placeholders bound by the driver, preventing SQL injection by construction.

35.12 Authentication and sessions

35.12.1 PLAIN - in simple words

1. **Authentication** is proving who you are. **Authorization** is deciding what you are allowed to do once you are known.
2. They are two different questions, they fail in two different ways, and mixing them up is one of the most common security bugs there is.
3. A site must never keep your password the way you typed it. If a site can email your old password back to you, it stored it wrongly.
4. Instead it stores a scrambled fingerprint of the password, produced by a one-way function called a **hash**.
5. One-way means you can go from password to fingerprint easily, and from fingerprint back to password essentially never.
6. It also mixes in a random extra value called a **salt**, different for every single user, so two people with the same password get different fingerprints.
7. And the hash used for passwords is made deliberately slow. That sounds mad, and it is the entire point.
8. Once you have logged in, the site has to remember you on the next click, because HTTP forgets everything between one request and the next.
9. There are two ways to remember. Keep a note on the server and hand you a ticket number: that is a **session**. Or hand you a sealed pass that you carry and the server does not store: that is a **token**.
10. When you click "sign in with Google", you never give Google's password to the site you are visiting. A protocol called OAuth 2.0 arranges that carefully.

35.12.2 PLAIN - a picture in your head

1. Think of a cloakroom at a theatre, and a wristband at a festival.
2. A session is the cloakroom ticket. Your coat stays with the cloakroom staff and you carry nothing but a number.
3. The number means nothing on its own. Only the staff's book turns number 47 into "the long grey coat belonging to that person".
4. If you lose the ticket, the staff can cross out number 47 in the book instantly, and it stops working everywhere, at once.
5. A token is the festival wristband. Your name is printed on it and it carries a tamper-proof seal that any gate can check without phoning anyone.
6. That makes gates fast and independent. Ten gates can check ten thousand wristbands with no shared book at all.
7. But if you are thrown out of the festival, the wristband is still on your wrist. It keeps working until it expires, unless every gate is handed a list of banned wristbands - and a shared list is a cloakroom book again.

Where this comparison breaks:

8. A photograph of a wristband gets you nowhere at a real gate, because a human looks at your face. A copied token is not a picture of you. To the server it simply is you, completely, until the moment it expires.

9. A real cloakroom holds a few hundred coats in one room. A session store holds millions of tickets across many servers, which is why real systems keep it in Redis or a database rather than in one server's memory.

35.12.3 PLAIN - a worked example

1. Here is a bcrypt password record, the kind that sits in a `users` table. It is one string that carries four things at once.

```
$2b$12$KIXqf8vN1sT.u2WcQ7pOyeR3hZ...

 $2b$   the algorithm and version (bcrypt)
 12$    the cost factor: 2^12 = 4096 key rounds
 KIX... the first 22 characters are the salt
 R3h... the rest is the hash itself
```

2. Notice there is no separate salt column. The salt travels inside the record, because it is not a secret. It only has to be unique and unpredictable.
3. Checking a login runs the same function again with the stored salt and cost, then compares the results. It never reverses anything.
4. The cost factor is a power of two, so each step up doubles the work. Rough times on one ordinary 2025 server core:

Cost	Rounds	Time per hash
8	256	about 6 ms
10	1,024	about 25 ms
12	4,096	about 100 ms
14	16,384	about 400 ms

5. Why slow is the goal: an attacker who steals the table wants to try billions of guesses. At cost 12 a single core manages about 10 guesses a second.
6. A fast hash such as SHA-256 runs at billions of guesses a second on a GPU. The same stolen table would fall in hours instead of centuries.
7. The salt kills the other attack. Without salts, an attacker hashes the top ten million common passwords once and looks up every user at the same time.
8. With a unique salt per user, that precomputed table is worthless, and the attacker must redo all the work separately for every single account.
9. Now authorization, with a worked role-based example. Three roles, four permissions, and one rule table.

Role	Read post	Edit any post	Delete user
reader	yes	no	no
editor	yes	yes	no
admin	yes	yes	yes

10. A user is given roles. A role is given permissions. Code never asks "is this user an admin"; it asks "does this user have `post.edit`".

```
ROLE_PERMS = {
  "reader": {"post.read"},
  "editor": {"post.read", "post.edit"},
  "admin":  {"post.read", "post.edit", "user.delete"},
}
```

```
def can(user, perm):
    return any(perm in ROLE_PERMS[r] for r in user.roles)
```

11. The difference matters when the rules change. Adding a "moderator" role who may edit but not delete is one line in the table.
12. If the code had said `if user.is_admin` in forty places, the same change is forty edits and you will miss one.
13. Authentication failed gives HTTP 401. Authorization failed gives HTTP 403. The 401 name in the standard is "Unauthorized", which is a historical mistake; it really means unauthenticated.

35.12.4 PLAIN - what is really happening inside

1. Session flow, step by step. You POST your email and password over HTTPS.
2. The server looks up the row, runs the hash function with the stored salt, and compares. It uses a constant-time comparison so timing leaks nothing.
3. On success it creates a random session identifier, at least 128 bits from a cryptographically secure generator, and stores it with your user id.
4. It sends that identifier back in a `Set-Cookie` header. The browser stores it and attaches it to every later request to that site automatically.
5. The cookie carries flags that do the real security work. `HttpOnly` hides it from JavaScript. `Secure` sends it only over HTTPS. `SameSite` controls whether other sites can cause it to be sent.
6. Logging out deletes the server record. The ticket is dead immediately.
7. Token flow, step by step. Same login, but the server returns a **JWT**, a JSON Web Token, and stores nothing.
8. A JWT is three base64url-encoded parts joined by dots: a header saying which algorithm, a payload of claims such as user id and expiry, and a signature.
9. The signature is computed by the server over the first two parts using a secret key. Anyone can read a JWT; only the key holder can forge one.
10. Read that again, because it is the mistake people make: a JWT is signed, not encrypted. Never put anything private in the payload.
11. To check a request, the server verifies the signature and the expiry. No database lookup at all, which is why tokens scale across many servers.
12. The cost is revocation. There is no record to delete, so a stolen token works until it expires. That is why access tokens are given minutes, not days.
13. The standard fix is a pair: a short access token, plus a long-lived **refresh token** kept in an `HttpOnly` cookie, which can be revoked and rotated.
14. Now the delegated login flow. The point of OAuth 2.0 is that the site you are visiting never sees your Google password, and never can.
15. The four parties: you (the resource owner), the site (the client), Google's login server (the authorization server), and Google's API (the resource server).

```
|              |<--your profile data-------|
|<--logged in--|               |           |
```

16. Step 1: the site sends your browser to the authorization server with its client id, the scopes it wants, a redirect address, a random `state` value, and a code challenge.
17. Step 2: you log in on Google's own page, on Google's own domain, and approve the list of scopes. The site is not involved and cannot see any of it.
18. Step 3: Google redirects your browser back to the site with a short-lived **authorization code** in the URL.
19. Step 4: the site's server exchanges that code, over a direct back-channel call, for tokens. This step needs the client secret and the code verifier, which never touch the browser.
20. Step 5: the site calls the API with the access token to fetch the data it was granted, and nothing more.
21. Why the extra hop with a code, instead of just handing over a token? Because URLs leak. They land in browser history, server logs and referrer headers. A code is single-use and worthless without the secret.
22. OpenID Connect is a thin layer on top. OAuth 2.0 alone answers "may this site read your calendar". It does not answer "who are you".
23. OpenID Connect adds an **ID token**, a JWT about you specifically, plus a standard `userinfo` endpoint and the standard scopes `openid`, `profile` and `email`. That is the part that makes it a login system.

35.12.5 TECHNICAL - the engineer's version

1. bcrypt was published by Niels Provos and David Mazieres at USENIX in 1999, built on the Blowfish cipher (Bruce Schneier, 1993). Its cost parameter is log-2, so cost 12 means 2^12 key expansion rounds.
2. bcrypt's real limits: it truncates input at 72 bytes, and some implementations mishandle the NUL byte. Do not pre-hash into it without base64 encoding first, or you can shorten the effective input.
3. scrypt came from Colin Percival in 2009 and is specified in RFC 7914 (2016). It added memory hardness, to blunt custom hardware.
4. Argon2 won the Password Hashing Competition in July 2015. Authors: Alex Biryukov, Daniel Dinu and Dmitry Khovratovich. It is specified in RFC 9106 (September 2021). Use the Argon2id variant.
5. Current OWASP Password Storage Cheat Sheet settings, as of 2026:

Algorithm	Recommended setting
Argon2id	m=19456 KiB, t=2, p=1
bcrypt	cost factor 10 or more
scrypt	N=2^17, r=8, p=1
PBKDF2-SHA256	600,000 iterations

6. A **pepper** is a site-wide secret mixed in before or after hashing, kept outside the database, ideally in a key management service. It protects against a database-only leak. It is optional; salting is not.
7. NIST SP 800-63B-4, "Digital Identity Guidelines: Authentication and Authenticator Management", published 31 July 2025, is the current reference.
8. It says: require at least 8 characters, accept at least 64, allow all printable ASCII and Unicode, and do not impose composition rules such as "one capital and one symbol".
9. It also says do not force periodic rotation without evidence of compromise, and do check new passwords against known-breached lists. Forced 90-day rotation is now explicitly discouraged, reversing decades of advice.
10. JWT is RFC 7519 (May 2015), sitting on JWS (RFC 7515), JWE (RFC 7516), JWK (RFC 7517) and JWA (RFC 7518). Bearer usage is RFC 6750.
11. The four classic JWT mistakes, all still seen in production:

Mistake	Why it is fatal
Storing in localStorage	any XSS reads it directly
No `exp` claim	the token never dies
Accepting `alg: none`	signature check skipped
Not pinning the algorithm	RS256 forced down to HS256

12. The `alg: none` and algorithm-confusion attacks were publicized by Tim McLean in March 2015. In the confusion attack the server is tricked into verifying an HMAC using the public RSA key as the shared secret, which the attacker also has. Always pin the expected algorithm on the verify side.
13. `localStorage` versus cookies, stated plainly: `localStorage` is readable by any script running on the page, so one cross-site scripting hole exports every token. An `HttpOnly` cookie is not reachable from JavaScript at all.
14. The cost of cookies is cross-site request forgery, which `SameSite=Lax` or `Strict` plus a CSRF token handles. Chrome made `SameSite=Lax` the default in version 80, February 2020.
15. OAuth 2.0 is RFC 6749 (October 2012), replacing OAuth 1.0 (RFC 5849, 2010). PKCE, Proof Key for Code Exchange, is RFC 7636 (September 2015).
16. RFC 9700, "Best Current Practice for OAuth 2.0 Security", published 30 January 2025 as BCP 240, is now the document to follow. It requires PKCE for all clients including confidential ones, forbids the implicit grant and the resource owner password credentials grant, and requires exact string matching of redirect URIs.
17. OAuth 2.1 is an IETF draft, not yet an RFC as of 2026. It folds RFC 9700's advice into one document. Say "draft", not "standard".
18. OpenID Connect Core 1.0 was published by the OpenID Foundation in February 2014. Discovery lives at `/.well-known/openid-configuration`; signing keys at a JWKS endpoint.
19. Single sign-on in enterprises is still largely SAML 2.0, an OASIS standard from March 2005, using signed XML assertions over HTTP POST. New systems use OIDC; you will meet SAML because it is entrenched, not because it is better.
20. Multi-factor authentication means two of: something you know, something you have, something you are. Two passwords are not two factors.

Factor	Standard	Real weakness
SMS code	none, carrier route	SIM swap, SS7
TOTP app	RFC 6238, 2011	phishable in real time
Push approve	vendor specific	fatigue, blind taps
Passkey	WebAuthn, FIDO2	device loss, recovery

21. TOTP (RFC 6238, May 2011) builds on HOTP (RFC 4226, December 2005): a 30-second time step, 6 digits, HMAC-SHA1 by default over a shared secret.
22. WebAuthn became a W3C Recommendation on 4 March 2019, with Level 2 in April 2021. Passkeys are discoverable WebAuthn credentials synced by the platform.
23. Passkeys are phishing-resistant by construction: the credential is bound to the origin, so a look-alike domain simply gets no signature. That property is an established fact from the protocol, not a marketing claim.
24. RBAC was formalized by David Ferraiolo and Richard Kuhn at NIST in 1992 and by Ravi Sandhu and colleagues in 1996, and standardized as INCITS 359-2004.
25. ABAC, attribute-based access control (NIST SP 800-162, 2014), decides from attributes of subject, object, action and environment. It is more expressive and much harder to audit. Most teams want RBAC with a few ownership checks.

26. Tools: `bcrypt`, `argon2-cffi`, `libsodium`; `jwt.io` and the `jose` libraries for inspecting tokens; `openssl rand -base64 32` for secrets; Keycloak, Auth0, Okta and Entra ID as identity providers.

35.12.6 WORDS - remember these

1. Authentication — proving who you are — verifying a claimed identity against one or more authenticators.
2. Authorization — deciding what you may do — evaluating a policy over subject, action and resource after identity is established.
3. Salt — a per-user random extra — a unique non-secret value that defeats precomputed rainbow tables.
4. Pepper — a site-wide secret extra — a key held outside the database and mixed into the hash input or output.
5. Work factor — how slow the hash is on purpose — the cost, memory and parallelism parameters tuned to a target verify time.
6. Session — a ticket the server remembers — server-side state keyed by an opaque high-entropy identifier held in a cookie.
7. JWT — a signed pass the user carries — a JSON Web Token, base64url header, payload and signature under RFC 7519.
8. Access token — the short-lived key to the API — a bearer credential with a minutes-long `exp` claim.
9. Refresh token — the long-lived key to get new keys — a revocable credential exchanged at the token endpoint, usually rotated on each use.
10. OAuth 2.0 — letting a site act for you without your password — a delegated authorization framework, RFC 6749, with PKCE per RFC 9700.
11. OpenID Connect — the login layer on top of OAuth — an identity layer adding an ID token and a userinfo endpoint.
12. RBAC — permissions attached to job titles — role-based access control per INCITS 359-2004, users to roles to permissions.

35.13 Application architecture

35.13.1 PLAIN - in simple words

1. Architecture is the set of decisions that are expensive to change later. It is not about which framework you pick.
2. A **monolith** is one program that does everything. One codebase, one build, one deploy, usually one database.
3. **Microservices** means the same work split into many small programs that talk over the network, each with its own deploy and often its own database.
4. The honest advice, given by most experienced engineers, is this: start with a monolith. Almost every team should.
5. Inside any of them, the code is arranged in **layers**, so that the part drawing the screen is not also the part talking to the database.
6. Some answers are expensive to compute and rarely change, so we keep a copy ready to hand. That copy is a **cache**.
7. Some work does not need to happen while the user waits. We drop a note into a **queue** and a separate worker picks it up later.
8. Sometimes the server needs to tell the browser something without being asked, which normal request-and-reply cannot do.
9. And because anyone can call your server as fast as they like, you have to cap how often each caller may do it. That is **rate limiting**.

35.13.2 PLAIN - a picture in your head

1. Think of a restaurant.
2. A monolith is one kitchen. Everybody cooks in the same room, shares the same fridge, and shouts across the pass when something is ready.
3. Communication is instant and free. Nobody has to book a phone call to ask whether the sauce is done.
4. When the room gets too crowded, you cannot make just the grill section bigger. You have to build a second identical kitchen.
5. Microservices are several small kitchens in separate buildings: one does bread, one does sauces, one does desserts.
6. Each can be scaled, replaced or rebuilt without closing the others. But every request between them is now a phone call that can be busy, slow or dropped.
7. The caching layer is the pass with finished dishes under a heat lamp. Serving from there takes seconds instead of minutes.
8. The queue is the ticket rail. Orders go up, cooks take them down in order, and nothing is lost if one cook steps away.

Where this comparison breaks:

9. A kitchen phone call fails loudly and immediately. A network call can succeed at one end and be lost at the other, so the sender never learns which.
10. Real kitchens do not silently duplicate an order because the first acknowledgement went missing. Distributed systems do exactly that, which is why every message consumer must be safe to run twice.

35.13.3 PLAIN - a worked example

1. A product page that shows a price, a stock count and a review average.
2. Version one asks the database on every single request. At 500 requests a second, that is 1,500 queries a second for data that changes hourly.
3. Version two caches the assembled result in Redis for 60 seconds.

```
def product(pid):
    key = f"product:{pid}"
    hit = redis.get(key)
    if hit:
        return json.loads(hit)          # about 0.5 ms
    data = build_from_db(pid)           # about 40 ms
    redis.setex(key, 60, json.dumps(data))
    return data
```

4. With a 95 percent hit rate the average response time falls from about 40 ms to about 2.5 ms, and the database sees 75 queries a second instead of 1,500.

Layer	Typical hit time	Who it serves
CDN edge	5 to 30 ms	everyone, worldwide
HTTP browser cache	0 ms	one returning user
Redis	0.3 to 1 ms	all app servers
In-process memory	0.0001 ms	one server only
PostgreSQL	1 to 50 ms	the source of truth

5. Now the queue. When an order is placed we must charge the card, send an email, generate a PDF invoice and update a search index.
6. Doing all four inline makes the user wait about 3 seconds and couples the checkout button to the email provider being awake.

7. Doing only the charge inline, and pushing the other three onto a queue, gives a 200 ms response and a checkout that survives the email provider going down.

```
@app.post("/orders")
def create_order(req):
    order = save_order(req)     # inline, must be certain
    charge_card(order)          # inline, user needs the answer
    queue.enqueue("send_receipt", order.id)
    queue.enqueue("build_invoice", order.id)
    queue.enqueue("index_order", order.id)
    return {"id": order.id}     # about 200 ms
```

8. The rule that makes this safe: the worker must be **idempotent**, meaning running it twice has the same effect as running it once.
9. So `send_receipt` records that it sent receipt number 5512 before it exits, and checks that record on entry. Otherwise a retry mails the customer twice.

35.13.4 PLAIN - what is really happening inside

1. Layers, from the outside in. The transport layer receives HTTP and knows nothing about your business. The service or application layer holds the rules. The data access layer talks to the database. The domain holds the concepts.
2. The discipline is that dependencies point one way, inward. The database layer must never import the HTTP layer.
3. **MVC** is the oldest version of this idea. The Model holds data and rules, the View renders, the Controller receives input and coordinates.
4. In a modern web app the View often lives in the browser and the Controller is an HTTP route handler, which is why the term drifted.
5. Caching happens at four separate places and they are independent: in the browser, at a CDN, in a shared store like Redis, and inside the process.
6. HTTP caching is negotiated with headers. `Cache-Control: max-age=3600` says how long a copy stays fresh. `ETag` gives a version fingerprint, so a stale copy can be revalidated with a cheap 304 Not Modified instead of a resend.
7. A **CDN** is a network of servers placed near users. It caches your responses physically close to the reader, which cuts the speed-of-light cost that no amount of server tuning can remove.
8. The hard part of caching is not storing. It is knowing when a stored answer became a lie. There are two strategies: expire on a timer, or actively delete the entry when the underlying data changes.
9. Queues decouple in time. The producer writes a message and returns. The broker stores it durably. A consumer takes it, does the work, and acknowledges.
10. If the consumer crashes before acknowledging, the message reappears. That is at-least-once delivery, and it is why duplicates happen.
11. Failed messages retry with growing gaps, and after a limit are moved to a **dead-letter queue** so one poisoned message cannot block the line forever.
12. Server-to-browser messaging has three shapes. Polling asks repeatedly. Server-sent events keeps one HTTP response open and streams text down it. WebSockets upgrades the connection to a two-way channel.
13. Rate limiting works by counting requests per key, usually per API key or per IP address, and refusing once the count passes a threshold in a time window.

35.13.5 TECHNICAL - the engineer's version

1. Conway's Law, from Melvin Conway's 1967 paper "How Do Committees Invent?", states that a system's structure copies the communication structure of the organization that built it. Microservices without matching team boundaries produce a distributed monolith, which has every cost and no benefit.

2. The term microservices was popularized by James Lewis and Martin Fowler in March 2014. Fowler's own follow-up, "MonolithFirst" (June 2015), argues that teams who started monolithic and split later did better than those who began distributed.
3. The honest trade-off table:

Concern	Monolith	Microservices
Local call cost	nanoseconds	milliseconds, can fail
Deploy	one unit, all at once	independent per service
Transaction	one ACID commit	sagas, eventual consistency
Debugging	one stack trace	distributed tracing needed
Scaling	whole app together	per service, precisely
Team size that fits	1 to 30	many teams, own on-call

4. Amazon's Prime Video team published in March 2023 that moving an audio-and-video monitoring service from distributed serverless components back into a single process cut infrastructure cost by over 90 percent. It is one case, not a law, but it punctured the idea that splitting always wins.
5. A **modular monolith** is the current mainstream advice: one deployable unit with strictly enforced internal module boundaries, so a later split is mechanical rather than archaeological.
6. MVC comes from Trygve Reenskaug at Xerox PARC in 1979, in Smalltalk-80. Related patterns: MVP, MVVM (Microsoft, 2005), and hexagonal architecture (Alistair Cockburn, 2005).
7. HTTP caching is specified in RFC 9111 (June 2022), which replaced RFC 7234. Key directives: `max-age`, `s-maxage` for shared caches only, `no-store`, `private`, `must-revalidate`, and `stale-while-revalidate` from RFC 5861.
8. Validators are `ETag` with `If-None-Match`, and `Last-Modified` with `If-Modified-Since`. A match returns 304 with no body.
9. Cache patterns: cache-aside (read through the app, the code above), write-through, write-behind, and refresh-ahead. Cache-aside is the default because it is the only one that is simple to reason about.
10. A **cache stampede** happens when a popular key expires and a thousand requests all miss and all rebuild it at once. Fixes: a mutex or single-flight lock, probabilistic early expiry, or `stale-while-revalidate`.
11. Akamai, founded in 1998 out of MIT work by Tom Leighton and Daniel Lewin, created the commercial CDN. Cloudflare followed in 2009, Fastly in 2011.
12. Message brokers: RabbitMQ (2007, implementing AMQP 0-9-1), Apache Kafka (built at LinkedIn by Jay Kreps, Neha Narkhede and Jun Rao, open sourced in 2011), Amazon SQS (generally available 2006), NATS, and Redis Streams.
13. Kafka is a distributed append-only log with consumer offsets, so replay is normal. RabbitMQ is a broker that deletes on acknowledgement. They solve different problems and are frequently confused.
14. Job frameworks by language: Celery and RQ for Python, Sidekiq for Ruby, BullMQ for Node, Quartz and Spring Batch for Java, Hangfire for .NET.
15. Retry policy in practice: exponential backoff with full jitter, a maximum attempt count, and a dead-letter queue. Without jitter, retries synchronize and produce a thundering herd against a recovering service.
16. The real-time comparison:

Property	Polling	SSE	WebSocket
Direction	client asks	server to client	both ways
Protocol	plain HTTP	plain HTTP	upgrade, RFC 6455
Reconnect	trivial	automatic	you write it
Binary data	yes	no, UTF-8 text	yes
Proxy trouble	none	rare	occasional

Property	Polling	SSE	WebSocket
Good for	slow updates	feeds, tokens, logs	chat, games, cursors

17. WebSocket is RFC 6455 (December 2011); it starts as an HTTP request with `Upgrade: websocket` and then leaves HTTP framing entirely. RFC 8441 (2018) added bootstrapping over HTTP/2.
18. Server-sent events are part of the HTML Living Standard, exposed as the `EventSource` API, with `Content-Type: text/event-stream`. The browser reconnects on its own and resumes with `Last-Event-ID`.
19. SSE is the quiet default for AI token streaming, dashboards and progress bars, because it is ordinary HTTP and needs no special infrastructure.
20. Rate limiting algorithms and what they actually do:

Algorithm	Behaviour
Fixed window	simple, allows 2x burst at edge
Sliding log	exact, memory grows with traffic
Sliding counter	near-exact, cheap, common
Token bucket	allows a controlled burst
Leaky bucket	smooths output to a fixed rate

21. The response is 429 Too Many Requests, defined in RFC 6585 (April 2012), with a `Retry-After` header from RFC 9110. The `RateLimit-Limit` and `RateLimit-Remaining` headers are widely used by convention but are still an IETF Internet-Draft, not an RFC, as of 2026.
22. Tools: `redis-cli --stat` and `INFO stats` for hit ratio, `curl -I` to read cache headers, `nginx limit_req` and Envoy for edge limiting, OpenTelemetry with Jaeger or Tempo for distributed tracing.

35.13.6 WORDS - remember these

1. Monolith — one program that does it all — a single deployable unit sharing one process, build and usually one datastore.
2. Microservice — a small program with its own deploy — an independently deployable service owning its data behind a network API.
3. Modular monolith — one deploy, firm inner walls — enforced module boundaries inside a single artifact, splittable later.
4. Cache — a ready-made copy of a slow answer — a store trading staleness for latency, governed by a TTL or explicit invalidation.
5. CDN — servers near the user — a geographically distributed edge cache for static and cacheable dynamic responses.
6. ETag — a version fingerprint for a response — a validator enabling conditional requests and 304 Not Modified replies.
7. Queue — a durable to-do list between programs — a broker giving at-least-once delivery with acknowledgement and retry.
8. Idempotent — safe to run twice — an operation whose repeated application gives the same result as one application.
9. Dead-letter queue — the box for messages that keep failing — a sink for messages exceeding the retry limit, kept for inspection.
10. WebSocket — a two-way phone line — a full-duplex framed protocol over one TCP connection, RFC 6455.
11. Server-sent events — a one-way news feed — a long-lived HTTP response streaming `text/event-stream` with automatic reconnection.

12. Rate limit — a cap on how often you may ask — a per-key request budget over a window, refused with HTTP 429.

35.14 Mobile apps

35.14.1 PLAIN - in simple words

1. A phone app can be built in three broadly different ways, and the choice decides most of what follows.
2. **Native** means writing in the language the phone maker intends, with the phone maker's own toolkit. Two apps, two codebases, two teams' worth of work.
3. **Cross-platform** means writing once in a shared language and letting a framework produce something that runs on both phones.
4. **Web-based** means it is really a website, wrapped so it can be installed and opened from the home screen. That is a progressive web app, a PWA.
5. On iPhone, native means Swift, usually with the SwiftUI toolkit. On Android, native means Kotlin, usually with Jetpack Compose.
6. The main cross-platform choices are React Native, which uses JavaScript and the real platform controls, and Flutter, which uses Dart and paints every pixel itself.
7. There is one more thing that has no equivalent on the web: a stranger at Apple or Google reads your app and decides whether it may exist.
8. On the web you deploy and it is live. On a phone store you submit, you wait, and you can be told no.
9. What you actually upload is a file. On Android it is an APK or an AAB. On iPhone it is an IPA. Both are, underneath, ordinary ZIP archives.

35.14.2 PLAIN - a picture in your head

1. Think about writing a book that must be published in two countries with different languages and different censors.
2. Native is hiring two native writers, each producing a book that reads perfectly naturally in its own country. Best result, double the cost, and the two books drift apart over time.
3. Cross-platform is writing once and translating. React Native is a good human translator using each country's own idioms and typesetting. Flutter is printing your own book, with your own fonts and paper, and shipping it unchanged to both countries.
4. Flutter's version always looks identical everywhere, which is a feature if you want your brand, and a problem if readers expected the local style.
5. A PWA is publishing on a website that both countries can already read. No censor, no printing, instant updates, and no place on the bookshop shelf.

Where this comparison breaks:

6. A translated book cannot suddenly need a printing press the translator does not own. A cross-platform app frequently needs a device feature the framework has not wrapped yet, and then you write native code anyway.
7. And a censor reads a book once. App review happens on every single update, so it is a recurring tax on shipping, not a one-time gate.

35.14.3 PLAIN - a worked example

1. The same screen, a counter with a button, written four ways. Notice how similar the modern declarative shapes have become.

```
// iOS native: Swift with SwiftUI
struct CounterView: View {
  @State private var n = 0
  var body: some View {
```

```
    VStack {
      Text("Count: \(n)")
      Button("Add") { n += 1 }
    }
  }
}
```

```
// Android native: Kotlin with Jetpack Compose
@Composable
fun Counter() {
  var n by remember { mutableStateOf(0) }
  Column {
    Text("Count: $n")
    Button(onClick = { n++ }) { Text("Add") }
  }
}
```

```
// Flutter: Dart
class Counter extends StatefulWidget { ... }
// inside State:
Column(children: [
  Text('Count: $n'),
  ElevatedButton(
    onPressed: () => setState(() => n++),
    child: const Text('Add')),
])
```

```
// React Native: JavaScript
function Counter() {
  const [n, setN] = useState(0);
  return (<View>
    <Text>Count: {n}</Text>
    <Button title="Add" onPress={() => setN(n + 1)} />
  </View>);
}
```

2. Now the honest comparison for a real product decision.

Approach	Performance	Device access
Native Swift or Kotlin	best available	everything, day one
Flutter	very close to native	good, plugin gaps
React Native	good, JS bridge cost	good, plugin gaps
PWA	fine for content	most limited

3. And the effort side, which is what actually decides most projects.

Approach	Effort for both phones	Team you need
Native	roughly 180 percent	iOS plus Android devs
Flutter	roughly 110 percent	Dart devs, some native
React Native	roughly 110 percent	web devs, some native
PWA	roughly 60 percent	your existing web team

4. The “180 percent” is not 200 because design, backend and product work are shared. The extra 10 percent on cross-platform is the native code you end up writing anyway for permissions, push and store plumbing.

5. A concrete rule that holds up: if the app is mostly forms, lists and content, any of these works and you should pick by team. If it is a camera, a game, a map with heavy custom rendering, or something that must feel invisible, go native.

35.14.4 PLAIN - what is really happening inside

1. Native code is compiled ahead of time to machine code for the device's processor and calls the operating system's own UI objects directly. There is no layer in between, so there is no layer to be slow.
2. React Native runs your JavaScript in a JavaScript engine on the phone, and that JavaScript creates and updates real platform views. A `Text` really is a `UILabel` on iOS and a `TextView` on Android.
3. So React Native inherits the platform's look and accessibility for free, and pays a cost whenever data crosses between JavaScript and the native side.
4. Flutter does the opposite. It ships its own rendering engine and draws every button, scrollbar and cursor itself onto a blank canvas.
5. That gives identical output on both platforms and total design control, and means Flutter must re-implement anything the platform changes, including accessibility behaviour and text selection.
6. A PWA is a website with three additions: a manifest file naming the app and its icons, a service worker script that can serve files while offline, and a requirement to be served over HTTPS.
7. The service worker is the key piece. It is a script that sits between the page and the network, can answer requests from a local cache, and keeps running briefly after the page closes.
8. Store review, step by step. You build a signed package, upload it, fill in metadata and screenshots, and it enters a queue.
9. Automated scanning runs first: malware checks, private API usage, permission declarations, size limits. Then, on Apple's store, a human opens the app.
10. The human checks the guidelines: does it do something useful, does it match its description, does it collect data it declared, does it route payments for digital goods through the store's own system.
11. Rejection is normal and is not the end. You get a message, you fix it, you resubmit, and the clock restarts.
12. What this means for shipping is the real point: you cannot fix a live bug in ten minutes. Plan for a review delay on every release, keep a server-side switch to turn broken features off without a new build, and never ship on a Friday hoping to patch on Saturday.
13. Now the files themselves, which ties directly to the file-format material in Chapter 20: a file's identity comes from the bytes at its start, not from its extension.
14. Open an APK in a hex viewer and the first four bytes are `50 4B 03 04`, which is `PK` followed by two control bytes. That is the ZIP local file header signature, named after Phil Katz.
15. An APK is a ZIP. Rename it to `.zip`, unzip it, and you can read the whole structure. So is an IPA. So is a `.jar`, a `.docx` and an `.epub`.
16. Inside an APK you find compiled code, compiled resources, a compiled manifest and a signature folder. Inside an IPA you find a folder called `Payload` holding one `.app` bundle.

```
app.apk (ZIP)               app.ipa (ZIP)
+- AndroidManifest.xml      +- Payload/
|    (binary XML, not text) |   +- MyApp.app/
+- classes.dex              |       +- MyApp  (Mach-O)
+- resources.arsc           |       +- Info.plist (binary)
+- res/                     |       +- Assets.car
+- lib/arm64-v8a/*.so       |       +- _CodeSignature/
+- assets/                  |       +- embedded.mobileprovision
+- META-INF/  (signatures)  +- iTunesMetadata.plist
```

17. The manifest inside an APK is not readable text. It is Android binary XML, a compact encoding with a string pool, which is why opening it in a text editor shows mush. The tool `aapt2 dump badging` prints it properly.
18. `classes.dex` holds Dalvik Executable bytecode: all your Kotlin and Java classes merged into one register-based bytecode file, not the stack-based `.class` files the Java compiler first produced.

19. `Info.plist` inside an iOS app is a property list, usually stored in Apple's binary plist format starting with the bytes `bplist00`, not XML.

35.14.5 TECHNICAL - the engineer's version

1. Swift was announced at Apple's WWDC in June 2014, designed by a team led by Chris Lattner, and open sourced in December 2015. Swift 6.3 is the current release in 2026, shipping with Xcode 26.
2. SwiftUI was announced at WWDC in June 2019. UIKit, from iPhone OS 2.0 in 2008, remains fully supported and is still required for some controls, so real apps mix both through `UIViewRepresentable`.
3. Kotlin came from JetBrains, announced in 2011, version 1.0 in February 2016. Google announced first-class Android support in May 2017 and a Kotlin-first policy in May 2019.
4. Jetpack Compose reached 1.0 in July 2021. The older approach, XML layouts with `findViewById` or view binding, is still everywhere in existing code.
5. React Native was open sourced by Facebook in March 2015. Its New Architecture, with the Fabric renderer and TurboModules replacing the old asynchronous bridge, became the default in version 0.76 (October 2024). Version 0.87 was released on 10 August 2026.
6. Flutter's first stable release, 1.0, was December 2018, after a 2017 alpha. Flutter 3.47 is current in 2026. It uses Dart, compiled ahead of time to native code for release builds, and the Impeller renderer, which replaced Skia as the default on iOS and on Android API 29 and above.
7. Other real options: Kotlin Multiplatform, stable since November 2023, which shares business logic while leaving the UI native; .NET MAUI (May 2022); Capacitor with Ionic; and Tauri for desktop.
8. Progressive web app is not a standard; it is a name for a set of standards used together: the Web App Manifest (a W3C specification), Service Workers (W3C), and a secure context requirement.
9. The term was coined by Frances Berriman and Alex Russell in 2015. Safari on iOS 16.4, released 27 March 2023, finally added Web Push and badging for home-screen web apps, closing the largest gap.
10. Remaining PWA gaps on iOS as of 2026: no Web Bluetooth, no Web NFC, limited background execution, storage that can be evicted, and no place in the App Store search results, which is a distribution problem, not a technical one.
11. App Store review facts. The App Store opened on 10 July 2008. Apple states that on average 90 percent of submissions are reviewed in less than 24 hours. That is an average, and complex or first-time submissions take longer.
12. Common rejection causes are guideline 2.1 (crashes and incomplete information), 4.2 (minimum functionality, which kills thin website wrappers), 3.1.1 (digital goods must use in-app purchase), and privacy label mismatches.
13. Commission is 30 percent, or 15 percent under the App Store Small Business Program for developers below one million US dollars a year, and 15 percent on subscriptions after a subscriber's first year.
14. The EU Digital Markets Act forced alternative app marketplaces and third-party browser engines on iOS in the EU from iOS 17.4, March 2024. This is jurisdiction-specific and still changing; check the current position rather than trusting any book, including this one.
15. Google Play review facts. Review is mostly automated with human escalation. New personal developer accounts must run a closed test with at least 12 testers for 14 continuous days before applying for production access, a rule introduced in November 2023 and later relaxed from 20 testers to 12.
16. Google Play has required the Android App Bundle for new apps since August 2021. Play requires apps to target API level 36, Android 16, by 31 August 2026.
17. An AAB is a publishing format, not an installable one. Play uses it to generate split APKs per device: one base, plus configuration splits for the device's screen density, CPU architecture and language. Typical download size saving is 15 to 20 percent.
18. APK signing schemes, and which Android version introduced each:

Scheme	Since	What it covers
v1 (JAR)	Android 1.0	each file, per entry
v2	Android 7.0, 2016	whole APK bytes
v3	Android 9, 2018	adds key rotation
v3.1	Android 13, 2022	rotation targeting
v4	Android 11, 2020	streaming install

19. iOS code signing uses a certificate plus a provisioning profile embedded as `embedded.mobileprovision`. `_CodeSignature/CodeResources` holds a hash of every file in the bundle, so changing one byte after signing breaks launch.
20. App thinning means the store delivers only the slices a device needs. Apple calls the pieces app slicing, bitcode (deprecated with Xcode 14, 2022) and on-demand resources.
21. Inspection commands worth knowing:

```
unzip -l app.apk            # it really is a ZIP
aapt2 dump badging app.apk  # readable manifest
apksigner verify -v app.apk # which schemes signed it
unzip -p app.ipa 'Payload/*.app/Info.plist' | plutil -p -
codesign -dvvv MyApp.app    # signing identity
```

22. Where experts disagree: whether Flutter's own-rendering approach is a strength or a liability. One camp values pixel-identical output and one codebase. The other argues that redrawing platform widgets means permanently chasing platform changes, especially in accessibility and text input. Both positions have shipped very large apps.

35.14.6 WORDS - remember these

1. Native app — built with the phone maker's own tools — compiled against the platform SDK, using the platform's UI framework directly.
2. Cross-platform — one codebase, both phones — a shared runtime or compiler targeting iOS and Android from common source.
3. PWA — a website you can install — a manifest plus a service worker over HTTPS, installable to the home screen.
4. Service worker — a script that answers requests offline — a background worker intercepting fetches and managing a cache.
5. APK — the Android app file — a ZIP containing DEX bytecode, binary XML manifest, `resources.arsc` and signature blocks.
6. AAB — what you upload, not what installs — the Android App Bundle, from which Play generates per-device split APKs.
7. IPA — the iPhone app file — a ZIP containing a `Payload` folder holding one signed `.app` bundle with a Mach-O binary.
8. DEX — Android's compiled code format — Dalvik Executable, register-based bytecode ahead-of-time compiled by ART at install.
9. App review — a person deciding if your app may ship — guideline conformance checking applied to every submission and update.
10. Code signing — a seal proving nobody changed the app — certificate-backed hashes over every file, verified at install and launch.

35.15 Testing, performance and accessibility for the web

35.15.1 PLAIN - in simple words

1. Three things separate a demo from a product: it keeps working when you change it, it is fast on a normal phone, and everyone can use it.
2. Tests are code that checks your other code. You run them automatically, every time, so a mistake is caught in seconds rather than by a customer.
3. There are three sizes of test. Small ones check one function. Medium ones check that two real parts fit together. Large ones drive a real browser like a real person.
4. Small tests are fast and cheap, so write many. Large tests are slow and fragile, so write few, and only for the paths that must never break.
5. Speed is not a feeling, it is a measurement, and Google publishes three specific numbers called the **Core Web Vitals**.
6. One measures how long until the main thing appears. One measures how long the page takes to answer a tap. One measures how much the page jumps around while loading.
7. **Lighthouse** is a free tool built into Chrome that scores a page and tells you what to fix.
8. **Accessibility** means the site works for people who cannot see it, cannot use a mouse, or need much larger text.
9. That is not a nice extra. It is a legal requirement in many places, and the basics are genuinely small: correct HTML tags, text for images, working keyboard navigation, readable contrast, a visible focus outline, and labels on inputs.

35.15.2 PLAIN - a picture in your head

1. Think of building a bridge and checking it three ways.
2. You test each bolt in a jig before it goes in. That is a unit test: thousands of them, seconds each, and a failure tells you exactly which bolt.
3. You test one assembled span under load. That is an integration test: fewer, slower, and a failure tells you the span is wrong but not which bolt.
4. You drive a lorry across the finished bridge. That is an end-to-end test: a handful, minutes each, and a failure tells you only that something is wrong.
5. All three are worth doing. Only doing the lorry is how you end up rebuilding the bridge to find one bad bolt.

Where this comparison breaks:

6. A bridge bolt does not change while you are testing it. Software does. Tests also protect against your future self, which a bolt jig never has to do.
7. And a lorry crossing is reliable. Browser tests fail randomly for reasons that have nothing to do with your code, which is called flakiness, and a test suite nobody trusts is worse than no suite at all.

35.15.3 PLAIN - a worked example

1. A single button component, tested at all three levels.

```
// unit: does the function do arithmetic
test("adds tax", () => {
  expect(withTax(100, 0.2)).toBe(120);
});

// integration: do component and state work together
test("shows total after adding", async () => {
  render(<Cart />);
  await user.click(screen.getByRole("button",
                   { name: /add/i }));
  expect(screen.getByText("120")).toBeVisible();
});
```

```
// end-to-end: a real browser, a real server
test("checkout works", async ({ page }) => {
  await page.goto("/product/42");
  await page.getByRole("button", { name: "Buy" }).click();
  await expect(page.getByText("Order placed")).toBeVisible();
});
```

2. Notice the queries use `getByRole` and the accessible name. A test written that way fails if the button stops being reachable by a screen reader, so the test suite quietly enforces accessibility.
3. Now the performance numbers. Here is one real page before and after work.

Metric	Before	After	Good if
LCP	4.8 s	1.9 s	2.5 s or less
INP	420 ms	140 ms	200 ms or less
CLS	0.31	0.02	0.1 or less
Lighthouse	38	94	90 or more

4. What actually produced those changes, in order of effect.
5. LCP: the hero image was 2.4 MB PNG. Converting to a 180 KB WebP, adding `width` and `height`, and marking it `fetchpriority="high"` removed most of the delay.
6. CLS: the same missing `width` and `height` were letting the text jump when the image finally arrived. Reserving the space fixed it. A web font swapping in late caused the rest, fixed with `font-display: optional`.
7. INP: a 300 ms sorting loop ran inside the click handler. Breaking it up and deferring the non-urgent part let the browser paint first.
8. The general lesson: two of the three fixes were one HTML attribute pair. Most real performance problems are not clever, they are neglected basics.

35.15.4 PLAIN - what is really happening inside

1. A test runner imports your code, runs functions, compares results, and reports the ones that differ. There is no magic in it.
2. An end-to-end runner starts a real browser process and drives it through an automation protocol, then reads back the rendered page.
3. Test doubles replace slow or unreliable parts. A **mock** stands in for a thing and records how it was called. A **fake** is a working cheap version, such as an in-memory database.
4. The trap is mocking so much that the test only proves your mocks agree with each other. Prefer a real database in a container over a mocked one.
5. **LCP**, Largest Contentful Paint, is the time from navigation start until the biggest image or text block in the visible area finishes rendering. The browser keeps updating its guess and stops at the first user interaction.
6. **INP**, Interaction to Next Paint, watches every click, tap and key press for the whole visit, measures from the input to the next frame painted, and reports roughly the worst one.
7. **CLS**, Cumulative Layout Shift, adds up how much visible content moved without the user causing it. Each shift scores the fraction of the screen affected times the distance moved. It has no unit.
8. There are two kinds of measurement and they answer different questions. Lab data comes from one run on one machine with simulated conditions. Field data comes from millions of real visits on real phones.
9. Lighthouse gives lab data. It is repeatable, so it is good for finding causes and for stopping regressions in a build pipeline.
10. Core Web Vitals as Google actually uses them are field data, collected from real Chrome users who opted in, and judged at the 75th percentile.

11. That percentile choice matters: it means three quarters of your visits must be good. Your own fast laptop on office fibre tells you nothing.
12. Accessibility works through the **accessibility tree**, a second structure the browser builds beside the DOM, where each element has a role, a name, a state and a value.
13. A screen reader reads that tree, not your pixels. A `<button>` arrives with role "button" and is focusable and clickable by keyboard, all for free.
14. A `<div onclick=...>` styled to look like a button arrives with no role, no name, no keyboard focus and no Enter key handling. That is why the tag you choose is the single biggest accessibility decision you make.

35.15.5 TECHNICAL - the engineer's version

1. The testing pyramid was described by Mike Cohn in his 2009 book "Succeeding with Agile". A common rule of thumb is roughly 70 percent unit, 20 percent integration, 10 percent end-to-end, though the exact split is convention, not a finding.
2. Kent C. Dodds' "testing trophy" (2018) is the main counter-argument: it puts the most weight on integration tests, on the grounds that they catch the most real bugs per minute spent. Experts genuinely disagree here; both shapes ship good software.

Level	Tools	Typical time
Unit	Vitest, Jest, pytest, JUnit 5	1 to 50 ms
Integration	Testing Library, supertest	50 to 500 ms
Container-backed	Testcontainers, MSW	0.5 to 5 s
End-to-end	Playwright, Cypress, Selenium	5 to 60 s

3. Dates: Selenium began in 2004 with Jason Huggins at ThoughtWorks; WebDriver became a W3C Recommendation in June 2018. Cypress launched in 2017. Playwright came from Microsoft in 2020, written by former Puppeteer authors. Jest came from Facebook in 2014; Vitest arrived in 2021 alongside Vite.
4. Coverage percentages measure lines executed, not behaviour verified. A suite at 100 percent coverage with no assertions proves nothing. Use coverage to find untested files, never as a target to hit.
5. Core Web Vitals thresholds, current as of 2026. A page passes only if all three are in the good band at the 75th percentile, split by mobile and desktop.

Metric	Good	Needs work	Poor
LCP	2.5 s or less	2.5 to 4.0 s	over 4.0 s
INP	200 ms or less	200 to 500 ms	over 500 ms
CLS	0.1 or less	0.1 to 0.25	over 0.25

6. History: LCP and CLS became Core Web Vitals in May 2020. INP was announced as a replacement for First Input Delay in May 2023 and became a Core Web Vital on 12 March 2024. FID was retired because it measured only the delay before the first handler ran, not how long the page took to respond.
7. Field data comes from the Chrome User Experience Report, CrUX, which publishes monthly aggregates. PageSpeed Insights shows CrUX and Lighthouse side by side, and they routinely disagree, which confuses people. They are measuring different populations.
8. Lighthouse performance score weights, current version:

Metric	Weight
Total Blocking Time	30 percent

Metric	Weight
Largest Contentful Paint	25 percent
Cumulative Layout Shift	25 percent
First Contentful Paint	10 percent
Speed Index	10 percent

9. Score bands are 0 to 49 poor, 50 to 89 needs improvement, 90 to 100 good. Lighthouse applies simulated mobile throttling by default, which is why a local score is lower than what your machine feels like.
10. Note that INP is not in the Lighthouse score. Lighthouse uses Total Blocking Time as its lab proxy, because a single automated load performs no interactions. Fixing TBT usually fixes INP, but not always.
11. Other measurement tools: WebPageTest for filmstrips and multi-location runs, the Chrome DevTools Performance panel for flame charts, and the `web-vitals` JavaScript library to report real user metrics to your own backend.
12. Accessibility is governed by WCAG, the Web Content Accessibility Guidelines. WCAG 2.0 became a W3C Recommendation in December 2008, 2.1 in June 2018, and WCAG 2.2 on 5 October 2023. WCAG 3.0 remains a working draft.
13. Conformance levels are A, AA and AAA. AA is the level almost all law and procurement references. AAA is not expected for whole sites.
14. The genuinely non-negotiable basics, each tied to its success criterion:

Basic	Criterion	Concrete rule
Semantic HTML	1.3.1, 4.1.2	real button, nav, h1 to h6
Alt text	1.1.1	describe it, or empty `alt`
Keyboard	2.1.1, 2.1.2	everything reachable, no traps
Contrast	1.4.3, 1.4.11	4.5:1 text, 3:1 large and UI
Focus visible	2.4.7, 2.4.11	a clear ring, never obscured
Labels	3.3.2, 1.3.5	every input has a `<label for>`

15. Contrast detail: 4.5 to 1 for normal text, 3 to 1 for large text, meaning at least 18 point or 14 point bold, which is about 24 CSS pixels or 18.66 pixels bold. AAA raises those to 7 to 1 and 4.5 to 1.
16. New in WCAG 2.2 and often missed: 2.4.11 Focus Not Obscured, 2.5.8 Target Size Minimum at 24 by 24 CSS pixels, and 3.3.8 Accessible Authentication, which forbids requiring a cognitive test such as transcribing a code by hand with no paste allowed.
17. The first rule of ARIA, from the W3C ARIA Authoring Practices Guide: do not use ARIA if a native HTML element with the needed semantics exists. Incorrect ARIA is measurably worse than none, because it overrides what the browser already knew.
18. Never remove a focus outline without replacing it. `outline: none` with no substitute is the single most common accessibility regression in modern CSS.
19. Automated tools find only part of the problem. Deque, the maker of axe, reports that automated testing catches roughly a third to a half of issues depending on the ruleset; the figure varies by study, so treat it as approximate. Nothing replaces a keyboard-only pass and a screen reader pass.
20. Law, briefly and with dates, because it drives budgets: Section 508 in the United States was refreshed in January 2017 to reference WCAG 2.0 AA. The European Accessibility Act's requirements applied from 28 June 2025. The United States Department of Justice ADA Title II rule of April 2024 requires WCAG 2.1 AA for state and local government sites, phasing in from 2026.
21. Tools: `axe DevTools` and `@axe-core/playwright` for automated checks, Lighthouse's accessibility category, WAVE, and the real screen readers: NVDA and JAWS on Windows, VoiceOver on macOS and iOS, TalkBack on Android.

35.15.6 WORDS - remember these

1. Unit test — checks one small piece alone — an isolated assertion over a single function or module with no external dependencies.
2. Integration test — checks two real parts together — a test exercising collaborating components, often with a real database in a container.
3. End-to-end test — drives the app like a person — a full-stack test through a real browser via an automation protocol.
4. Flaky test — passes and fails without changes — a non-deterministic test, usually a timing or ordering race, that destroys trust in the suite.
5. LCP — how long until the main thing appears — Largest Contentful Paint, good at 2.5 seconds or less at the 75th percentile.
6. INP — how long the page takes to answer a tap — Interaction to Next Paint, good at 200 milliseconds or less.
7. CLS — how much the page jumps about — Cumulative Layout Shift, unitless, good at 0.1 or less.
8. Lab data — one measured run — a synthetic test under fixed throttled conditions, repeatable and good for regression gates.
9. Field data — what real users got — real user monitoring aggregated at a percentile, such as the Chrome User Experience Report.
10. Accessibility tree — what a screen reader reads — a parallel tree of roles, names, states and values derived from the DOM.
11. WCAG AA — the level the law usually means — the middle conformance level of the Web Content Accessibility Guidelines, currently 2.2.
12. Semantic HTML — using the tag that means the thing — elements chosen for meaning, giving roles and keyboard behaviour without extra code.

35.16 A realistic learning path

35.16.1 PLAIN - in simple words

1. You have almost certainly seen a picture called a 180-day web development roadmap. It usually says something close to this.

Days	Topic
1 to 25	HTML and CSS
26 to 55	JavaScript
55 to 60	responsive design
61 to 75	React
76 to 105	Node and MongoDB
106 to 125	GitHub and APIs
126 to 180	cloud and projects

2. We are going to take that roadmap seriously, because it is not stupid. Parts of it are genuinely right, and it has helped a lot of people start.
3. What it gets right: the big order. Structure first, then styling, then the language, then a framework, then a server. You cannot begin at React.
4. What it also gets right: giving JavaScript more time than HTML and CSS. That ratio is correct, and most beginners get it backwards.
5. And it ends with projects and putting things online, which is exactly where a learning path should end.
6. Now the three things it gets wrong, said kindly, because each one costs beginners real months.
7. First, it teaches MongoDB before SQL, and in fact never teaches SQL at all. That is the wrong way round.

8. Second, it puts GitHub on day 106. Version control belongs on day one, before your first file, not after your first framework.
9. Third, and biggest: days are not the unit. "25 days of HTML" measures how long you sat there. It does not measure whether you can build a page.
10. The honest replacement is a list of stages, each defined by a thing you have built and can show, and each finished when the thing works.
11. Some people reach stage four in two months. Some take eight. Both are normal and neither number tells you anything about how good they will be.

35.16.2 PLAIN - a picture in your head

1. Think about learning to cook rather than learning to watch cooking.
2. A day-count roadmap is a timetable that says "week three: sauces". You attend week three. At the end of it you have watched sauces being made.
3. A stage-based path says "you are finished with sauces when you have served four different sauces to someone who ate them and told you the truth".
4. The second one cannot be faked. There is no way to have served the sauce without having made the sauce.
5. It also self-corrects. If your sauce splits, you go back and learn about emulsions because you need to, and that lesson sticks permanently, in a way no scheduled lecture on emulsions ever does.
6. And notice what the timetable never mentions: cleaning as you go, knife safety, tasting. Those are not week seven. They are every single day.
7. Version control, testing and accessibility are the cleaning as you go.

Where this comparison breaks:

8. Cooking gives you an answer in twenty minutes. Software gives you a bug that takes two days, and during those two days it feels like you are learning nothing. You are learning the most.
9. Also, a chef can taste the dish. You cannot tell whether your code is good by looking at it. That is what other people's review and real users are for.

35.16.3 PLAIN - a worked example

1. Here is the replacement. Eight stages. Each one names what to learn, what to build, and the specific signal that you are finished with it.

Stage	Learn	Build	Done when
0 Tools	git, shell, editor	a repo of notes	you can undo a commit
1 Pages	HTML, CSS, layout	3 real pages	it works at 320px
2 Language	JavaScript, DOM	a to-do, no library	you explain the loop
3 Data	SQL, schema design	a 4-table schema	you write a join
4 Server	HTTP, one backend	a JSON API + login	passwords are hashed
5 Client	React or Svelte	rebuild stage 2	you know what it saved
6 Ship	deploy, CI, logs	put it online	a stranger used it
7 Depth	tests, a11y, speed	improve stage 6	Lighthouse over 90

2. Stage 0, tools, roughly one week. Learn `git init`, `add`, `commit`, `push`, `branch`, `merge`, and how to read `git log`. Learn `cd`, `ls`, `grep`, and how to kill a stuck process.
3. Build: a repository where you keep your notes from this book, committed daily. The point is not the notes. The point is that committing becomes automatic before you have anything valuable to lose.

4. Done when: you can deliberately make a mess, and get out of it with `git restore`, `git reset` or `git revert`, without asking anyone.
5. Stage 1, pages, roughly three to five weeks. HTML semantics, the box model, flexbox, grid, and media queries. Responsive is not a separate topic; it is how you write CSS from the first line.
6. Build three pages that are not tutorials: a page about something you actually care about, a copy of a real site's layout, and a form-heavy page.
7. Done when: you resize the browser to 320 pixels wide and nothing overlaps, nothing overflows, and you did not need to look up flexbox.
8. Stage 2, the language, roughly six to eight weeks, and this is the stage people rush. Variables, types, functions, arrays, objects, closures, `this`, promises, `async`/`await`, modules, and the DOM API.
9. Build a to-do application with no framework at all. Then a small game. Then something that calls a public API and handles the error case properly.
10. Done when: you can predict the output order of a piece of asynchronous code on paper before running it, and you are right.
11. Stage 3, data, roughly three weeks, and this is the stage the popular roadmap skips entirely. Learn SQL before you learn any other database.
12. Build a schema for something real with four or five related tables: users, posts, comments, tags, and a join table. Load it with a few thousand rows.
13. Done when: you can write a query with two joins, a `GROUP BY` and a `HAVING` clause, and explain why you added an index.
14. Stage 4, the server, roughly four to six weeks. Pick one backend and stay there. Node with Express or Fastify, or Python with FastAPI or Django.
15. Build a JSON API with real registration and login, sessions or tokens done properly, input validation, and correct status codes.
16. Done when: passwords are stored with bcrypt or Argon2id, a wrong password gives 401, a valid user hitting somebody else's record gives 403, and you can say why those two are different.
17. Stage 5, the client framework, roughly four weeks. Now, and only now, learn React, Svelte or Vue.
18. Build: rebuild your stage 2 to-do app in the framework, against your stage 4 API. Same app, second time.
19. Done when: you can name three specific things the framework did for you that you had written by hand, and one thing it made harder.
20. Stage 6, shipping, roughly two weeks. A domain, HTTPS, environment variables, a build pipeline, a real deploy, and logs you can read at 2 a.m.
21. Done when: someone you have never met has used the thing and told you something about it. That single event teaches more than the previous month.
22. Stage 7, depth, ongoing forever. Tests, accessibility, performance, security and reading other people's code.
23. Done when: nothing. This stage does not end, and that is the job.

35.16.4 PLAIN - what is really happening inside

1. Why SQL before MongoDB, concretely. A relational schema forces you to decide, on day one, what your data actually is and how the pieces relate.
2. That decision is the hard part of software, and doing it early with a database that argues back is how you learn it.
3. A document database lets you postpone every one of those decisions. That feels faster and it genuinely is faster, for about six weeks.
4. Then you need to know which users commented on which posts, and you discover you now have to write the join yourself, in application code, without transactions, at three in the morning.
5. There is nothing wrong with MongoDB. It is a good database for documents that really are documents. It is a bad first database because it does not teach modelling, and modelling is the transferable skill.
6. Also, in almost every job you will apply for, the database is PostgreSQL, MySQL or SQL Server. SQL is asked about in interviews. Mongo query syntax almost never is.
7. Why git on day one. Every hour you spend without version control is an hour where the worst outcome is losing everything and the second worst is being afraid to try things.
8. Version control is not an advanced topic. It is the thing that makes learning safe, because you can break anything and get back.
9. It is also, bluntly, the one skill every single job requires. Learning it on day 106 means 105 days of work you cannot show anyone.
10. And be precise about the words: **git** is the version control program, made by Linus Torvalds in 2005. **GitHub** is one website that hosts git repositories, founded in 2008 and bought by Microsoft in 2018. Learn git. GitHub takes an afternoon once you know git.
11. Why days are the wrong unit. Two people both "do 25 days of HTML". One watches videos for two hours a day. One builds and breaks pages for four. They are not in the same place, and the calendar cannot tell them apart.
12. Worse, a day count creates a false finish line. Day 26 arrives, you move to JavaScript because the plan says so, and you carry a shaky understanding of the box model into everything that follows.
13. A build target has no such problem. The page either survives 320 pixels or it does not. You cannot argue with it and you cannot schedule past it.
14. What the popular roadmap leaves out entirely, and should not: testing, accessibility, security, debugging, and reading code you did not write.
15. Those are not advanced. Debugging in particular is most of the job, and no roadmap ever allocates a single day to it.
16. One more honest thing. You will feel, somewhere around stage 2, that everyone else understands this and you do not. That feeling is not evidence. It is what learning something genuinely hard feels like from the inside, and it arrives for everyone, including the people you are comparing yourself to.

35.16.5 TECHNICAL - the engineer's version

1. Realistic hour costs, from teaching experience rather than from a course listing. Treat these as approximate and personal.

Stage	Focused hours	At 15 h/week
0 Tools	10 to 20	about 1 week
1 Pages	60 to 100	4 to 7 weeks
2 Language	120 to 200	8 to 13 weeks
3 Data	40 to 70	3 to 5 weeks
4 Server	80 to 120	5 to 8 weeks
5 Client	50 to 80	3 to 5 weeks
6 Ship	20 to 40	1 to 3 weeks

2. That totals roughly 380 to 630 focused hours to be employable at a junior level, which at a genuine 15 hours a week is about 26 to 42 weeks.
3. "180 days" at 3 hours a day is 540 hours, so the popular roadmap's total is not unreasonable. Its problem is the distribution and the ordering, not the size.
4. The distribution the roadmap gets wrong, in numbers: it gives 25 days to HTML and CSS and 15 days to React, while giving zero to SQL, zero to testing and zero to debugging.
5. A portfolio that actually gets interviews contains three items, not fifteen:

Project	What it must prove
A full-stack app	auth, a real schema, deploy
A focused tool	you can finish something small
A contribution	you can read others' code

6. Concrete quality bar for the full-stack app: a live URL, a README with setup steps that work on a clean machine, at least a handful of tests, migrations rather than a hand-edited database, and no secrets in the repository.
7. Reading the git history of a candidate's project tells an interviewer more than the project does. Many small commits with real messages beat one commit called "initial commit" containing 40,000 lines.
8. Order of technologies, with the reasoning stated as a rule: learn the thing that is underneath before the thing that abstracts it, unless the underneath thing is genuinely obsolete.
9. So: HTML and CSS before Tailwind. JavaScript before React. SQL before an ORM. HTTP before a framework's router. Git before any hosting website.
10. The exception is real. Nobody should learn assembly before Python, or Objective-C before Swift. The rule applies one layer down, not all the way.
11. TypeScript timing: learn it after stage 2, not during. Types over a language you cannot yet read are noise. Once you can read JavaScript, TypeScript takes about a week and pays for itself immediately.
12. On tutorials, stated plainly: following a tutorial produces the feeling of competence without the substance, because every decision was already made for you. The cure is to build the same thing again, from an empty folder, with the tutorial closed.
13. On AI assistants, separating fact from claim. Established fact: they generate working code for common tasks quickly and are excellent at explaining unfamiliar code. Active research: how much they change long-term skill acquisition in beginners; the studies are early and mixed. Marketing claim: that they remove the need to understand the fundamentals. If you cannot read the answer, you cannot tell when it is wrong, and it is confidently wrong often enough to matter.
14. A defensible use during learning: write it yourself first, then ask for a critique. That order keeps the skill and gains the speed.
15. What does not decide your career: which framework you chose. React, Vue, Svelte and Angular all teach the same underlying ideas of state, components and reconciliation, and moving between them takes a fortnight.
16. What does decide it: whether you can take a vague problem, model the data, build something that works, and explain the trade-offs you made.

35.16.6 WORDS - remember these

1. Roadmap — a picture of what to learn next — an ordered curriculum, useful as a sequence and misleading as a schedule.
2. Stage gate — a built thing that proves readiness — an objective completion criterion replacing elapsed time as the unit of progress.
3. Tutorial trap — following along and learning nothing — passive completion without independent recall or decision-making.

4. Portfolio — the few things you can show — a small set of deployed, documented, version-controlled projects with real commit history.
5. git — the version control program — a distributed content-addressable version control system, Linus Torvalds, 2005.
6. GitHub — one website that stores git projects — a hosting and collaboration platform for git repositories, founded 2008.
7. Full stack — you can do both halves — competence across the browser client, the server application and its datastore.
8. Junior-ready — good enough to be paid to learn — able to ship a reviewed feature end to end with guidance, not to work unsupervised.

35.98 Common wrong ideas

1. Wrong: HTML is a programming language and knowing it means you can program. Right: HTML is a markup language. It describes structure and has no variables, no conditions and no loops. JavaScript is the only one of the three web languages that can compute anything.
2. Wrong: `!important` fixes a specificity problem. Right: it hides one. It wins by leaving the normal cascade entirely, so the next person needs a second `!important` to override it, and after four rounds nobody can predict which rule applies. The real fix is a lower-specificity selector.
3. Wrong: `async` and `await` make JavaScript run on several threads. Right: your JavaScript still runs on exactly one thread. `await` yields the thread while waiting, so other queued work can run. Parallelism needs Web Workers.
4. Wrong: the virtual DOM is faster than touching the DOM. Right: a hand-written direct DOM update is always faster than a diff plus the same update. The virtual DOM buys predictable performance from declarative code, not raw speed, and its own authors have said so.
5. Wrong: you need a framework to build a website. Right: a framework earns its place when many pieces of shared state must stay in sync. A content site, a form, a landing page and a blog are all better plain, and ship faster.
6. Wrong: MongoDB should be your first database because it scales. Right: PostgreSQL handles the workload of almost every application ever written, has transactions and joins, and has a `jsonb` column type when you genuinely need flexible documents. Learn relational modelling first; it transfers everywhere.
7. Wrong: hashing a password with SHA-256 is secure because SHA-256 is a strong hash. Right: SHA-256 is strong and fast, and fast is exactly the wrong property. A GPU tries billions of SHA-256 guesses a second. Use bcrypt, scrypt or Argon2id with a per-user salt, tuned to take about 100 milliseconds.
8. Wrong: JSON Web Tokens are more secure than sessions. Right: they are a different trade, not a better one. They remove the server lookup and, with it, the ability to revoke. Sessions in an `HttpOnly` cookie are the safer default, and a JWT in `localStorage` is stolen by the first cross-site scripting bug.
9. Wrong: microservices scale better, so start with them. Right: they let teams deploy independently, which is an organizational benefit with a technical cost. Every in-process call becomes a network call that can fail, and one ACID transaction becomes a saga. Start with a modular monolith and split when a specific team or scaling boundary forces it.
10. Wrong: accessibility is a polish step near launch, and mostly means adding ARIA attributes. Right: it is decided by the tags you choose on the first day. A real `<button>` arrives with a role, keyboard focus and Enter handling; a styled `<div>` arrives with none. The W3C's own first rule of ARIA is not to use ARIA when a native element already does the job.

35.99 Chapter summary in 20 lines

1. A web application runs in two places: a frontend on the user's device that you do not control, and a backend on a server that you do. Never trust the first.
2. HTML gives structure, CSS gives presentation, JavaScript gives behaviour, and only JavaScript is a programming language.
3. Semantic HTML tags are not decoration. They set the role, keyboard behaviour and screen-reader name of everything on the page, for free.
4. CSS resolves conflicts by origin, then specificity counted as inline, id, class and element, then source order. The box model, flexbox and grid replace guessing with rules.
5. JavaScript has one thread, a call stack, a task queue and a microtask queue. Microtasks drain completely before the next task, which is why promise callbacks beat `setTimeout(0)` every time.
6. The browser parses HTML into the DOM and CSS into the CSSOM, combines them into a render tree, then runs layout, paint and composite. Changing geometry forces a reflow; changing transform and opacity does not.
7. The browser is a platform in its own right: storage, workers, fetch, canvas, WebGL and WebAssembly, all inside a same-origin sandbox that CORS relaxes deliberately and explicitly.
8. A frontend framework's real product is keeping the screen in agreement with the state. Virtual DOM diffing, fine-grained signals and compilation are three answers to that one problem.
9. Build tooling exists because browsers want a few small optimized files while humans want many readable ones: bundling, transpiling, minifying, tree shaking and content-hashed names for caching.
10. A backend program listens on a port, parses HTTP, routes the request through middleware to a handler, talks to a database, and serializes a reply. Every framework in every language is that same loop.
11. Relational databases store facts once, in typed tables, joined on keys, with B+ tree indexes turning scans into lookups and ACID transactions making partial failure impossible.
12. Choose a datastore by workload, not by fashion. PostgreSQL is the correct default; Redis, Elasticsearch, Cassandra and Neo4j each answer a specific question that PostgreSQL answers less well.
13. An ORM removes repetitive code and adds the N+1 query trap. Use it for the ordinary ninety percent, read the SQL it generates, and write SQL yourself when the query is the point.
14. Store passwords with a deliberately slow, salted hash: Argon2id or bcrypt, never a bare fast hash, and never the password itself.
15. Sessions keep state on the server and can be revoked instantly. Tokens keep state with the client and scale without a lookup. Pick the trade knowingly, and keep credentials in `HttpOnly` cookies.
16. OAuth 2.0 delegates authorization through an authorization code exchanged on a back channel with PKCE; OpenID Connect adds the ID token that turns it into a login system. Authentication is who you are; authorization is what you may do, and RBAC keeps the second one auditable.
17. Architecture is caching at the right layer, moving slow work to a queue, choosing polling, server-sent events or WebSockets on merit, rate limiting every public endpoint, and starting with a monolith you could split later.
18. Mobile means choosing between native Swift or Kotlin, cross-platform React Native or Flutter, and an installable web app, then living with a store review on every release. An APK and an IPA are both ZIP archives, which is Chapter 20's lesson that the bytes, not the extension, define a file.
19. Quality is measurable: a pyramid of fast unit tests under fewer browser tests, LCP at 2.5 seconds, INP at 200 milliseconds, CLS at 0.1 measured at the 75th percentile of real users, and WCAG 2.2 level AA as the floor rather than the ambition.
20. Learn in stages defined by what you have built, not in days spent: tools, then pages, then the language, then SQL, then a server, then a framework, then shipping, then depth forever.

36

CHAPTER

KB

APIs — What They Are and Where They Came From

36.0 What this chapter gives you

1. You will be able to say what an API actually is in one sentence, and to defend that sentence: it is a published agreement, not a technology.
2. You will be able to point at an API at five different scales, from one function inside one file up to a service on the other side of the planet.
3. You will be able to tell the real history, with names and years: subroutine libraries in 1951, the term itself in 1968, remote procedure calls in 1984, CORBA in 1991, SOAP and REST around 2000, GraphQL and gRPC after 2015.
4. You will be able to explain the difference between an API and an ABI, and say why a program can compile fine and still crash at run time.
5. You will be able to read semantic versioning and say exactly what a change in each of the three numbers promises you.
6. You will be able to list the eight fallacies of distributed computing and say what each one costs you when you believe it.
7. You will be able to write out a raw HTTP request and response by hand, name every part, and pick the right status code for a situation.
8. You will be able to state what Roy Fielding actually specified in 2000, and why almost nothing sold as REST today meets that specification.
9. You will be able to compare REST, GraphQL, gRPC, SOAP, JSON-RPC, WebSockets and server-sent events on payload size, typing, streaming and tooling.
10. You will be able to walk through the OAuth 2.0 authorization code flow with PKCE step by step, decode a JWT field by field, and say what a scope does.
11. You will be able to make a client that does not fall over: retries with backoff and jitter, idempotency keys, timeouts and circuit breakers.
12. You will be able to design an API other people can use, document it with OpenAPI, and explain why some companies live or die by that decision.

36.1 What an API actually is

36.1.1 PLAIN – in simple words

1. Software is made of parts. Parts need to ask each other to do things.
2. An **API** is the published agreement about how one part may ask another part to do something.
3. The letters stand for **application programming interface**. That name is not very helpful, so ignore the letters and remember the idea: an agreement.
4. The agreement says three things. What you may ask for. How you must ask. What you will get back.

5. It also says, quietly, one more thing: everything not written down is not promised, and may change without warning.
6. An API is not a program. It is the shape of the doorway into a program.
7. The most important word in this chapter is **contract**. An API is a contract between the person who wrote a piece of software and the person who wants to use it.
8. Because it is a contract and not a technology, an API can be made of almost anything. Text over a wire. A name in a file. A number in a register.
9. So the sentence "we built an API" tells you nothing about the technology. It only tells you that somebody wrote down a promise.

36.1.2 PLAIN - a picture in your head

1. Think of a restaurant kitchen. You are hungry. The kitchen can cook.
2. You do not walk into the kitchen. You are handed a menu.
3. The menu is the API. It lists exactly what you may ask for. It gives the name of each dish and what comes with it.
4. You do not need to know whether the soup is made from a family recipe or from a packet. You do not need to know who the cook is today.
5. The kitchen can throw out its old stove and buy a new one. Your order form does not change. That is the whole benefit in one line.
6. And if the menu says "soup" but the kitchen sends a sandwich, the kitchen has broken the contract, even if the sandwich is delicious.

Where this comparison breaks:

7. A menu is a suggestion. An API is checked by a machine, and a machine has no good manners. A single wrong letter in the order and nothing is served.
8. A restaurant serves one customer at a time at one table. A busy API may be serving fifty thousand callers in the same second, and they can interfere with each other in ways that diners cannot.
9. A menu never fails halfway. An API call over a network can be taken, cooked, and then lost on the way back, which is a case with no restaurant equivalent. Section 36.5 deals with exactly that case.

36.1.3 PLAIN - a worked example

1. Here is the same idea, an API, written at five completely different scales. All five are APIs. None of them is more of an API than the others.

Scale	The published agreement
One function	`int strlen(const char *s)`
One library	`curl_easy_setopt(h, opt, v)`
The kernel	`write(fd, buf, count)`
A GPU driver	`vkQueueSubmit(queue, n, ...)`
The internet	`GET /repos/{owner}/{repo}`

2. Scale one is inside one program. The function `strlen` promises: give me the address of a run of text ending in a zero byte, and I return its length.
3. Scale two is a library on your machine. The C library libcurl promises: set this option to this value on this handle, and I return zero if I accepted it.
4. Scale three crosses the boundary into the operating system. The `write` call promises: I will try to write that many bytes, and tell you how many I wrote.
5. Scale four crosses into a device driver and then into hardware. Vulkan promises: I will hand this batch of work to the graphics card's queue.

6. Scale five crosses the planet. The GitHub HTTP API promises: ask me for that path with a valid token and I will return a JSON description of that repository.
7. Notice what changed across the five, and what did not. The technology changed completely. The idea did not change at all.
8. What did change is how much can go wrong. At scale one, essentially nothing. At scale five, everything. That is section 36.5.

36.1.4 PLAIN - what is really happening inside

1. An API has three parts, and all three matter.
2. The **surface** is the list of things you can call, and their names.
3. The **shape** of each call is the data you must supply and the data you get back: how many items, of what kind, in what order, in what units.
4. The **behaviour** is what the call actually does, including how it fails, how long it may take, and whether calling it twice is different from once.
5. Beginners think an API is only the surface. Most real bugs live in the behaviour.
6. The surface can be checked by a machine. A compiler will refuse to build code that calls `strlen` with two arguments.
7. The shape can be partly checked by a machine. Types catch some mistakes.
8. The behaviour usually cannot be checked at all. Nothing stops a function documented as "returns quickly" from taking nine seconds.
9. So a good API is a small surface, a strict shape, and behaviour written down in plain sentences that a person actually reads.

The honest version: the contract also includes things nobody wrote down.

10. If your code depends on a list coming back in alphabetical order, and the documentation never promised an order, you have invented a clause.
11. When the other side changes and your code breaks, both of you will believe you were right. This is the single most common source of API arguments.
12. The rule that resolves it: if it is not in the document, it is not promised, no matter how long it has been true in practice.

36.1.5 TECHNICAL - the engineer's version

1. Formally, an API is an **interface specification**: a set of named operations, each with a signature, a set of pre-conditions and post-conditions, and an error model.
2. Two systems that share an interface specification are said to be **interoperable** at that interface. Neither needs the other's source.
3. The specification may be enforced statically (the compiler rejects bad calls), dynamically (the runtime throws), or not at all (documentation only).
4. The formal notion behind this is **design by contract**, named by Bertrand Meyer for the Eiffel language in the 1980s: pre-conditions, post-conditions and invariants stated as part of the interface.
5. An API is distinct from a **protocol**. A protocol specifies bytes on a wire and the ordering rules between peers. An API specifies operations available to a caller. HTTP is a protocol; the GitHub REST API is an API that uses it.
6. An API is distinct from an **ABI** (application binary interface), which fixes register use, calling convention, structure layout and symbol naming. Section 36.3 measures the difference.
7. Real interface surfaces, for scale. These are order-of-magnitude figures from published documentation and headers, not exact counts, because every one of them depends on version and configuration.

Interface	Rough size of surface
C standard library	about 200 functions
Linux system calls	about 350 on x86-64
Win32 core DLLs	thousands of functions
POSIX.1-2024	over 1,200 interfaces
GitHub REST API	over 1,000 endpoints

8. Tools that let you observe an interface directly: `nm -D libc.so.6` lists exported symbols, `objdump -T` shows the dynamic symbol table, `man 2 syscalls` lists Linux system calls, `strace` records them as they happen, and `curl -v` prints an HTTP exchange in full.

36.1.6 WORDS - remember these

1. API — a published agreement about how to ask software to do something — an interface specification of operations, signatures and error behaviour.
2. Contract — a promise both sides rely on — the pre-conditions, post-conditions and invariants attached to an interface.
3. Interface — the doorway into a component — the named boundary across which two components interact without sharing internals.
4. Protocol — the rules for talking on a wire — the byte format and message ordering between peers, independent of any one caller's API.
5. Implementation detail — how it happens to work today — behaviour observable but not specified, therefore free to change without notice.
6. Interoperable — two things can work together — able to interact correctly through a shared interface specification without shared source code.

36.2 The history, properly

36.2.1 PLAIN - in simple words

1. The reader asked how APIs were invented. The honest answer is that nobody invented them in one moment. They were arrived at, in stages, over 75 years.
2. Stage one, around 1951. People noticed they were writing the same little piece of code again and again, so they saved it and reused it.
3. That reusable piece was called a **subroutine**. To reuse it you needed to know what to give it and what it gave back. That written-down knowledge was the first API, before anybody had a name for it.
4. Stage two, the 1960s. Computers grew operating systems. A program could no longer touch the disk directly. It had to ask the operating system.
5. That asking needed rules, so operating systems published their own lists of calls. The name **application program interface** appears in print in 1968.
6. Stage three, the 1980s. Machines were joined by networks. Somebody asked: why can we not call a piece of code on another machine the same way we call one on this machine?
7. That idea is the **remote procedure call**. It was tried properly in 1984, and it worked, and it also created every hard problem in the rest of this chapter.
8. Stage four, the 1990s. Big vendors built big frameworks to make remote calls easy across languages. They were called CORBA and DCOM. They mostly failed.
9. Stage five, from 1998 to 2002. People gave up on frameworks and used the web instead: plain text over HTTP, through the same port a browser uses.
10. Stage six, the 2000s. Companies started publishing these web interfaces to the outside world, so that strangers could build on top of their business.
11. Stage seven, since 2015. New shapes appeared for particular problems: GraphQL for clients that want to choose their own data, gRPC for fast machine-to-machine traffic inside a company.

36.2.2 PLAIN - a picture in your head

1. Think of a workshop that makes wooden chairs.
2. At first one person makes the whole chair. Everything is in their head.
3. Then somebody makes a jig, a wooden guide for cutting a leg to the right length. Now anyone can cut a leg. That is a subroutine.
4. Then a drawer of jigs appears, with labels. That is a library.
5. Then the workshop gets so big that the wood store is locked, and only a storekeeper may enter. You fill in a slip to get wood. That is a system call.
6. Then a second workshop opens across town. You want to send slips there too. Someone invents a courier. That is a remote procedure call.
7. And the moment the courier exists, new problems exist: the van breaks down, the slip arrives twice, the other workshop is closed.

Where this comparison breaks:

8. Jigs do not have versions, and a jig never returns a different result on Tuesday. Software interfaces change under you, which is section 36.3.
9. A courier who fails tells you he failed. A network can swallow a message and tell nobody, which is the hardest single fact in this chapter.

36.2.3 PLAIN - a worked example

1. Take one task and watch it move across the eras: "give me the photographs belonging to user 42".
2. In 1951, the task does not exist, but the shape does. You would look up a subroutine in a paper catalogue, copy its tape into your program, and know from the catalogue that it wanted a number in one register.
3. In 1975, on a Unix machine, you would `open` a file, `read` it, and `close` it. Three system calls, all local, all fast, all likely to succeed.
4. In 1988, on a network, you would call a function named `getPhotos(42)` that secretly sent a message to another machine. It looked local. It was not.
5. In 1995, with CORBA, you would write an interface file, run a compiler on it, and get a stub object in your language that hid the network again.
6. In 1999, with SOAP, you would send a page of XML describing a method call over HTTP, and get back a page of XML describing the answer.
7. In 2005, with a web API, you would ask for a URL and get a small block of JSON. Here is the whole thing:

```
GET /users/42/photos HTTP/1.1
Host: api.example.com

HTTP/1.1 200 OK
Content-Type: application/json

{"photos":[{"id":9,"title":"Pune, monsoon"}]}
```

8. In 2016, with GraphQL, you would send one query naming exactly the fields you wanted, and get back exactly those fields and nothing else.
9. Look at step 6 against step 7. Same task. The XML version was often two to five kilobytes; the JSON version above is under 60 bytes of payload. That size difference is a large part of why the industry moved.

36.2.4 PLAIN - what is really happening inside

1. The history is not random. Each step is a response to a specific pain.
2. Subroutines answered "I am typing the same thing repeatedly".
3. Subroutine libraries answered "I cannot find the piece I wrote last year".

4. Operating system calls answered “two programs both wrote to the disk and destroyed each other’s data”.
5. Shared libraries answered “forty programs each carry their own copy of the same code and the machine has 4 MB of memory”.
6. Remote procedure calls answered “the data I need is on another machine”.
7. CORBA and DCOM answered “my two programs are in different languages”.
8. XML over HTTP answered “corporate firewalls block everything except port 80, and I cannot get the network team to open anything”.
9. REST-style web APIs answered “SOAP is enormous and my phone has a slow connection and a small battery”.
10. GraphQL answered “my mobile screen needs six fields and the endpoint sends ninety”.
11. gRPC answered “we make two million internal calls per second and JSON parsing is now a measurable part of our electricity bill”.
12. Read that list backwards and you can predict the next step: whatever hurts most at scale today is what gets replaced next.

36.2.5 TECHNICAL - the engineer’s version

1. The timeline, with checkable dates.

Year	Event	Who
1951	Subroutine library book	Wilkes, Wheeler, Gill
1968	Term “application program interface”	Cotton, Greatorex
1974	API applied to databases	C. J. Date
1981	RPC named in a PhD thesis	Bruce Jay Nelson
1984	Working RPC published	Birrell and Nelson
1988	POSIX 1003.1 published	IEEE
1991	CORBA 1.0, October	OMG
1993	Win32 with Windows NT 3.1	Microsoft
1996	DCOM ships	Microsoft
1998	XML-RPC published	Dave Winer, UserLand
2000	SOAP 1.1 W3C Note, 8 May	W3C submission
2000	REST defined in a thesis	Roy Fielding
2000	First commercial web API, 7 Feb	Salesforce
2000	Web API, 20 November	eBay
2002	Amazon Web Services, 16 July	Amazon
2004	Photo API	Flickr
2006	Platform API, August	Facebook
2006	Public API, September	Twitter
2012	OAuth 2.0, RFC 6749, October	IETF
2015	GraphQL specification opened	Facebook
2016	gRPC 1.0, 23 August	Google

2. **1951.** Maurice Wilkes, David Wheeler and Stanley Gill published *The Preparation of Programs for an Electronic Digital Computer* at Cambridge, generally called the first book on programming. It grew out of a September 1950 internal report on the EDSAC subroutine library. The subroutines lived on punched paper tape in a filing cabinet, and the catalogue describing how to call each one is, in every meaningful sense, the first API document.
3. **1968.** The term “application program interface”, without the “-ing”, appears in the paper *Data structures and techniques for remote computer graphics* by Ira W. Cotton and Frank S. Greatorex, presented at the AFIPS Fall Joint Computer Conference held 9 to 11 December 1968. It described the boundary that

gave a graphics program independence from the hardware. This is the commonly cited first published use. It is a first citation, not a moment of invention: the practice was already twenty years old.

4. **1974.** C. J. Date's paper *The Relational and Network Approaches: Comparison of the Application Programming Interface* carried the idea into databases, and the "-ing" form became standard.
5. **1981 and 1984.** Bruce Jay Nelson's doctoral work at Carnegie Mellon named the remote procedure call. Andrew Birrell and Nelson then built one at Xerox PARC, reported in Xerox technical report CSL-83-7 in December 1983 and published as *Implementing Remote Procedure Calls* in ACM Transactions on Computer Systems, volume 2, number 1, February 1984. It ran on the Cedar system over Ethernet, and reported a null local call at a few microseconds against a null remote call at about 1.1 milliseconds. That ratio is the whole story of distributed systems in one measurement.
6. **1988.** IEEE Std 1003.1-1988 standardized the Unix system call interface. Richard Stallman suggested the name POSIX in place of IEEE-IX.
7. **1991 to 1997.** The Object Management Group, founded 1989, published CORBA 1.0 in October 1991 and CORBA 2.0 with the IIOP wire protocol in 1997. Microsoft shipped DCOM in 1996. Both aimed at language-independent remote objects.
8. Why they lost, in the analysis of Michi Henning's 2006 ACM Queue article *The Rise and Fall of CORBA*: specifications too large and partly never implemented; a committee process driven by vendor politics; no answer to firewalls; and, for DCOM, Windows only. Experts still disagree on the weighting. Henning stresses process failure; Douglas Schmidt's published reply stresses that the technology worked and the market moved.
9. There is a second reason, and it is the deeper one. Both tried to make a remote call look exactly like a local call. Section 36.5 explains why that is a lie that cannot be sustained.
10. **1998 to 2000.** Dave Winer of UserLand shipped XML-RPC in 1998, subset from an unshipped 1998 SOAP draft. SOAP 1.1 was published as a W3C Note on 8 May 2000, and SOAP 1.2 became a W3C Recommendation on 24 June 2003.
11. **2000.** Roy Thomas Fielding's doctoral dissertation *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, 2000, supervised by Professor Richard N. Taylor, defined REST in Chapter 5. Fielding was a co-author of the HTTP specifications, so this was a description of the web's actual architecture, derived, not invented. Section 36.7 covers what it says.
12. **The web API era.** Salesforce demonstrated an XML API at the IDG Demo conference on 7 February 2000, widely called the first commercial web API. eBay launched its API and developer programme on 20 November 2000. Amazon launched Amazon Web Services on 16 July 2002. Flickr's API followed its February 2004 launch. Facebook's platform arrived in August 2006 and Twitter's API in September 2006.
13. **2015 to 2016.** GraphQL was built inside Facebook in 2012 for its mobile apps, and the specification and reference implementation were opened during 2015, announced publicly on 14 September 2015. Google built gRPC from its internal Stubby system, announced it in 2015, and released version 1.0 on 23 August 2016.

36.2.6 WORDS - remember these

1. Subroutine — a saved, reusable piece of code — a callable block with a defined entry point, parameters and return convention.
2. Subroutine library — a labelled drawer of saved pieces — an indexed collection of reusable routines with published calling documentation.
3. RPC — calling code on another machine like a local function — remote procedure call, with client stub, marshalling, transport and server skeleton.
4. Marshalling — packing arguments for travel — serializing typed values into a byte stream and reconstructing them at the far end.
5. CORBA — a 1990s cross-language remote object standard — Common Object Request Broker Architecture, OMG, CORBA 1.0 in October 1991.
6. IDL — a file describing an interface once, for many languages — interface definition language, compiled into stubs and skeletons.
7. SOAP — a heavy XML message format for method calls — W3C Note in May 2000, Recommendation 1.2 in June 2003, usually carried over HTTP.

36.3 Local APIs: signatures, libraries and versions

36.3.1 PLAIN - in simple words

1. The simplest API is a function inside a program you are writing.
2. Its **signature** is its name, the things it takes, and the thing it gives back. That single line is the whole contract for a lot of code.
3. When you package many functions together for other people to use, you have a **library**.
4. To use a library in C or C++ you need a **header file**: a file that lists the signatures without the code, so the compiler knows what is allowed.
5. In other languages the same job is done by import statements, package manifests or type definition files. The idea is identical.
6. Once other people depend on your library, you are stuck with your own promises. Changing them breaks their programs.
7. So library authors need a way to say "this change is safe" or "this change will break you". That way is **versioning**.
8. A change that breaks existing callers is called a **breaking change**. The whole discipline of versioning exists to warn about exactly that.

36.3.2 PLAIN - a picture in your head

1. Think of a plug and a socket in a wall.
2. The API is the shape of the plug: three pins, in that arrangement, that size.
3. Any lamp with that plug shape works. The wall does not care what the lamp is.
4. Now imagine the electricity company changes the voltage from 230 to 400 while keeping the socket shape identical.
5. Your plug still fits. Your lamp explodes.
6. That is the difference between the shape of the interface and its behaviour, and it is exactly how most painful breakages feel.

Where this comparison breaks:

7. Sockets are regulated by law and change once a century. Libraries change weekly, and any one program may depend on two hundred of them.
8. A plug either fits or does not. A software interface can half-fit: it compiles, links, runs for six weeks, and then fails on one unusual input.

36.3.3 PLAIN - a worked example

1. Here is a C function signature and every promise inside it.

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

2. `fread` is the name. That is the only part most people remember.
3. `void *ptr` is where to put the data. `void *` means "any kind of thing", so the compiler will not check what you point at. That is a hole.
4. `size_t size` is how big one item is, in bytes. `size_t n` is how many.
5. `FILE *stream` is the open file to read from.
6. `size_t` on the left is the return: the number of complete items actually read, which may be fewer than `n`. Not bytes. Items.
7. That last point is a real bug factory. The type says nothing about it. Only the documentation does.
8. Now the versioning question. Suppose a library goes from 2.4.1 to 3.0.0. Here is exactly what each position promises, under semantic versioning.

Change	Example	What it promises
Patch	2.4.1 to 2.4.2	Bug fix only, safe
Minor	2.4.1 to 2.5.0	New features, still safe
Major	2.4.1 to 3.0.0	Something broke, read notes

9. Real cases. Python 2 to Python 3, released December 2008, changed `print` from a statement to a function and forced a decade-long migration. That is a major change and the number said so.
10. Adding a new optional argument with a default value is normally a minor change: old callers keep working.
11. Removing a function, renaming it, changing an argument's type, or changing what a return value means are all major changes, even when the code still compiles.

36.3.4 PLAIN - what is really happening inside

1. There are two different boundaries and people confuse them constantly.
2. The **API** is what the compiler checks: names, argument counts, types.
3. The **ABI** is what the machine checks at run time: which register holds which argument, how big a structure is, where each field sits inside it, what the function symbol is called in the compiled file.
4. You can keep the API identical and break the ABI. Add one field to the middle of a structure and every already-compiled program that uses that structure now reads the wrong bytes.
5. Nothing warns you. The program starts. Then it produces nonsense, or dies.
6. That is why shared libraries carry a separate number in the file name, like `libssl.so.3`. That number is the ABI version, not the product version.
7. Rebuilding from source fixes an ABI break. It cannot fix an API break, because the source itself no longer matches.
8. So the two rules are: change the API and callers must edit their code; change the ABI and callers must at least rebuild.

36.3.5 TECHNICAL - the engineer's version

1. Semantic Versioning, written by Tom Preston-Werner, co-founder of GitHub, published as version 1.0.0-beta in 2010 and version 2.0.0 in 2013, states the rule as MAJOR.MINOR.PATCH.
2. It is a **convention**, not a standard body's specification. Package managers such as npm, Cargo and Go modules enforce it mechanically, which makes it feel like a law. Many projects publish semver numbers and break them anyway.

3. The exact promises from the specification: MAJOR when you make incompatible API changes, MINOR when you add functionality in a backward-compatible way, PATCH when you make backward-compatible bug fixes.
4. Version 0.y.z means anything may change at any time. That clause is heavily used and heavily abused.
5. Go modules make major versions part of the import path from v2 onward, so `example.com/pkg/v2` and `example.com/pkg` can coexist in one build. That is a deliberate answer to the diamond dependency problem.
6. ABI stability in practice, with real numbers:

Interface	ABI policy
Linux kernel to userspace	Stable, never break it
Linux internal module ABI	No stability promise
glibc	Stable, symbol versioning
C++ standard library	Compiler and flag dependent
Windows Win32	Stable since 1993

7. glibc uses **symbol versioning**: one shared object can export `memcpy@GLIBC_2.2.5` and a newer variant at once, so old binaries keep the old behaviour. Inspect it with `objdump -T /lib/x86_64-linux-gnu/libc.so.6`.
8. Linus Torvalds's rule for the kernel, stated repeatedly on the kernel mailing list since the 1990s, is that breaking userspace is never acceptable, whatever the technical merit. That policy is why binaries from 1998 still run.
9. Tools: `abidiff` from libabigail compares two builds and reports ABI changes; `nm -D --defined-only` lists exported symbols; `readelf -d` shows the SONAME that encodes the ABI version; `pkg-config --modversion` reports the declared version.
10. Deprecation in code is itself part of the contract: `__attribute__((deprecated))` in GCC and Clang, `@Deprecated` in Java, `#[deprecated]` in Rust, `DeprecationWarning` in Python. A warning is a promise of a future break.

36.3.6 WORDS - remember these

1. Signature — the name, inputs and output of a function — the typed interface of a callable, checked by the compiler at the call site.
2. Header — a file listing signatures without code — a declaration-only translation unit included to satisfy the compiler.
3. ABI — the machine-level agreement — application binary interface: calling convention, register use, structure layout and symbol naming.
4. Breaking change — a change that makes working code stop working — any change violating the published compatibility promise for that version position.
5. Semantic versioning — three numbers that say how risky an upgrade is — MAJOR.MINOR.PATCH per the SemVer 2.0.0 convention of 2013.
6. Symbol versioning — one library holding old and new behaviour at once — versioned symbol names in ELF so old binaries bind to old semantics.
7. Deprecation — a warning that something will be removed — a documented period during which an interface still works but is scheduled for removal.

36.4 System call APIs

36.4.1 PLAIN - in simple words

1. Chapter 18 covered operating systems. Here is the one-line recap needed now.
2. A running program is not allowed to touch the disk, the network card or another program's memory. The operating system owns those.
3. So the program asks. That request is a **system call**, and the list of possible requests is the operating system's API.
4. This is the most important API on any machine, because everything else is built on top of it.
5. Different operating systems chose different lists. That is why a program written for one does not simply run on another.
6. To fix that, people wrote down a common list that many systems agreed to provide. That agreement is called **POSIX**.
7. Windows did not follow POSIX. It has its own list, called **Win32**.
8. So writing a file looks different on each, and that difference is the reason for a huge amount of the software world's duplicated effort.

36.4.2 PLAIN - a picture in your head

1. Think of a bank counter with a thick glass screen.
2. You cannot reach the money. You write on a slip, push it under the glass, and the clerk does the work.
3. The list of slips the bank accepts is the system call API. There are maybe 350 kinds of slip on Linux.
4. POSIX is like an agreement between many banks in many countries to accept the same slips, so a traveller can use any of them.
5. Win32 is a bank that never signed that agreement and prints its own slips, which are longer and ask for more detail.

Where this comparison breaks:

6. A bank clerk can ask you what you meant. A kernel cannot. A malformed slip returns a number and nothing else.
7. A bank does one thing at a time per counter. A kernel handles millions of these per second and must keep every program's slips separate and safe.

36.4.3 PLAIN - a worked example

1. Writing the five bytes "hello" to a new file, on both systems, cut to the bone.

```
/* POSIX */
int fd = open("out.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
write(fd, "hello", 5);
close(fd);
```

```
/* Win32 */
HANDLE h = CreateFileA("out.txt", GENERIC_WRITE, 0, NULL,
                       CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
DWORD written;
WriteFile(h, "hello", 5, &written, NULL);
CloseHandle(h);
```

2. Same task. Same result on disk. Completely different shape.
3. POSIX gives back a small integer, the **file descriptor**, usually 3 for your first opened file, since 0, 1 and 2 are already taken by input, output and errors.
4. Win32 gives back a **HANDLE**, an opaque value that is not an index into anything you can reason about.
5. POSIX signals failure by returning -1 and setting a separate global number called `errno`. Win32 returns a special invalid handle value and you call `GetLastError()`.
6. POSIX packs the options into one integer of flags. Win32 spreads them across seven separate arguments, several of which are usually zero or NULL.

7. Notice `write` returns how many bytes it wrote. It may write fewer than you asked. Ignoring that return value is one of the classic C bugs.

36.4.4 PLAIN - what is really happening inside

1. A system call is not an ordinary function call. It crosses a privilege boundary, and that is deliberate and expensive.
2. Your program puts a number identifying the call into a register, puts the arguments into other registers, and executes one special instruction.
3. On x86-64 Linux that instruction is `syscall`, and the call number goes in register `rax`. Write is number 1.
4. The processor switches from user mode to kernel mode, jumps to a fixed handler, and the kernel checks everything you passed.
5. The kernel does the work, puts a result back in a register, and returns to user mode.
6. The C library function `write()` is a thin wrapper around this. It is not the system call itself; it is a normal function that performs one.
7. POSIX does not define the instruction, the register or the call number. It defines the C function and its behaviour. That is the portability layer.
8. So POSIX is an API standard, not an ABI standard: source portability, not binary portability. Code recompiles; compiled binaries do not travel.

36.4.5 TECHNICAL - the engineer's version

1. POSIX is IEEE Std 1003.1, first published 1988. The current joint text is maintained by the Austin Group as the Single UNIX Specification and IEEE Std 1003.1; the 2024 edition is the current revision at the time of writing in 2026.
2. Real conformance, honestly stated: Linux is broadly POSIX-compatible but has never been certified. macOS is certified UNIX. AIX and z/OS are certified. Windows is not POSIX and dropped its old POSIX subsystem long ago.
3. Windows Subsystem for Linux version 2, shipped from 2019, runs a real Linux kernel in a lightweight virtual machine rather than translating calls. That is why its system call fidelity is high and its file access across the Windows boundary is slow.
4. Comparison of the two families at the same job:

Task	POSIX	Win32
Open a file	`open`	`CreateFileW`
Write bytes	`write`	`WriteFile`
Handle type	`int` descriptor	`HANDLE`
Error report	`errno`	`GetLastError`
Start a process	`fork` + `exec`	`CreateProcess`
Load a library	`dlopen`	`LoadLibrary`
Memory map	`mmap`	`MapViewOfFile`

5. Cost, measured. A system call on modern x86-64 Linux costs roughly 100 to 300 nanoseconds when nothing goes wrong, against roughly 1 to 2 nanoseconds for a plain function call. Kernel page-table isolation, added in January 2018 against the Meltdown vulnerability, increased that cost measurably on affected processors.
6. This is why high-throughput code avoids system calls: buffered I/O to batch `write` calls, `io_uring` on Linux since kernel 5.1 in 2019 to submit many operations with one crossing, and kernel bypass networking for the extreme cases.

7. Observation tools: `strace -c ./prog` counts and times every system call; `ltrace` does the same for library calls; `dtruss` is the macOS equivalent; Windows uses Process Monitor and Event Tracing for Windows.
8. Linux keeps roughly 350 system calls on x86-64, and the list is architecture-specific. `ausyscall --dump` or the table in `arch/x86/entry/syscalls/syscall_64.tbl` in the kernel source gives the numbers.

36.4.6 WORDS - remember these

1. System call — a request from a program to the operating system — a controlled entry into kernel mode with checked arguments.
2. File descriptor — the small number that stands for an open file — a per process index into the kernel's open file table, an `int` on POSIX.
3. POSIX — the agreed common list of Unix-style calls — IEEE Std 1003.1, first published 1988, source-level portability across conforming systems.
4. Win32 — the Windows system interface — the C API exported by kernel32, user32, gdi32 and friends, ABI-stable since Windows NT 3.1 in 1993.
5. User mode and kernel mode — the two privilege levels — CPU rings restricting which instructions and memory a running thread may touch.
6. Wrapper — the ordinary function that performs the special instruction — the libc stub that loads the call number and executes `syscall`.

36.5 Remote APIs: what the network changes

36.5.1 PLAIN - in simple words

1. This is the single biggest conceptual jump in a developer's life, and most people make it without noticing, which is why it hurts.
2. A local function call either happens or the program crashes. There is no third case.
3. A remote call has a third case, and a fourth, and a fifth.
4. It can fail before arriving. It can arrive and fail. It can arrive, succeed, and the answer can be lost on the way home.
5. In that last case, the work was done and you do not know it. You cannot know it. There is no way to find out from the failure itself.
6. This is called **partial failure**, and it does not exist inside one machine.
7. Also, remote calls take real time. A local call costs about a nanosecond. A call across a city costs about ten million nanoseconds.
8. That is a factor of ten million. No amount of clever coding hides a factor of ten million.
9. So the rule is: never pretend a remote call is a local call. Every framework that tried to hide the difference eventually failed at exactly this point.

36.5.2 PLAIN - a picture in your head

1. Asking a colleague sitting beside you for a number is a local call. You ask, they answer, you carry on.
2. Now the colleague moves to another building and you must send letters.
3. You post a letter asking for the number. Silence for a week.
4. Did the letter get lost? Did they get it and not reply yet? Did they reply and that letter got lost? Did they leave the company?
5. You cannot tell. All four look identical from your desk.
6. Now suppose the letter asked them to transfer money. Do you send a second letter? If the first one arrived, you have transferred twice.
7. That is the entire problem, and section 36.10 gives the fix, which is to put a unique reference number on the letter.

Where this comparison breaks:

8. Letters arrive in days. Network messages arrive in milliseconds, which tempts you into thinking they are instant. They are not; they are just fast.
9. The post has no equivalent of a message being delivered twice because your own side automatically resent it. Networks do that constantly.

36.5.3 PLAIN – a worked example

1. The reader's own session shows this exactly. From a flat in India, `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and then timed out after 15 seconds with no response of any kind.
2. Nothing came back. No refusal, no error message from the network, nothing. Silence.
3. On the same phone, over mobile data, the same site loaded instantly.
4. Now ask the question that matters. Was GitHub down? No: the mobile test proves it was serving.
5. Was the request received? Unknown, and unknowable from the client side.
6. For a `GET` that does not matter, because reading a page twice is harmless.
7. For a `POST` that creates something, it matters enormously.
8. In the same session, two `git push` operations failed with `send-pack: unexpected disconnect while reading sideband packet`. Both times the push had not been applied, but that could not be known from the error.
9. The disconnect happened while reading the *response*. A lost response tells you nothing about whether the *write* landed. The only way to find out was to ask the server again afterwards.
10. That is partial failure, observed in a real session, on a real evening.

36.5.4 PLAIN – what is really happening inside

1. Here is the timeline of one remote call, with the failure windows marked.

```
client                                              server
  |                                                   |
  |--(1) request leaves ----------------------------->|
  |        [lost here -> nothing happened]            |
  |                                                   |
  |                          (2) server does work     |
  |                          [crashes here -> ???]    |
  |                                                   |
  |<-(3) response returns ---------------------------|
  |        [lost here -> IT HAPPENED, you dont know]  |
  |                                                   |
  |  (4) timeout fires. Client sees identical error   |
  |      for cases 1, 2 and 3.                        |
```

2. Case 1 and case 3 produce the same visible result at the client: a timeout.
3. In case 1 retrying is correct and free. In case 3 retrying may double the effect. The client cannot tell them apart.
4. This is not a bug in any particular system. It is a proved limit. The general result is that no protocol can guarantee both sides agree, in the presence of arbitrary message loss.
5. So the industry does not solve it. It works around it, in three ways.
6. Make operations **idempotent**, so that doing them twice equals doing them once. Then retrying is always safe.
7. Or attach a unique key to each attempt, so the server can recognize a repeat and return the first answer instead of acting again.
8. Or accept the risk and reconcile later, which is what accounting systems did for centuries before computers.

36.5.5 TECHNICAL - the engineer's version

1. The eight fallacies of distributed computing. The list originated at Sun Microsystems: L. Peter Deutsch drafted seven in 1994, drawing on earlier work by Bill Joy and Dave Lyon, and James Gosling added the eighth around 1997. Attribution of individual items is disputed in the sources; the list is not.

#	Fallacy	What it costs you
1	The network is reliable	No retry path
2	Latency is zero	Chatty designs
3	Bandwidth is infinite	Huge payloads
4	The network is secure	Plaintext, no auth
5	Topology does not change	Hardcoded addresses
6	There is one administrator	No cross-team plan
7	Transport cost is zero	Ignored serialization
8	The network is homogeneous	Protocol assumptions

2. Fallacy 1. Packets are dropped, links flap, and TCP connections are reset by middleboxes. Every remote call needs an explicit failure path, not an exception handler bolted on afterwards.
3. Fallacy 2. Latency has a floor set by physics. Light in fibre travels about 200,000 km per second, so Mumbai to London and back, about 14,400 km of fibre, cannot beat roughly 72 ms of pure propagation, and measures 110 to 130 ms in practice. A design that makes 50 sequential calls has bought itself six seconds before writing a line of logic.
4. Fallacy 3. Bandwidth is finite and often asymmetric, and mobile uplinks are far slower than downlinks. Payload size is a design decision, not an afterthought.
5. Fallacy 4. Everything crossing a network is hostile territory. This is why Chapter 32 covered TLS and why section 36.9 covers tokens.
6. Fallacy 5. Addresses change. Containers get new addresses on every restart, which is why service discovery and DNS-based routing exist.
7. Fallacy 6. Your call crosses networks owned by people you will never meet. In the reader's traceroute, the path crossed the home router, the ISP's private core at addresses like `172.31.0.17`, the ISP's public edge at `137.97.29.249`, and then Microsoft's backbone through Delhi, Mumbai and Pune. That is at least three administrative domains, none of which the reader could telephone.
8. Fallacy 7. Serializing and deserializing costs CPU and memory. At Google scale this is why Protocol Buffers exist; JSON parsing at millions of requests per second is a real budget line.
9. Fallacy 8. Different operating systems, TLS versions, HTTP versions, proxies and MTUs. A path that works over one network may silently fail over another, which is precisely what the reader observed between broadband and mobile.
10. The standard model: **at-most-once** delivery risks losing work, **at-least-once** risks duplicating it, and **exactly-once** delivery is not achievable at the transport level. What is achievable is exactly-once *effect*, built from at-least-once delivery plus idempotency. Say it that way and you will never be wrong.

36.5.6 WORDS - remember these

1. Partial failure — the work happened but you never found out — a call whose outcome is unknown to the caller because the response was lost.
2. Idempotent — doing it twice is the same as doing it once — an operation whose repeated application produces the same state as a single application.
3. Latency — the waiting time before an answer starts — round-trip time, floored by propagation delay and inflated by queueing and processing.
4. Timeout — giving up after waiting — a client-side deadline that converts an unknown outcome into a definite local error.

5. At-least-once — it will arrive, maybe more than once — a delivery guarantee requiring the receiver to tolerate duplicates.
6. Exactly-once effect — the result happens once even if the message does not — at-least-once delivery combined with server-side deduplication.

36.6 HTTP APIs in practice

36.6.1 PLAIN - in simple words

1. Most APIs you meet today are carried over HTTP, the same protocol your browser uses for web pages.
2. That was not a technical decision at first. It was a political one. Port 443 was the one port that was open everywhere.
3. An HTTP exchange has exactly two messages: a **request** you send and a **response** you get back.
4. A request has four parts: a **method** saying what kind of action, a **URL** saying what to act on, **headers** giving extra facts, and an optional **body** carrying data.
5. A response has three parts: a **status code**, a three-digit number saying how it went, then headers, then an optional body.
6. The methods you need are `GET` to read, `POST` to create, `PUT` to replace, `PATCH` to change part, and `DELETE` to remove.
7. The body is almost always **JSON** these days: text that describes data using braces, square brackets, names and values.
8. That is the whole thing. Everything else in this section is detail.

36.6.2 PLAIN - a picture in your head

1. An HTTP request is a filled-in form pushed through a letterbox.
2. The method is the big word at the top: COLLECT, DEPOSIT, REPLACE, DESTROY.
3. The URL is the address line: which office, which department, which file.
4. The headers are the small boxes down the side: who you are, what format you can read, how long you will wait.
5. The body is the attached sheet, and only some kinds of form have one.
6. The response comes back with a three-digit stamp at the top. 200 means done. 404 means no such file. 500 means the office caught fire.

Where this comparison breaks:

7. Paper forms are read by a person who can guess your intent. HTTP is parsed by a machine, and an unexpected header can be silently ignored or fatal.
8. A form goes to one office. An HTTP request may pass through a browser cache, a company proxy, a content delivery network and a load balancer, any of which may answer it without the real server ever hearing about it.

36.6.3 PLAIN - a worked example

1. Here is a complete raw exchange, exactly as it goes on the wire, with every part labelled. This is what `curl -v` shows you.

```
POST /v1/orders?notify=true HTTP/1.1
Host: api.example.com
Authorization: Bearer sk_live_9f2a...
Content-Type: application/json
Accept: application/json
Content-Length: 52
Idempotency-Key: 8f14e45f-ea4f-4c1b-9b4a-0c1d2e3f4a5b

{"item_id":"sku_771","qty":2,"currency":"INR"}
```

2. Line 1 has three things: the method `POST`, the path `/v1/orders`, and the query string `?notify=true`. Then the protocol version.
3. `Host` says which site, because one server address may serve thousands.
4. `Authorization` carries the credential. `Bearer` means "whoever holds this may act", which is exactly as dangerous as it sounds.
5. `Content-Type` says what the body is. `Accept` says what you want back. These are two different things and mixing them up is a common mistake.
6. `Content-Length` is the body size in bytes. `Idempotency-Key` is a unique reference for this attempt, explained in section 36.10.
7. A blank line ends the headers. Everything after it is the body.
8. The response:

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /v1/orders/ord_5512
RateLimit-Remaining: 98
Cache-Control: no-store

{"id":"ord_5512","status":"pending","total":1798}
```

9. `201 Created` says a new thing exists. `Location` says where it now lives.
10. `RateLimit-Remaining` says how many more calls you may make in this window.
11. `Cache-Control: no-store` forbids anyone in the middle from keeping a copy.

36.6.4 PLAIN – what is really happening inside

1. Two properties decide how a method may be treated, and they are not the same property.
2. **Safe** means the call is not supposed to change anything. It is a read.
3. **Idempotent** means doing it many times leaves the same state as doing it once. Safe methods are automatically idempotent.
4. `GET`, `HEAD` and `OPTIONS` are safe. `PUT` and `DELETE` are idempotent but not safe. `POST` and `PATCH` are neither.
5. Why this matters in practice: browsers, proxies and client libraries are allowed to retry safe and idempotent requests automatically.
6. So `DELETE /orders/5512` sent twice must leave the order deleted, not produce an error the second time. Many APIs get this wrong.
7. And `POST /orders` sent twice creates two orders. That is correct behaviour and it is exactly why idempotency keys exist.

Method	Safe	Idempotent	Body
GET	Yes	Yes	No
HEAD	Yes	Yes	No
OPTIONS	Yes	Yes	No
POST	No	No	Yes
PUT	No	Yes	Yes
PATCH	No	No	Yes
DELETE	No	Yes	Rare

36.6.5 TECHNICAL – the engineer's version

1. HTTP semantics are now specified in RFC 9110, published June 2022, which replaced the RFC 7230 to 7235 series from 2014, which in turn replaced RFC 2616 from June 1999. Quote RFC 9110 for semantics and RFC 9112 for HTTP/1.1 message syntax.

2. JSON is specified in RFC 8259, December 2017, with media type `application/json`. It has no comment syntax, no trailing commas, and its numbers are IEEE 754 doubles in practice, so integers above 2^53 lose precision. Send large identifiers as strings.
3. The status codes worth knowing, and when to use each:

Code	Meaning	Use when
200	OK	Read or update succeeded
201	Created	New resource made
202	Accepted	Queued, not done yet
204	No Content	Success, nothing to send
301	Moved Permanently	URL changed forever
304	Not Modified	Client's cache is valid
400	Bad Request	Malformed or invalid input
401	Unauthorized	No or bad credential
403	Forbidden	Valid identity, not allowed
404	Not Found	No such resource
405	Method Not Allowed	Wrong verb for that URL
409	Conflict	State clash, version clash
410	Gone	Deleted on purpose
415	Unsupported Media Type	Wrong Content-Type sent
422	Unprocessable Content	Syntax fine, semantics bad
429	Too Many Requests	Rate limit hit
500	Internal Server Error	Server bug
502	Bad Gateway	Upstream returned garbage
503	Service Unavailable	Overloaded or in maintenance
504	Gateway Timeout	Upstream did not answer

4. The 401 versus 403 distinction is worth memorizing, because it is wrong in most codebases. 401 means "I do not know who you are, authenticate"; the header name `WWW-Authenticate` is required with it. 403 means "I know exactly who you are and you still may not".
5. 429 comes from RFC 6585, April 2012, which also defined 428, 431 and 511. Pair it with `Retry-After`, which takes either seconds or an HTTP date.
6. 4xx means the client should change something before retrying. 5xx means the client may retry the identical request later. That single rule drives most retry logic.
7. Content negotiation uses `Accept`, `Accept-Encoding` and `Accept-Language` in the request against `Content-Type`, `Content-Encoding` and `Content-Language` in the response, with `Vary` telling caches which of them affected the answer. Omitting `Vary` is a classic cache poisoning bug.
8. Conditional requests: the server sends `ETag`, the client sends `If-None-Match` on the next read and gets 304 with no body, or sends `If-Match` on a write and gets 412 if someone else changed it first. That second pattern is optimistic concurrency control and costs almost nothing.
9. HTTP/2, RFC 9113, multiplexes many requests over one TCP connection and compresses headers with HPACK. HTTP/3, RFC 9114, runs over QUIC on UDP, removing head-of-line blocking at the transport layer. Both change performance dramatically and change the semantics above not at all.

36.6.6 WORDS - remember these

1. Method — the verb saying what kind of action — the HTTP request method defining semantics, safety and idempotency per RFC 9110.
2. Status code — the three-digit result — 1xx informational, 2xx success, 3xx redirection, 4xx client error, 5xx server error.

3. Header — a small labelled fact attached to a message — a field name and value carrying metadata separate from the body.
4. Safe method — a read that changes nothing — a method whose semantics are defined as read-only, therefore automatically retryable.
5. ETag — a short label for a version of a thing — an opaque validator used with `If-None-Match` and `If-Match` for caching and concurrency.
6. Content negotiation — asking for the format you can read — matching `Accept` family request fields against server representations, signalled by `Vary`.

36.7 REST, properly

36.7.1 PLAIN - in simple words

1. REST is the most misused word in this entire subject.
2. It was defined in 2000 by Roy Fielding in his doctoral thesis. He was describing why the web itself works so well at enormous scale.
3. He did not describe a technology. He described a set of six rules that an architecture may follow, and what each rule buys you.
4. The rules are: separate client from server; keep no client state on the server between calls; say clearly what may be cached; allow layers in between; use one uniform way of addressing and acting on things; and, optionally, allow the server to send code the client runs.
5. Almost nothing sold as REST today follows all six.
6. Most of what people call REST is really "JSON sent over HTTP with nouns in the URL". That is a useful style. It is not what the thesis specified.
7. This matters less than purists claim and more than beginners think. It matters because two of the six rules, statelessness and cacheability, are the reason the web scales, and people abandon them without realizing.

36.7.2 PLAIN - a picture in your head

1. Think of a public library with open shelves.
2. Every book has a fixed shelf mark. You can hand that mark to anyone and they find the same book. That is addressability.
3. The librarian does not remember you between visits. You bring your card each time. That is statelessness, and it means any librarian can serve you.
4. Some books are marked "reference, always current" and some "loan copy, fine for a month". That is cacheability.
5. And each book ends with a list of related shelf marks, so you can keep going without asking anyone. That last one is the rule everybody skips.

Where this comparison breaks:

6. A library has one building. A REST system may have caches and proxies that answer you without the library ever knowing you asked.

36.7.3 PLAIN - a worked example

1. Resource-oriented design in one page. The nouns get URLs; the verbs come from HTTP.

```
GET    /v1/orders                list orders
POST   /v1/orders                create one
GET    /v1/orders/ord_5512       read one
PATCH  /v1/orders/ord_5512       change some fields
DELETE /v1/orders/ord_5512       remove it
GET    /v1/orders/ord_5512/items sub-collection
```

2. Plural nouns, no verbs in the path, lower case, hyphens not underscores. All of that is **convention**, not standard. Nothing enforces it.
3. Filtering, sorting and paging go in the query string, never in the path: `GET /v1/orders?status=paid&sort=-created_at&limit=50`.
4. Now paging, which is where designs break at scale. Two options.
5. Offset paging: `?limit=50&offset=1000`. Simple, and lets you jump to page 21.
6. Cursor paging: `?limit=50&after=eyJpZCI6Nzcx...`. The cursor is an opaque marker for "the last row you saw".
7. Cursor wins at scale for two reasons. First, a database must count and skip 1,000 rows to serve offset 1,000, so page 400 is far slower than page 1.
8. Second, if a new row is inserted while you page, offset paging shows you one row twice and skips another. Cursor paging does not.
9. Partial responses let the client ask for less: `?fields=id,status,total`. That is a convention borrowed from GraphQL's core idea.

36.7.4 PLAIN - what is really happening inside

1. The rule that carries the most weight is statelessness.
2. Stateless means every request carries everything needed to understand it. The server keeps nothing about you between calls.
3. The payoff is that any server can answer any request. So you can put ten servers behind one address, and add an eleventh at lunchtime.
4. The moment you store a session in one server's memory, that request must come back to that server, and your ten servers become ten fragile pets.
5. The cost of statelessness is that every request repeats its credentials and context, so requests are bigger. That trade was made deliberately.
6. The rule everyone skips is called **HATEOAS**: hypermedia as the engine of application state. It means the response tells you what you can do next, by including links, so the client never hardcodes URLs.
7. The web browser does this perfectly. You do not type URLs; you follow links a server gave you. Almost no JSON API does it.

The honest version: an API without HATEOAS is not REST as Fielding defined it.

8. Fielding said so directly in a 2008 post on his own site titled "REST APIs must be hypertext-driven". The industry read it, agreed, and carried on.
9. Practical position: use the six constraints as a checklist of trade-offs, not a purity test. Know which ones you are breaking and why.

36.7.5 TECHNICAL - the engineer's version

1. The six constraints from Chapter 5 of Fielding's 2000 dissertation, with what each one actually buys:

Constraint	What it buys
Client-server	Independent evolution
Stateless	Horizontal scaling
Cacheable	Fewer round trips
Layered system	Proxies, CDNs, gateways
Uniform interface	Generic tooling
Code on demand	Optional, rarely used

2. Code on demand is the only optional constraint. JavaScript delivered to a browser is the canonical example, and it is why the web can extend clients without redeploying them.

3. The uniform interface itself decomposes into four sub-constraints: identification of resources, manipulation through representations, self-descriptive messages, and hypermedia as the engine of application state.
4. A **resource** in Fielding's terms is a concept, not a row and not a file. A **representation** is one rendering of it at one moment, in one media type. The URL identifies the resource; the response body is only a representation.
5. Richardson's Maturity Model, presented by Leonard Richardson in 2008, classifies real APIs: level 0 is one URL and one verb, level 1 adds resources, level 2 adds HTTP verbs and status codes, level 3 adds hypermedia. Most commercial APIs sit at level 2. This is a **convention** for describing maturity, not a standard.
6. Hypermedia formats that do exist: HAL, JSON:API version 1.1, Siren, and `application/problem+json` from RFC 9457, July 2023, which replaced RFC 7807 from March 2016 for error bodies.
7. Cursor paging in production: GitHub uses `Link` headers with `rel="next"`, an application of RFC 8288 web linking. Stripe uses `starting_after` with an object identifier. Both are cursor schemes; neither exposes an offset.
8. Cost of offset paging, measured: a `LIMIT 50 OFFSET 1000000` on PostgreSQL must scan and discard a million rows, taking seconds where a keyset query using `WHERE id > ? ORDER BY id LIMIT 50` on an indexed column takes sub-millisecond time regardless of depth.
9. Statelessness has a cost you can measure: repeating a 900-byte JWT on every request costs 900 bytes uplink per call. HTTP/2 HPACK header compression removes most of that repetition on a persistent connection.

36.7.6 WORDS - remember these

1. REST — a style of network architecture with six rules — Representational State Transfer, defined in Chapter 5 of Fielding's 2000 dissertation.
2. Stateless — the server remembers nothing about you between calls — each request carries all context needed for the server to understand it.
3. Resource — the thing a URL names — a concept with an identifier, distinct from any single representation of it.
4. Representation — one rendering of a thing at one moment — the bytes plus media type returned for a resource.
5. HATEOAS — the answer tells you what you can do next — hypermedia as the engine of application state, the constraint almost nobody implements.
6. Cursor paging — "carry on from this row" — keyset pagination using an opaque marker, stable under insertion and constant-time at any depth.

36.8 The alternatives, compared honestly

36.8.1 PLAIN - in simple words

1. REST-style JSON over HTTP is the default, not the only choice. Four alternatives matter.
2. **GraphQL** gives one endpoint and lets the client write a query saying exactly which fields it wants. The server returns exactly that shape.
3. **gRPC** sends compact binary messages defined in advance by a schema file. It is fast and strictly typed, and it can stream in both directions.
4. **SOAP** is the older XML approach. It is verbose and heavy, and it is still running in banking, insurance, telecoms and government.
5. **JSON-RPC** is the minimal option: one endpoint, a method name, parameters, an identifier. About two pages of specification in total.
6. And for pushing data to a client, **WebSockets** give a two-way permanent connection, while **server-sent events** give a simpler one-way stream from server to client.

7. None of these replaces the others. Each solves a specific pain.

36.8.2 PLAIN – a picture in your head

1. Ordering food. REST is a menu with fixed dishes; you order dish 12 and get what dish 12 contains, including the garnish you did not want.
2. GraphQL is telling the kitchen your exact ingredients, and getting a plate with only those. Wonderful for you, much harder for the kitchen to plan.
3. gRPC is a factory canteen with pre-printed cards and numbered slots. Fast, rigid, and useless to a visitor who does not have the card printer.
4. WebSockets are leaving the intercom switched on both ways all evening.

Where this comparison breaks:

5. A kitchen can refuse a silly order. A GraphQL server can be handed a query that legally asks for a million rows through four levels of nesting, and will try, unless somebody wrote a cost limit.

36.8.3 PLAIN – a worked example

1. The same request, three ways. First GraphQL:

```
POST /graphql
{"query":"{ user(id:42){ name photos(first:2){ title } } }"}

{"data":{"user":{"name":"Asha","photos":[
  {"title":"Pune, monsoon"},{"title":"Bhor ghat"}]}}}
```

2. One call, one round trip, exactly the fields asked for. With REST this often needs two calls: one for the user, one for the photos.
3. That is the **over-fetching** and **under-fetching** problem GraphQL solves, and on a mobile connection it is worth real seconds.
4. Now the same thing as a gRPC schema, written in Protocol Buffers:

```
message PhotoRequest { int64 user_id = 1; int32 first = 2; }
message Photo { string title = 1; int64 taken_at = 2; }
service Photos {
  rpc List(PhotoRequest) returns (stream Photo);
}
```

5. The numbers 1 and 2 are field tags. They go on the wire instead of the field names, which is most of why the encoding is small.
6. `stream Photo` means the server can send photos one at a time as it finds them, rather than making the client wait for all of them.
7. And JSON-RPC, for contrast, is this and nothing more:

```
{"jsonrpc":"2.0","method":"listPhotos",
 "params":{"user":42},"id":7}
```

36.8.4 PLAIN – what is really happening inside

1. Every one of these has a cost, and the honest version of each is the part the marketing leaves out.
2. GraphQL solves over-fetching and creates two new problems.
3. Problem one is caching. HTTP caching works on URLs and methods. GraphQL sends everything as a `POST` to one URL, so every proxy, CDN and browser cache in the world becomes useless to it.
4. Problem two is the **N+1 query problem**. Asking for 100 users and each user's photos naively becomes 1 query for users plus 100 for photos. The standard fix is a batching loader that gathers the 100 into one query, but you must build it. The API shape hides the database cost from the caller.

5. gRPC is fast and typed, and it is rare in browsers, for a concrete reason: it needs control over HTTP/2 frames and trailers that browser JavaScript cannot access. The workaround, gRPC-Web, needs a proxy to translate, so it is extra machinery.
6. SOAP is heavy but it brought things REST had to reinvent: a machine-readable contract in WSDL, and standards for signing and encrypting individual message parts rather than the whole channel.
7. WebSockets give you a persistent two-way channel, and in exchange you now own reconnection, heartbeats, backpressure and message ordering yourself.
8. Server-sent events do far less and cost far less: it is plain HTTP, it reconnects automatically, and the browser gives you an event identifier so you can resume. It only goes one way, which is what most apps need.

36.8.5 TECHNICAL - the engineer's version

1. Comparison on the axes that decide real choices:

Aspect	REST/JSON	GraphQL	gRPC
Payload	Text, medium	Text, tuned	Binary, small
Browser	Native	Native	Needs proxy
Typing	Optional	Schema, strong	Schema, strong
Streaming	SSE or WS	Subscriptions	Native, 2-way
HTTP caching	Excellent	Poor	Not applicable
Tooling	Universal	Good, newer	Good, generated

2. Encoding sizes, illustrative, for one small record of five fields: SOAP envelope roughly 1,200 bytes, JSON roughly 150 bytes, Protocol Buffers roughly 40 to 60 bytes. Exact figures depend entirely on field names and values, so measure your own payloads rather than quoting these.
3. Protocol Buffers achieve that by sending field tag numbers instead of names, using variable-length integers, and omitting fields at default values. The cost is that the message is unreadable without the schema.
4. gRPC supports four call shapes: unary, server streaming, client streaming and bidirectional streaming. It runs over HTTP/2, uses trailers for status, and carries a status code set of its own, where `OK` is 0 and `DEADLINE_EXCEEDED` is 4.
5. GraphQL's specification is maintained by the GraphQL Foundation, hosted by the Linux Foundation since 2018. Practical defences you must add yourself: query depth limits, query cost analysis, persisted queries, and a batching loader such as DataLoader.
6. SOAP lives on where contracts and message-level security matter: SWIFT and ISO 20022 messaging in banking, many telecom provisioning systems, and government tax and customs filing. It is not a legacy curiosity; it is running production traffic in 2026.
7. JSON-RPC 2.0, specified in 2010, is used by Ethereum nodes, the Language Server Protocol behind editor tooling, and the Model Context Protocol used for tool calling by AI systems. Small specifications survive.
8. Push mechanisms: WebSockets are RFC 6455, December 2011, upgrading an HTTP connection to a two-way frame protocol. Server-sent events are part of the HTML living standard, media type `text/event-stream`, one way, with automatic reconnection and a `Last-Event-ID` header for resumption. Long polling still exists and still works where nothing else gets through.
9. Selection rule, stated plainly. Public API for unknown clients: REST. Many different client shapes over one graph of data: GraphQL. Internal service-to-service at high volume: gRPC. Live updates to a browser, one way: server-sent events. Live two-way, such as a game or a chat: WebSockets.

36.8.6 WORDS - remember these

1. Over-fetching — getting more fields than you needed — a fixed representation returning data the client discards, wasting bandwidth and battery.
2. N+1 problem — one query becomes a hundred — resolving a list then issuing one dependent query per element instead of one batched query.
3. Protocol Buffers — a compact binary format with a schema file — Google's IDL and wire format using numbered field tags and varint encoding.
4. Streaming — sending results as they appear — a call that yields many messages over one open call instead of one response.
5. WebSocket — a permanent two-way channel — RFC 6455 framing over an upgraded HTTP connection, full duplex, no request-response structure.
6. Server-sent events — a one-way live feed over plain HTTP — the `text/event-stream` media type with automatic client reconnection.

36.9 Authentication and authorization for APIs

36.9.1 PLAIN - in simple words

1. Two different questions, and confusing them causes real breaches.
2. **Authentication** asks: who are you? **Authorization** asks: are you allowed to do this particular thing?
3. An **API key** is a long random string you send with each request. It answers neither question well. It identifies the caller's account and nothing else.
4. **Basic auth** sends a username and password on every request, lightly encoded, not encrypted. Only safe inside TLS, and even then it is blunt.
5. A **bearer token** is a string that means "whoever holds this may act". It is the standard today. It usually expires, which is its main improvement.
6. **OAuth 2.0** is the system that lets you grant one app limited access to your account on another service, without giving it your password.
7. A **scope** is a label on a token limiting what it may do. Not who you are. What this token may do.
8. A **JWT** is a token with readable information inside it and a signature proving nobody edited it.
9. **mTLS** is where both sides present certificates, so the connection itself proves identity before any request is sent.

36.9.2 PLAIN - a picture in your head

1. Think of a hotel.
2. Your passport at the desk is authentication. It proves who you are, once.
3. The room card you get is a bearer token. It expires on checkout day, and anyone who finds it can open your room. It does not know your name.
4. The card opens room 402 and the gym, but not the wine cellar. Those are scopes.
5. Handing your passport to a tour company so they can collect your parcel is the bad old way. OAuth is instead telling the desk "issue a card that opens only the parcel locker, valid for one hour".
6. And mTLS is the hotel checking your face against a photo while you check the hotel's licence on the wall. Both sides prove identity.

Where this comparison breaks:

7. A hotel card is physical, so only one person can hold it. A leaked token can be copied a million times in a second, and the copies are indistinguishable.

36.9.3 PLAIN - a worked example

1. The OAuth 2.0 authorization code flow with PKCE, step by step. This is the flow you should use for web apps, mobile apps and command-line tools alike.

```
user      app                 auth server        API
 |         |                       |              |
 |         |-1 make verifier+chal->|              |
 |<-2 open browser to /authorize---|              |
 |-3 log in and approve ---------->|              |
 |         |<-4 redirect with code-|              |
 |         |-5 POST code+verifier->|              |
 |         |<-6 access + refresh --|              |
 |         |-7 Bearer access token --------------->|
 |         |<-8 data or 403 if scope missing ------|
```

2. Step 1. The app invents a random secret called the **code verifier**, then hashes it to make a **code challenge**. Real values, computed with SHA-256:

```
verifier : AAECAwQFBgcICQoLDA0ODxAREhMUFRYXGBkaGxwdHh8
challenge: 6oZqdX5MOLq_qBJ8vppAnT4fk6AP8UiP9zX8-Rev_9A
```

3. Step 2. The app opens the user's browser at the authorization server, sending its client identifier, the redirect address, the scopes it wants, a random `state` value, and the challenge. Never the verifier.
4. Step 3. The user logs in at the authorization server, not at the app. The app never sees the password. That is the entire point of OAuth.
5. Step 4. The server redirects back to the app with a short-lived **authorization code**, plus the same `state` so the app can detect forgery.
6. Step 5. The app sends the code and the original verifier to the token endpoint, over a direct connection, not through the browser.
7. Step 6. The server hashes the verifier, compares it with the challenge from step 1, and only then issues an **access token** and a **refresh token**.
8. Why PKCE matters: if an attacker steals the code in step 4, they still cannot use it, because they do not have the verifier. Without PKCE, a stolen code is a stolen account.
9. Step 7 and 8. The app calls the API with `Authorization: Bearer ...`. If the token lacks the needed scope, the answer is 403, not 401.

36.9.4 PLAIN - what is really happening inside

1. A JWT is three base64url-encoded parts joined by dots: header, payload, signature. Here is a real one, signed with HMAC-SHA256 and the secret `kedbyte-demo-secret`. The middle part is one line, wrapped here to fit the page.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
.
eyJpc3MiOiJrZWRieXRlIiwic3ViIjoidV80MiIsInNjb3BlIjoi
cmVwbyIsImlhdCI6MTc2NzIyNTYwMCwiZXhwIjoxNzY3MjI5MjAwfQ
.
mgYCqjUldLiGQs_GeAH1CvayV7pxsCKFLswilq9iX8o
```

2. Decoding part one gives `{"alg":"HS256","typ":"JWT"}`. `alg` is the signing algorithm. `typ` says this is a JWT.
3. Decoding part two gives the claims:

Claim	Value	Meaning
iss	kedbyte	Who issued it

Claim	Value	Meaning
sub	u_42	Who it is about
scope	repo	What it may do
iat	1767225600	Issued 1 Jan 2026 00:00 UTC
exp	1767229200	Expires one hour later

4. Part three is the signature: HMAC-SHA256 over the text `header.payload`, using the shared secret, base64url encoded.
5. Change one character anywhere in parts one or two and the signature no longer matches. That is the whole security model.
6. Now the critical point, and it catches people. Base64 is **not** encryption. Anyone holding the token can read every claim. Put nothing secret in a JWT payload.
7. The signature proves the token was not altered and came from someone with the key. It does not hide anything.
8. And what a scope actually restricts: it restricts the **token**, not the user. The user may have every permission in the world. If the token was issued with scope `read:user` and you attempt a write, the write is refused.
9. The reader's own session showed exactly this. A `git push` was refused because the personal access token lacked the `workflow` scope, and the push modified a CI workflow file. The account had permission. The token did not. Chapter 42 walks through that exact refusal end to end.

36.9.5 TECHNICAL – the engineer's version

1. Specifications, with dates: OAuth 2.0 is RFC 6749, October 2012. Bearer token usage is RFC 6750, October 2012. PKCE is RFC 7636, September 2015. JWT is RFC 7519, May 2015. JWS signatures are RFC 7515. Token introspection is RFC 7662. Token revocation is RFC 7009.
2. Current guidance is RFC 9700, Best Current Practice for OAuth 2.0 Security, published January 2025. It requires PKCE for all client types, requires exact redirect URI matching, and rules out the implicit grant and the resource owner password credentials grant.
3. That is a change with a date, so state it as one: advice that was correct in 2015, such as "use the implicit flow for single-page apps", is now explicitly wrong.
4. Credential mechanisms compared:

Mechanism	Expires	Scoped	Main weakness
API key	Rarely	Sometimes	Leaks live forever
Basic auth	No	No	Password on every call
Bearer token	Yes	Yes	Theft equals full use
JWT	Yes	Yes	Hard to revoke early
mTLS	Cert lifetime	Via cert fields	Certificate management

5. Why API keys are weak, precisely: they are long-lived, usually coarse-grained, frequently committed to source control, and often sent in query strings where they land in server access logs and browser history. GitHub's secret scanning finds millions of leaked credentials a year.
6. JWT signature algorithms: HS256 uses one shared secret, so every verifier can also mint tokens. RS256 and ES256 use a private key to sign and a public key to verify, so verifiers cannot mint. Prefer asymmetric for anything crossing a trust boundary.
7. The classic JWT vulnerability is accepting `alg: none`, or accepting an attacker-chosen algorithm. Always pin the expected algorithm in the verifier rather than reading it from the token you are verifying.

8. Access tokens should be short-lived, commonly 5 to 60 minutes. Refresh tokens are long-lived and must be stored securely; RFC 9700 requires that refresh tokens for public clients be sender-constrained or rotated on each use, so that a stolen one is detectable when both copies are presented.
9. mTLS, mutual TLS, means the client presents an X.509 certificate during the handshake. It is standard inside service meshes such as Istio and Linkerd, where certificates are issued automatically and rotated hourly. RFC 8705 defines binding OAuth tokens to a client certificate, which stops a stolen token being used from another machine.
10. Observation: `curl -v` shows the `Authorization` header being sent, `jwt` command line tools or any base64 decoder reveal claims, and the token introspection endpoint of RFC 7662 tells you server-side whether a token is still live.

36.9.6 WORDS - remember these

1. Authentication — proving who you are — verifying an identity claim against a credential the verifier trusts.
2. Authorization — deciding what you may do — evaluating a policy against an authenticated identity and a requested operation.
3. Bearer token — whoever holds it may act — a credential requiring no proof of possession beyond presentation, per RFC 6750.
4. Scope — a limit written into the token — a named restriction on the operations a specific token may perform, independent of the user's rights.
5. JWT — a signed, readable token — JSON Web Token, RFC 7519: base64url header, claims and signature, integrity-protected but not encrypted.
6. PKCE — proof that the app finishing the flow is the app that started it — RFC 7636 code verifier and S256 challenge, now required for all clients.
7. Refresh token — the long-lived key used to get new short-lived ones — a credential exchanged at the token endpoint, rotated or sender-constrained.
8. mTLS — both sides show certificates — mutual TLS authentication binding identity to the transport connection itself.

36.10 Making an API reliable in the real world

36.10.1 PLAIN - in simple words

1. An API that works when everything is calm is easy. An API that works when things go wrong is the whole job.
2. Six habits do almost all of the work. Learn these six and you are ahead of most working code.
3. **Rate limiting** is the server saying "you may only ask me this many times in this much time". It protects the server from being flattened.
4. **Retrying** is the client saying "that failed, let me try again". Done badly, it turns a small problem into an outage.
5. **Backoff** means waiting longer before each retry. **Jitter** means adding a random amount to that wait so that everyone does not retry together.
6. An **idempotency key** is a unique reference you attach to a request, so the server can recognize a repeat and refuse to do the work twice.
7. A **timeout** is a deadline. Without one, a slow answer becomes a stuck program, and stuck programs pile up until the machine dies.
8. A **circuit breaker** stops calling a service that is clearly broken, so you fail fast instead of queueing up thousands of doomed calls.
9. **Graceful degradation** means giving a worse but useful answer instead of no answer: a slightly old price, a shorter list, a page without the map.

10. Now the one rule to carry out of this section, stated plainly. A client that retries a request which is not idempotent, without an idempotency key, can charge a real person twice for the same purchase.
11. That is not a rare edge case. It is the normal outcome of a lost response plus an automatic retry, and it happens on ordinary evenings.

36.10.2 PLAIN - a picture in your head

1. Picture two buckets, side by side. They are the two standard ways to limit a rate, and they behave differently in a way that matters.
2. The first bucket holds tokens. A tap drips a new token in at a steady speed. The bucket has a maximum size and overflows if it fills.
3. To make a request you must take a token out. No token, no request.
4. If you have been quiet for a while, the bucket is full, and you may fire off a whole burst at once. That is the **token bucket**.
5. The second bucket has a small hole in the bottom. You pour requests in at the top. They drip out of the hole at one fixed speed, no matter how fast you pour.
6. If you pour too fast the bucket fills up and the rest spills on the floor. That is the **leaky bucket**.
7. So the token bucket forgives a burst after a quiet period. The leaky bucket never forgives a burst; it smooths everything into one steady stream.
8. Now the circuit breaker, which is a different picture. It is the fuse in the fuse box of a house.
9. When a wire keeps short-circuiting, the fuse blows and cuts the power. It does not keep pushing current into a fault until the house catches fire.
10. Later, somebody carefully switches it back on to see whether the fault has been fixed. If it fails again, it trips again straight away.

Where this comparison breaks:

11. Real buckets and taps are continuous. Software counts in discrete steps and checks the clock, so a token bucket is arithmetic, not plumbing.
12. A house fuse is reset by a person. A circuit breaker in software resets itself on a timer, which means it can flip back and forth if you choose the timer badly.
13. Water in a bucket is anonymous. Rate limits are almost always counted per caller, so there are millions of tiny buckets, not one big one.

36.10.3 PLAIN - a worked example

1. First, the maths of a herd, because this is the part people do not believe until they see the numbers.
2. Set the scene. Ten thousand clients are talking to one service. The service restarts and is unavailable for 30 seconds.
3. Every client fails at roughly the same instant, because they all failed for the same reason: the service went away.
4. Each client uses plain exponential backoff with a base of one second: wait 1, then 2, then 4, then 8, then 16, then 32 seconds.
5. Because they all started together, they all wake together. Here is when each retry wave lands, measured from the moment of failure.

Attempt	No jitter, all arrive at	Full jitter, spread over
1	t = 1 s	0 to 1 s
2	t = 3 s	0 to 2 s
3	t = 7 s	0 to 4 s
4	t = 15 s	0 to 8 s
5	t = 31 s	0 to 16 s

Attempt	No jitter, all arrive at	Full jitter, spread over
6	t = 63 s	0 to 32 s

6. Without jitter, all ten thousand requests arrive inside the same few milliseconds. Suppose the service can handle 500 requests per second.
7. Ten thousand requests in, say, 10 milliseconds is an instantaneous rate of one million requests per second. That is 2,000 times capacity.
8. So the wave fails. And because it fails together, the next wave is still synchronized. The herd never breaks up. This is the **thundering herd**.
9. Worse: the service comes back at t = 30 s, gets hit by the whole herd at t = 31 s, falls over again, and the outage extends itself.
10. Now add jitter. The rule called full jitter is: wait a random amount between zero and the current backoff window.
11. At attempt 6 the window is 32 seconds wide. Ten thousand clients spread evenly over 32 seconds is 10,000 divided by 32, which is about 312 requests per second.
12. 312 is comfortably under the 500 the service can take. The herd has become a queue, and the service recovers.
13. That is the entire argument for jitter, in one division. Randomness turns a spike into a flow, and it costs one call to a random number generator.
14. Second worked example: the double charge, step by step, with real money.

```
t=0.0s  client POSTs /charges  amount 4999  (49.99 in minor units)
t=0.4s  server creates charge ch_A, moves 4999, starts reply
t=0.4s  the reply is lost somewhere on the path back
t=5.0s  client timeout fires. Client sees: "no response"
t=5.0s  client retry logic: "safe to retry, it probably failed"
t=5.4s  server creates charge ch_B, moves 4999 AGAIN
        customer has now paid 99.98 for one order
```

15. Nothing here is a bug in the usual sense. Every component did what it was told. The client could not tell a lost request from a lost response.
16. Now the same story with an idempotency key. The client generates a random identifier once, before the first attempt, and reuses it on every retry.

```
t=0.0s  POST /charges  Idempotency-Key: 8f14e45f-ea4f-4c1b-9b4a-0c1d
t=0.4s  server records key 8f14... -> charge ch_A, result 201
t=0.4s  reply lost
t=5.0s  client retries with the SAME key 8f14...
t=5.4s  server finds the key already recorded
t=5.4s  server replays the stored result. No second charge.
t=5.4s  client receives 201 and the SAME charge id ch_A
```

17. Read the last line again. The client gets the original answer, not a new one, and not an error. From the client's view the call simply succeeded.
18. The key must be generated once per logical operation, not once per attempt. A new key on each retry recreates the original bug exactly.

36.10.4 PLAIN - what is really happening inside

1. A token bucket in code is three numbers per caller: how many tokens are left, the refill rate, and the time it was last touched.
2. Nothing runs on a timer. When a request arrives, the server works out how much time has passed and adds that many tokens, capped at the bucket size.

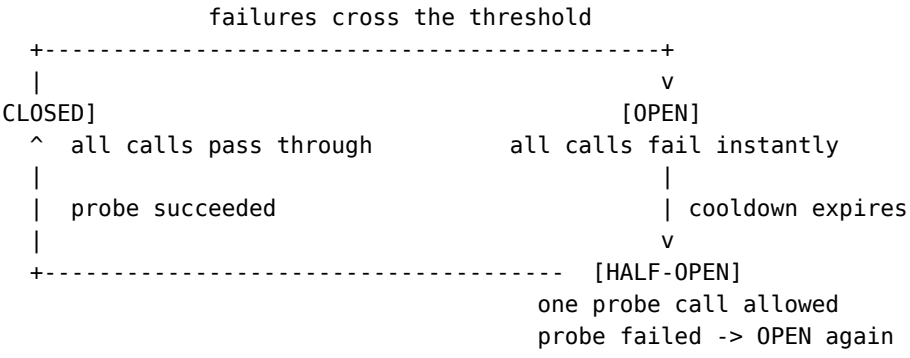
3. Worked with numbers. Bucket size 100 tokens, refill 10 tokens per second. The caller has been quiet for a minute, so the bucket is full at 100.

Event	Tokens before	Tokens after
Burst of 100 requests	100	0
Request 101, same instant	0	refused, 429
100 ms later, 1 request	1	0
Steady state	tops up 10/s	allows 10/s

4. So the shape is: a burst of up to 100, then a sustained 10 per second forever. That is usually what you want from a public API.
5. A leaky bucket used as a queue is different. Requests go into a queue of depth 100 and are released at exactly 10 per second.
6. Send 100 at once and none are refused, but the hundredth is not served for ten seconds. The burst is paid for in waiting instead of in refusals.
7. Compare the two directly:

Property	Token bucket	Leaky bucket
Bursts	Allowed up to depth	Smoothed away
Output shape	Bursty, then steady	Constant rate
Excess traffic	Refused immediately	Queued, then dropped
Adds delay	No	Yes, up to depth/rate

8. Which to choose: token bucket when callers are humans or apps that come and go, leaky bucket when you must protect something downstream that cannot take a spike at all, such as a payment processor or a legacy database.
9. When the limit is hit, the server answers 429 Too Many Requests, and should include a `Retry-After` header saying how long to wait.
10. `Retry-After` may be a number of seconds, such as `Retry-After: 30`, or an HTTP date. A client that ignores it and hammers on is the reason services start banning integrations.
11. Now the circuit breaker. It is a small state machine with three states, kept per downstream service.



12. Closed is normal. Calls go through and failures are counted.
13. Open means the breaker has tripped. Calls are not attempted at all. They fail immediately, locally, in microseconds, with no network involved.
14. That instant local failure is the point. It frees the threads, sockets and memory that would otherwise be stuck waiting on a dead service.
15. Half-open is the careful test. After a cooldown, one call is let through. If it works, close the breaker. If not, open it again and wait longer.

16. Timeouts sit under all of this. There is not one timeout; there is a timeout at every layer, and if any layer has none, the whole chain can hang.
17. And graceful degradation is the plan for when the breaker is open. You always need an answer for the question: what do we show the user now?

36.10.5 TECHNICAL - the engineer's version

1. Status codes and headers, with sources. 429 Too Many Requests comes from RFC 6585, April 2012, and is carried forward in RFC 9110, June 2022. `Retry-After` is specified in RFC 9110 section 10.2.3 and takes either delta-seconds or an HTTP-date.
2. Use 429 for "you exceeded your quota" and 503 Service Unavailable for "I am overloaded or in maintenance". Both may carry `Retry-After`. This distinction is a convention that most large APIs follow, not a rule the specification enforces.
3. Standard rate-limit headers are, as of August 2026, still not an RFC. The IETF HTTPAPI working group document `draft-ietf-httpapi-ratelimit-headers` reached revision 11 in May 2026 and defines two structured fields, `RateLimit` and `RateLimit-Policy`, replacing the earlier triple `RateLimit-Limit`, `RateLimit-Remaining` and `RateLimit-Reset`.
4. In practice, today, most services send their own `X-RateLimit-*` headers. Treat header names as per-service configuration, not as a standard, and read the documentation for each API you call.
5. GitHub's published REST limits, which are a good calibration point:

Caller	Requests per hour
Unauthenticated	60
Personal access token	5,000
GitHub App installation	5,000 to 12,500
Enterprise Cloud app	15,000

6. GitHub returns `x-ratelimit-limit`, `x-ratelimit-remaining`, `x-ratelimit-used`, `x-ratelimit-reset` (UTC epoch seconds) and `x-ratelimit-resource`, and answers with 403 or 429 when you exceed a limit. GitHub also enforces undocumented secondary rate limits on bursts and on expensive operations, which is why blind parallelism gets integrations blocked.
7. Backoff formulas. The reference treatment is Marc Brooker's AWS Architecture Blog post "Exponential Backoff And Jitter", 4 March 2015. Its three variants, exactly as published:

```
no jitter    : sleep = min(cap, base * 2 ** attempt)
full jitter  : sleep = random(0, min(cap, base * 2 ** attempt))
equal jitter : temp  = min(cap, base * 2 ** attempt)
               sleep = temp/2 + random(0, temp/2)
decorrelated : sleep = min(cap, random(base, sleep * 3))
```

8. The post's simulations found full jitter best overall on total work done and competitive on completion time. Use full jitter unless you have measured a reason not to. Typical parameters: base 100 ms, cap 20 s, 5 attempts.
9. Only retry what is retryable. Retry on connection errors, timeouts, 429, 502, 503 and 504. Do not retry 400, 401, 403, 404, 409 or 422; the answer will not change and you are just generating load.
10. Idempotency keys. `Idempotency-Key` is described in the IETF draft `draft-ietf-httpapi-idempotency-key-header`, revision 07, October 2025. It is still a draft, not an RFC, but the header name is a de facto industry convention.
11. Stripe's implementation is the reference behaviour, and its documented rules are worth copying exactly:

Rule	Stripe's behaviour
Key form	V4 UUID suggested, 255 chars max
Storage	Status code and body saved
Retention	Pruned after at least 24 hours
Different params, same key	Request errors

12. Note the fourth row. Reusing a key with different parameters is an error, not a silent overwrite. That catches the bug where a client reuses a key for a different purchase.
13. Stripe accepts idempotency keys on all `POST` requests, and states plainly that they have no effect on `GET` and `DELETE` because those are already idempotent by definition per RFC 9110.
14. Server-side, the usual implementation is a table keyed on (account, idempotency key) with a unique index, holding a state of in-progress or complete plus the serialized response, written in the same database transaction as the effect.
15. Timeouts, layer by layer. Every one of these has a default, and several common defaults are effectively infinite.

Layer	Setting	Sensible value
DNS lookup	resolver timeout	2 to 5 s
TCP connect	connect timeout	2 to 5 s
TLS handshake	handshake timeout	5 s
Response read	read timeout	10 to 30 s
Whole call	overall deadline	30 s

16. With `curl` these are `--connect-timeout` and `--max-time`, plus `--retry`, `--retry-delay`, `--retry-max-time` and `--retry-all-errors`. In Go they are fields on `http.Transport` plus a `context` deadline. In Python's `requests` a bare `timeout=` covers connect and read, and omitting it means wait forever, which is the single most common production mistake in that library.
17. Deadlines should propagate. gRPC does this properly: a deadline set by the caller travels in the `grpc-timeout` header and is reduced at each hop, so a service that is already out of time does not start new work. HTTP has no standard equivalent, which is why hand-rolled chains leak time.
18. Hard platform limits exist and will surprise you. Amazon API Gateway had a fixed 29-second integration timeout for years; in June 2024 AWS allowed that limit to be raised beyond 29 seconds for regional REST APIs, subject to an account-level quota increase. Check the number for your platform rather than assuming.
19. Circuit breakers. The pattern was named and popularized by Michael Nygard in "Release It!", 2007. Netflix's Hystrix was the best-known Java implementation; its last release, 1.5.18, was in November 2018, after which Netflix put it into maintenance mode and pointed new work at resilience4j. Envoy, Istio and Linkerd implement the same idea in the proxy layer, outside your code.
20. Practical breaker settings: trip on a rolling error rate rather than a raw count, for example 50 percent failures over a 10-second window with a minimum of 20 requests; cooldown 5 to 30 seconds; one probe in half-open.
21. Related and often confused: a **bulkhead** limits how many concurrent calls one dependency may consume, so a slow dependency cannot eat the whole thread pool. Breakers stop bad calls; bulkheads contain them.
22. Load shedding is the server-side twin of the breaker. Under overload, reject early and cheaply with 429 or 503 rather than accepting work you cannot finish. Accepting everything and timing out is the worst option because you pay the cost and deliver nothing.

23. Graceful degradation in concrete terms: serve a stale cached value and mark it stale, drop optional page sections, fall back to a smaller model or a simpler ranking, queue writes for later, or return partial results with a flag saying what is missing.
24. HTTP even has a header for the stale case: `Cache-Control: stale-if-error` from RFC 5861, May 2010, lets a cache serve a stale response when the origin errors.
25. Observation and testing: `curl -w` prints timing breakdowns such as `time_connect` and `time_starttransfer`; `ab`, `wrk`, `hey` and `k6` generate load; `tc netem` and Toxiproxy inject latency and loss; and chaos engineering, as practised by Netflix's Chaos Monkey from 2011, tests these paths on purpose rather than by accident.

36.10.6 WORDS - remember these

1. Rate limit — a cap on how often you may ask — a server-enforced quota per caller per window, signalled with 429 and `Retry-After`.
2. Token bucket — a jar of tickets that refills steadily — a limiter holding capacity C refilled at rate R, permitting bursts up to C.
3. Leaky bucket — a bucket with a hole, output always steady — a limiter that queues arrivals and releases them at a fixed rate, smoothing bursts.
4. Exponential backoff — wait longer after each failure — retry delay growing as base times two to the power of the attempt number, capped.
5. Jitter — randomness added to the wait — a randomized delay that de-synchronizes retrying clients and prevents a thundering herd.
6. Thundering herd — everybody retries at the same instant — synchronized client retries producing a load spike far above steady-state capacity.
7. Idempotency key — a reference number on the request — a client-generated unique value letting the server deduplicate retries and replay the result.
8. Circuit breaker — a fuse for a failing dependency — a three-state machine, closed, open and half-open, that fails fast while a service is unhealthy.
9. Bulkhead — a limit on how much one dependency may consume — a bounded concurrency pool per downstream, containing the blast radius of slowness.
10. Graceful degradation — a worse answer beats no answer — deliberately reduced functionality preserving the core path during partial failure.

36.11 Webhooks: the inversion where the server calls you

36.11.1 PLAIN - in simple words

1. Everything so far had the same shape. You call the server. The server answers. You are the one who starts the conversation.
2. A **webhook** turns that around. You give the other service a web address of yours, and when something happens, it calls you.
3. So your program stops being only a client. For webhooks it must also be a server, sitting there with a door open, waiting.
4. The name comes from "web" plus "hook", where a hook is an old programming word for a place you can attach your own code so it runs when an event occurs.
5. The reason webhooks exist is simple: without them you have to keep asking "has it happened yet? has it happened yet?", which is called **polling**.
6. Polling wastes almost all of its calls, and it is always a little late, because you only find out at the next check.
7. There is a price for the inversion. Anyone on the internet can send a POST to your address, so you must prove that a message really came from the sender you expect.

8. You do that with a **signature**: the sender computes a code from the message and a shared secret, and you compute the same code and compare.
9. And because the network can lose things, senders resend. So the same event can arrive twice, or three times, or out of order. Your handler must cope.
10. Finally, the practical wall every beginner hits: a laptop at home has no address the internet can reach, so nothing can call you at all. Section 36.11.4 explains why and section 36.11.5 gives the fixes.

36.11.2 PLAIN - a picture in your head

1. Polling is standing at the window every minute, all day, watching for the postman.
2. You look 480 times during the working day. The post arrives once. You looked 479 times for nothing, and you found out up to a minute late.
3. A webhook is a doorbell. You install it once, you give the wire an address, and you get on with your life. It rings when there is something to know.
4. The signature is the courier saying a password you agreed in advance. Anyone can press a doorbell. Only the real courier knows the word.
5. The password is not simply spoken, though. It is mixed with the parcel itself, so the word only matches if the parcel is unchanged.
6. And the courier is stubborn. If nobody answers, they come back in a minute, then in five, then in an hour. So the same parcel can be delivered twice if you took it and then failed to sign for it.

Where this comparison breaks:

7. A doorbell rings in your house, where you already are. A webhook has to reach a machine that in most cases has no public address at all. That is the difference the whole of section 36.11.4 is about.
8. A courier delivers parcels in the order they were posted, near enough. A webhook sender makes no such promise, and several large ones explicitly say so in their documentation.
9. A doorbell that rings twice is obvious to a human. Two identical webhook deliveries look identical to a program, which is why you need event identifiers.

36.11.3 PLAIN - a worked example

1. Compare the two designs with numbers. A shop gets 5 orders a day and wants to react when a payment succeeds.
2. Polling every 60 seconds: 60 times 24, which is 1,440 requests a day. Five of them find something. 1,435 are wasted, and news is up to 60 seconds old.
3. Webhooks: 5 deliveries a day, each arriving about a second after the event. That is 288 times fewer requests and 60 times fresher.
4. Polling every second to fix the delay makes it 86,400 requests a day for the same 5 pieces of news, and burns your rate limit for nothing.
5. Now the delivery itself. Here is a real signed webhook, in the style Stripe uses. The values below were computed for this book and are genuine.

```
POST /hooks/payments HTTP/1.1
Host: shop.example
Content-Type: application/json
Stripe-Signature: t=1767225600,
 v1=9b76fe28839a1536d4ba22b9641546bd14ad4da938f8fa3cfa3f
    e34d35633eb8
Content-Length: 57

{"id":"evt_9f2","type":"payment.succeeded","amount":4999}
```

6. The signature line is one line in reality; it is wrapped here to fit the page. `t` is the time the sender signed it. `v1` names the scheme.

7. To verify, you build the signed string by joining the timestamp, a full stop, and the exact body bytes:

```
1767225600.{"id":"evt_9f2","type":"payment.succeeded",
"amount":4999}
```

8. Then you compute HMAC-SHA256 over that string using the endpoint secret, here `whsec_kedbyte_demo`, and compare with the `v1` value. They match.
9. Now watch what happens if your web framework parses the JSON and your code re-serializes it before checking. Same data, spaces after the colons:

```
body as sent : {"id":"evt_9f2","type":"payment.succe...
signature    : 9b76fe28839a1536d4ba22b9641546bd14ad4d...

body re-made : {"id": "evt_9f2", "type": "payment.suc...
signature    : bd01c7e20696abf97e590d6b0e86cae26157bf...
```

10. Every character of the signature is different. Nothing about the meaning changed. Four space characters changed the bytes, and HMAC signs bytes.
11. This is the single most common webhook bug in the world, and now you will never write it: verify against the **raw body**, before any parsing.
12. Last piece: the timestamp is inside the signed string. So an attacker who captures a valid delivery and replays it tomorrow cannot change `t` without breaking the signature, and your tolerance check rejects the old one.

36.11.4 PLAIN - what is really happening inside

1. A webhook delivery is nothing exotic. It is an ordinary HTTP POST, made by the sender's servers, to a URL you registered with them earlier.
2. The sequence, end to end:

```
your app      payment service        your receiver
   |                |                      |
   |-1 register URL->|                     |
   |                |                      |
   |             2 event happens           |
   |                |-3 POST + signature->|
   |                |                 4 verify HMAC
   |                |                 5 dedupe on id
   |                |<-6 200 OK, fast ----|
   |                |                 7 do work async
   |                |                      |
   |                |-8 retry if no 200 ->|
```

3. Step 4 must come before anything else. An unverified webhook is an instruction from a stranger, and people have been robbed by acting on one.
4. Step 5 exists because delivery is at-least-once. Every serious sender attaches a unique event identifier, and your job is to remember the ones you have already handled and ignore repeats.
5. Step 6 must be fast. Senders time out. If your handler does ten seconds of database work before replying, you will be marked as failed and resent the same event, which multiplies your load exactly when you are slow.
6. So the pattern is: verify, deduplicate, write the event to a queue or table, return 200, and do the real work afterwards in the background.
7. Ordering. Events are generated in one order and delivered in whatever order the network and the retry queue produce. Stripe states in its documentation that it does not guarantee delivery order.
8. The fix for ordering is not to sort them. It is to stop depending on order: treat each event as a hint that something changed, and read the current state back from the API before acting on anything important.
9. Now the wall. Why can a webhook not reach your laptop?

10. Your machine at home has a private address, something like `192.168.0.14`, handed out by your router. Chapter 24 covered these: the ranges `10.0.0.0/8`, `172.16.0.0/12` and `192.168.0.0/16` from RFC 1918 are not routable on the public internet.
11. Your router performs NAT, network address translation. When you make an outbound connection it writes down a mapping and rewrites the addresses, so replies find their way back to you.
12. The mapping is created by your outbound packet. Nothing creates a mapping for an inbound connection from a stranger. So a POST from a payment company arrives at your router with no idea where to send it, and is dropped.
13. The reader's own network makes this worse in a way worth seeing. Their traceroute showed the ISP's own core using private addresses, `172.31.0.17` and `172.26.22.235` among them, before reaching a public address at `137.97.29.249`.
14. That pattern is carrier-grade NAT: the reader's home router is itself behind another layer of translation run by the ISP. Even opening a port on the home router would not help, because the public address is not theirs to control.
15. So the answer is not to accept an inbound connection at all. The answer is to make an outbound connection from inside, which NAT allows, and let the inbound requests travel back down it. That is a tunnel, and Chapter 34 covered exactly this mechanism for VPNs and proxies.

36.11.5 TECHNICAL - the engineer's version

1. History and standardization. Jeff Lindsay coined the term "web hook" in a blog post on 3 May 2007. There was no specification for twenty years. On 13 December 2023 the Standard Webhooks project published an open specification, with a steering committee drawn from Zapier, Twilio, Lob, Mux, ngrok, Supabase and Kong. It is a convention with momentum, not an RFC.
2. Header conventions in the wild differ, and there is no winner:

Sender	Signature header	Algorithm
GitHub	`X-Hub-Signature-256`	HMAC-SHA256, hex
Stripe	`Stripe-Signature`	HMAC-SHA256, hex
Standard Webhooks	`webhook-signature`	HMAC-SHA256, base64

3. GitHub signs the raw body alone and prefixes the digest with `sha256=`. Stripe signs `timestamp + "." + raw body`. Standard Webhooks signs `id + "." + timestamp + "." + raw body`. Read the sender's documentation; guessing the signed string is the second most common webhook bug.
4. GitHub also sends `X-GitHub-Event` naming the event type, `X-GitHub-Delivery` carrying a GUID that identifies the delivery, and `X-GitHub-Hook-ID` identifying the webhook itself.
5. Verification checklist, in order:
    1. Read the raw request body as bytes. Disable any middleware that parses, re-encodes, decompresses or pretty-prints it first.
    2. Recompute the HMAC with the endpoint secret.
    3. Compare with a constant-time comparison, such as `hmac.compare_digest` in Python or `crypto.timingSafeEqual` in Node, never with ==.
    4. Check the timestamp is recent. Stripe's libraries default to a 5-minute tolerance and warn against setting it to zero, which disables the check.
    5. Only then parse the JSON.
6. Constant-time comparison matters because a naive byte-by-byte comparison returns faster on an early mismatch, and that timing difference is enough to recover a signature one byte at a time over many attempts.
7. Secret rotation: senders usually support two active secrets during a change window. Stripe allows a rolled secret's predecessor to stay valid for up to 24 hours and sends one signature per active secret in the same header.

8. Retry behaviour from the sender's side, with real published policies. Stripe retries deliveries for up to three days in live mode with exponential backoff, and allows manual resends for up to 15 days from the Dashboard and 30 days from the CLI. Other senders differ enormously; some do not retry automatically at all and only offer manual redelivery. Never assume.
9. Because retries exist, deduplicate on the sender's event identifier. Stripe recommends recording processed event IDs, and warns that in some cases two distinct Event objects are generated for the same underlying change, so deduplicating on `data.object` identity plus `event.type` is sometimes needed as well.
10. Idempotent handlers are the real defence. Design the handler so that applying the same event twice leaves the same state. Then duplicates are free rather than dangerous, and you are back to the exactly-once effect described in section 36.5.
11. Return 2xx immediately, then process asynchronously. Stripe treats any 3xx as a failure, so a webhook URL that redirects, for example from a bare domain to `www`, silently fails every delivery.
12. Additional hardening: allowlist the sender's published IP ranges as a second layer; require HTTPS with TLS 1.2 or 1.3; exempt the webhook route from CSRF protection, which otherwise rejects every legitimate delivery; and never trust amounts or ownership from the payload alone for anything financial, since the correct move is to re-read the object from the API.
13. Receiving on a machine behind NAT. The options, honestly compared:

Approach	Works behind CGNAT	Cost of setup
Public server or function	Yes	Deploy something
Router port forward + DDNS	No	Router config
Reverse tunnel service	Yes	One command
Cloud queue relay	Yes	Extra service

14. Port forwarding fails under carrier-grade NAT for the reason Chapter 24 gave: the public address belongs to the ISP and is shared, so there is no port on it that is yours to map.
15. Reverse tunnels work because the connection is established outbound from inside the network, which NAT permits, and inbound requests are then multiplexed back down that already-open connection.
16. The common tools: ngrok, first released by Alan Shreve in 2013 and incorporated in 2015; Cloudflare Tunnel, launched as Argo Tunnel in 2018 and made free for everyone on 15 April 2021 with the `cloudflared` daemon; Tailscale Funnel; and plain `ssh -R` to any host you already own, which needs `GatewayPorts` enabled on that host.
17. Vendor CLIs do the same thing without a public URL at all. `stripe listen --forward-to localhost:4242/webhook` opens an outbound connection to Stripe and replays events to your local port, printing a signing secret for the session. `gh webhook forward` does the equivalent for GitHub.
18. Testing tools: `stripe trigger payment_intent.succeeded` fires a real test event; request-bin style services capture and display raw deliveries; and the sender's own delivery log, which shows the exact request and your exact response, is almost always the fastest way to find the fault.
19. Webhooks versus the alternatives, in one table:

Mechanism	Direction	Best for
Polling	Client asks	Simple, low event rate
Webhook	Server calls you	Rare events, any client
WebSocket	Both, held open	Live two-way, chat
Server-sent events	Server pushes	One-way live feed
Message queue	Broker in middle	High volume, ordering

20. At scale, webhooks become a queue problem, and teams move to a broker such as Amazon EventBridge, Google Pub/Sub or Kafka. Stripe supports sending events directly to EventBridge or Azure Event Grid instead of to an HTTP endpoint, which removes your receiver from the critical path entirely.

36.11.6 WORDS - remember these

1. Webhook — the other service calls you when something happens — a user-registered HTTP callback delivering event notifications by POST.
2. Polling — asking repeatedly whether anything changed — client-initiated periodic requests, costly and bounded in freshness by the interval.
3. HMAC — a code proving a message came from someone with the secret — keyed-hash message authentication code, RFC 2104, usually with SHA-256.
4. Raw body — the exact bytes that arrived — the unparsed request payload, the only correct input to signature verification.
5. Replay attack — resending a captured valid message — defended by signing a timestamp and rejecting deliveries outside a tolerance window.
6. Delivery identifier — the sender's unique name for this event — an event or delivery ID used by the receiver to detect and drop duplicates.
7. Reverse tunnel — an outbound connection used to carry inbound requests — a NAT-traversal technique multiplexing public traffic to a private host.
8. Carrier-grade NAT — the ISP translates too, not just your router — RFC 6598 shared address space, making inbound port mapping impossible for the user.

36.12 Designing an API people can actually use

36.12.1 PLAIN - in simple words

1. A usable API is mostly boring. It is boring in the same way, everywhere, and that sameness is the feature.
2. **Naming.** Use nouns for things and let the method say the verb. A path called `/orders/8812` with a `DELETE` is better than `/deleteOrder?id=8812`.
3. **Consistency.** If one part of your API uses `created_at` and another uses `creationTime`, every user of your API will get it wrong forever.
4. **Errors with codes.** Send back a short machine-readable code such as `insufficient_funds`, alongside the human sentence.
5. That code matters far more than the sentence, and here is why: sentences get reworded, translated and improved, and every rewording silently breaks any program that was matching on the text.
6. A code is part of the contract. A message is not. If you promise nothing about the message, you may fix its typos without breaking anyone.
7. **Versioning.** There are four honest answers to "how do we change things without breaking people", and each costs something. Section 36.12.5 compares them without pretending one is obviously right.
8. **Deprecation.** Say a thing is going away, say when, say what to use instead, and then actually keep to the date.
9. **Documentation.** Reference pages tell you what exists. Guides tell you how to do the common job. You need both, and you need examples that run.
10. **A machine-readable description.** If your API is described in a file a computer can read, then documentation, client libraries, test doubles and input checking can all be generated from that one file.
11. **A sandbox.** Somewhere users can try everything with fake money and fake data, and get predictable answers, without fear.

36.12.2 PLAIN - a picture in your head

1. Think of a large hardware shop with many aisles.
2. A good shop puts screws in one place, and every box is labelled the same way: size, then length, then material, then quantity.
3. Once you have read one label you can read all of them. You never have to learn the shop twice.
4. A bad shop labels some boxes by length first and some by material first, and keeps a few screws in the paint aisle for historical reasons.
5. The stock is identical in both shops. One is pleasant, one is exhausting. That difference is entirely consistency, and it is the same for APIs.
6. Now the error codes. Imagine the shop's till printing "Card refused" on one receipt and "Your card has been declined" on another for the same fault.
7. A person shrugs. A machine that was told to look for the exact words "Card refused" now silently does the wrong thing.
8. Put a number on the receipt as well, always the same number for the same fault, and the machine can be right regardless of the wording.

Where this comparison breaks:

9. A shop can rearrange the aisles overnight and customers adapt in a week. An API cannot, because its customers are programs that do not adapt at all.
10. A shop has one physical layout. An API often has to serve the old layout and the new layout at the same time, for years. That is versioning, and shops have no equivalent of it.

36.12.3 PLAIN - a worked example

1. Here is one endpoint designed badly, then the same endpoint designed well. Nothing about the underlying system changes.

```
BAD
GET /getOrderData?orderid=8812&fmt=json&full=1
200 OK
{"ok":0,"err":"Order not found or you dont have access"}

GOOD
GET /v1/orders/ord_8812
404 Not Found
Content-Type: application/problem+json
{"type":"/errors/not-found","code":"order_not_found", ...}
```

2. Five faults in the bad version, one at a time.
3. The path contains a verb, `getOrderData`, which duplicates what `GET` already says, and it will sit awkwardly beside `updateOrderData` later.
4. The identifier is in a query string, so `/getOrderData` with no arguments is a different thing again, and caches and logs treat them inconsistently.
5. The status is 200, meaning success, for a failure. Every client library, proxy and monitoring dashboard in the world will record this as fine.
6. The error is a sentence with a spelling mistake. Fix the spelling and you break anybody who matched on the string. So the mistake becomes permanent.
7. The message merges two different situations, "no such order" and "not yours", so the caller cannot tell them apart even if it wanted to.
8. Now the good version in full. This is what the body should look like, using the standard problem format:

```
HTTP/1.1 422 Unprocessable Content
Content-Type: application/problem+json
```

```
{
  "type": "/errors/insufficient-funds",
  "title": "Insufficient funds",
  "status": 422,
  "detail": "Balance 1200 is below the amount 4999.",
  "instance": "/orders/ord_8812",
  "code": "insufficient_funds",
  "balance_minor": 1200,
  "required_minor": 4999
}
```

9. `code` is the stable machine-readable name. `title` and `detail` are for humans and may be reworded at any time. `instance` says which thing failed.
10. The extra fields `balance_minor` and `required_minor` let a client build a useful message itself, in its own language, without parsing English.
11. Notice the money. It is an integer in minor units, 4999 meaning 49.99, not a decimal number. Section 36.12.5 explains why floats and money do not mix.
12. Now the same discipline applied to field naming across a whole API:

Decision	Pick one	Never mix
Field case	`snake_case`	with `camelCase`
Timestamps	RFC 3339 UTC	with epoch seconds
Identifiers	prefixed strings	with bare integers
Money	integer minor units	with decimals
Lists	always paginated	with sometimes-all

13. Prefixed identifiers such as `ord_8812` and `cus_41a` are a small idea with a large payoff: a support engineer can tell what a value is by looking at it, and a client cannot accidentally pass a customer where an order goes.

36.12.4 PLAIN - what is really happening inside

1. Versioning exists because of one asymmetry. You can deploy a change in a minute. Your callers may take three years to update, and some never will.
2. So the real question is never "what is the tidiest version scheme". It is "how long am I willing to run the old behaviour, and who pays for it".
3. There are four families of answer, and a fifth position that says do not version at all.
4. Put the version in the path, as `/v1/orders`. Simple, visible, easy to route in a proxy, easy to explain. The cost is that the version is attached to every URL, so identifiers are not stable across versions and a client that upgrades must rewrite every path it holds.
5. Put it in a header, as `API-Version: 2` or `Stripe-Version: 2026-07-29`. URLs stay stable and the change is invisible in logs and links, which is both the benefit and the drawback: it is easy to forget you set it.
6. Put it in the media type, as `Accept: application/vnd.kedbyte.v2+json`. This is what content negotiation in HTTP was actually designed for, and it is the most correct answer. It is also the one nobody finds friendly, and it breaks casual testing in a browser.
7. Use dates instead of numbers, so each account is pinned to the behaviour of the day it joined. Very kind to callers, very expensive to run, because the server keeps every old shape alive behind translation layers.
8. Or do not version at all. Only make additive changes: new fields may appear, nothing is ever removed or given a new meaning. This is what GraphQL encourages and what many internal APIs do. It works until the day you genuinely need to remove something, and then it does not.

9. Whichever you choose, the deprecation mechanics are the same. Announce with a date. Send a header on every response to the old thing. Publish what to use instead. Then remove it on the day you said.
10. A useful middle step is the **brownout**: on announced days before the removal, turn the old endpoint off for an hour, so that anybody still calling it notices while there is still time.
11. Now the machine-readable description. A file such as an OpenAPI document lists every path, every parameter, every response shape and every error.
12. Once that file exists, tools can read it and produce: reference documentation, client libraries in a dozen languages, a mock server that returns example data, request validation at the gateway, and a diff between two versions that tells you whether a change is breaking.
13. The important consequence is subtle. The specification stops being a document that describes the code and starts being a thing the code is checked against. Drift becomes a test failure instead of a support ticket.

36.12.5 TECHNICAL - the engineer's version

1. Error format. RFC 9457, Problem Details for HTTP APIs, published July 2023, obsoletes RFC 7807 from March 2016. Media type `application/problem+json`. Members: `type` (a URI reference identifying the problem type), `title`, `status`, `detail`, `instance`, plus any extension members you add.
2. RFC 9457 is explicit that `detail` is for humans and that clients should key off `type`. Add your own short `code` extension if you want a compact stable token; that is a convention, not part of the RFC.
3. Do not put secrets, stack traces or internal hostnames in `detail`. Error bodies are logged, screenshotted and pasted into public issue trackers.
4. Versioning strategies, compared honestly:

Strategy	Example	Main cost
URL path	`/v1/orders`	URLs churn on upgrade
Header	`API-Version: 2`	Invisible, easy to forget
Media type	`vnd.x.v2+json`	Correct but unfriendly
Date-pinned	`2026-07-29`	Server keeps every shape
None, additive	no marker	Cannot ever remove

5. Real practice, with dates. Stripe pins each account to a dated version and names major releases; as of this writing the current version string is `2026-07-29.dahlia`, sent in the `Stripe-Version` header, with monthly releases guaranteed backward compatible and named releases such as Acacia and Basil marking the breaking ones. GitHub uses a dated `X-GitHub-Api-Version` header. Kubernetes uses group-and-version in the path, such as `apps/v1`, with an explicit alpha, beta, stable ladder.
6. Experts genuinely disagree here. Roy Fielding has argued publicly that versioning a REST API in the URL is an admission that hypermedia was not used and that clients were coupled to URL structure. The pragmatic camp replies that `/v1/` is legible, greppable and routable, and that shipping beats purity. Both are right about different things.
7. What counts as a breaking change is the part teams fail to write down. A workable rule: adding an optional field, adding a new endpoint, and adding a new enum value to an output are non-breaking. Removing or renaming a field, adding a required input, narrowing a type, changing a status code, and changing the meaning of an existing value are breaking.
8. That third one bites. Adding a new enum value is only safe if you told clients in advance to tolerate unknown values. Say it in the documentation on day one, because you cannot add the rule retroactively.
9. Deprecation headers. RFC 9745, The Deprecation HTTP Response Header Field, published 17 March 2025, defines `Deprecation` as a structured field date. RFC 8594, May 2019, defines `Sunset` as the date the resource stops responding. RFC 9745 requires that `Sunset` not be earlier than `Deprecation`.

```
HTTP/1.1 200 OK
Deprecation: @1780272000
```

```
Sunset: Wed, 30 Sep 2026 23:59:59 GMT
Link: </docs/migrate-v2>; rel="deprecation"; type="text/html"
```

10. The `@` form is the structured-field syntax for a Unix timestamp; here it is 1 June 2026. `Link` with `rel="deprecation"` points at the migration guide, using web linking from RFC 8288, October 2017.
11. A humane deprecation policy in five parts: a minimum notice period, stated publicly, of 6 to 12 months for a paid public API; response headers from day one of the notice; direct email to accounts still calling it; a migration guide with before-and-after examples; and scheduled brownouts.
12. OpenAPI. History: Swagger began in 2010, written by Tony Tam at Wordnik; SmartBear donated the specification to the OpenAPI Initiative under the Linux Foundation in November 2015; it was renamed the OpenAPI Specification on 1 January 2016. Version 3.0.0 was released 26 July 2017, 3.1.0 on 18 February 2021, and 3.2.0 on 23 September 2025. Version 3.1 matters most, because it made the schema language a proper superset of JSON Schema 2020-12.
13. A minimal but real OpenAPI document:

```
openapi: 3.1.0
info:
  title: KedByte Orders API
  version: 1.4.0
paths:
  /orders/{orderId}:
    get:
      operationId: getOrder
      summary: Fetch one order
      parameters:
        - name: orderId
          in: path
          required: true
          schema: { type: string, pattern: "^ord_[a-z0-9]+$" }
      responses:
        "200":
          description: The order
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Order"
        "404":
          description: No such order
          content:
            application/problem+json:
              schema:
                $ref: "#/components/schemas/Problem"
components:
  schemas:
    Order:
      type: object
      required: [id, status, amount_minor, currency]
      properties:
        id: { type: string }
        status:
          type: string
          enum: [pending, paid, shipped, cancelled]
        amount_minor: { type: integer, minimum: 0 }
        currency: { type: string, minLength: 3, maxLength: 3 }
        created_at: { type: string, format: date-time }
```

14. What that file buys you, concretely: `swagger-ui` and `redoc` render documentation; `openapi-generator`

and `oapi-codegen` emit clients and server stubs; `prism` runs a mock server; `schemathesis` and `dredd` test the live API against the document; `oasdiff` reports breaking changes between two revisions; and gateways including Kong, Envoy and Amazon API Gateway can validate requests before they reach your code.

15. Equivalents in other ecosystems: `.proto` files for gRPC, the GraphQL SDL for GraphQL, WSDL for SOAP, and AsyncAPI for event-driven and webhook interfaces. OpenAPI 3.1 also gained a top-level `webhooks` section, so outbound callbacks can be described in the same document.
16. Generated SDKs, honestly. They give type safety, retries, pagination helpers, authentication and release-note churn for free, and they are how Stripe and Twilio made their APIs pleasant. The costs are real: generated code is often unidiomatic, one bad generator template produces the same bug in twelve languages at once, and every extra language is an extra thing to support forever. Many teams generate a thin core and hand-write the ergonomic layer on top.
17. Sandboxes. The requirements that matter: separate credentials that cannot touch production, deterministic magic values that force specific outcomes, no real money or messages, and the same version and error behaviour as production. Stripe's test card numbers are the canonical example, where a specific card number always produces a specific decline code.
18. Money and time, since they cause more API bugs than anything else. Money: integer minor units plus an ISO 4217 currency code, never a binary floating-point number, because 0.1 plus 0.2 is not 0.3 in IEEE 754. Time: RFC 3339 timestamps in UTC with an explicit offset, and a separate IANA time zone identifier such as `Asia/Kolkata` when local meaning matters. Never a bare local time.
19. Documentation that works has four layers: a five-minute quickstart that produces a real result, task guides for the common jobs, generated reference for every field, and a changelog with dates. The quickstart is the one that decides whether a developer stays.

36.12.6 WORDS - remember these

1. Error code — a short stable name for a failure — a machine-readable token that is part of the contract, unlike the human-readable message.
2. Problem Details — the standard shape for an error body — RFC 9457, media type `application/problem+json`, with `type`, `title` and `status`.
3. Breaking change — existing callers stop working — any change that invalidates a promise the previous version made, whatever the version number.
4. Deprecation — announced, still working — RFC 9745 `Deprecation` header marking a resource as scheduled for removal, with a migration path.
5. Sunset — the date it stops answering — RFC 8594 `Sunset` header giving the time after which a resource becomes unresponsive.
6. Brownout — a rehearsal for the switch-off — a deliberate short outage of a deprecated endpoint to surface remaining callers before removal.
7. OpenAPI — a description of the API a machine can read — the OpenAPI Specification, 3.1.0 from 2021 and 3.2.0 from 2025, formerly Swagger.
8. SDK — a ready-made client library — a generated or hand-written package wrapping transport, authentication, retries and pagination for one language.
9. Sandbox — a safe copy to experiment in — an isolated environment with test credentials, deterministic fixtures and no real-world side effects.

36.13 Consuming an API well

36.13.1 PLAIN - in simple words

1. Most of your working life you will be the caller, not the builder. Being a good caller is a skill, and it is mostly discipline.
2. **Read the documentation first.** Not the tutorial. Find four things: the base address, how to authenticate, the rate limits, and how lists are paged.
3. **Try it by hand before writing code.** One `curl` command tells you more than an hour of guessing, because it shows you the real bytes.
4. **Check the status code.** The most common bug in code that calls APIs is assuming it worked and reading a field out of an error body.
5. **Respect the rate limit.** The service is telling you, in a header, how much you have left. Read it. Slow down before you are refused.
6. **Do not fetch what has not changed.** If the server gave you a label for the version you already have, send it back, and it will tell you "no change" with an empty body.
7. **Page properly.** Lists are not lists. They are the first page of a list, and there is a documented way to ask for the next one. Use it, in a loop, with a stopping condition.
8. **Never put a secret in anything the user can see.** A key in a web page, a phone app, or a browser extension is a public key, no matter what you named the variable.
9. That last one is not a style preference. It is the reason a large number of real breaches happened, and the fix is always the same: keep the secret on a server you control and let the user's device talk to that.

36.13.2 PLAIN - a picture in your head

1. Calling somebody else's API is like being a guest researcher in a large library that belongs to another institution.
2. You get a reader's card. It is yours, it has your name on it, and lending it to a friend gets you both banned. That is your token.
3. The library allows twenty book requests an hour. A sign at the desk tells you how many you have left today. That sign is the rate-limit header.
4. Books come from the stacks a trolley at a time, twenty per trolley, with a slip saying "more available". That slip is the pagination link.
5. If you already copied a chapter, the librarian will happily tell you "that chapter has not been revised since you last saw it" instead of fetching the whole volume again. That is the conditional request.
6. And when the librarian says no, they say why: no such book, not allowed, closed for lunch, or you have asked too often. Four different answers that need four different reactions from you.

Where this comparison breaks:

7. A librarian notices a rude guest and speaks to them. An API notices nothing and simply blocks your account, often without warning and often at 3 a.m.
8. A human researcher naturally slows down when tired. A `for` loop does not, so the politeness must be written into the code deliberately.

36.13.3 PLAIN - a worked example

1. A full session against a real public API, using the GitHub REST API. The header names and limits below are exactly as GitHub documents them.
2. Step one, the plain call, with `-i` so we see the response headers as well as the body:

```
curl -i \
  -H "Accept: application/vnd.github+json" \
  -H "X-GitHub-Api-Version: 2022-11-28" \
  https://api.github.com/repos/kedbyte/book
```

3. Step two, read the top of the answer before reading any of the body:

```
HTTP/2 200
content-type: application/json; charset=utf-8
etag: W/"7f2c1a9b6e0d4c8f1a2b3c4d5e6f7a8b"
x-ratelimit-limit: 60
x-ratelimit-remaining: 57
x-ratelimit-used: 3
x-ratelimit-reset: 1767229200
x-ratelimit-resource: core
```

4. Four facts, immediately. The call worked. There is an `etag` label for this version. We have 57 of 60 calls left this hour. The window resets at the epoch second 1767229200, which is 1 January 2026 at 01:00 UTC.
5. The limit is 60 because we sent no token. With a personal access token the same headers would say 5,000. That single header is why anonymous scripts fail at scale and authenticated ones do not.
6. Step three, ask again politely, sending the label back:

```
curl -i -H "If-None-Match: \
W/\"7f2c1a9b6e0d4c8f1a2b3c4d5e6f7a8b\"" \
  https://api.github.com/repos/kedbyte/book
```

7. If nothing changed, the answer is:

```
HTTP/2 304
etag: W/"7f2c1a9b6e0d4c8f1a2b3c4d5e6f7a8b"
x-ratelimit-remaining: 57
```

8. Read the last line carefully. `x-ratelimit-remaining` did not go down. GitHub documents that a conditional request returning 304 does not count against your primary rate limit when the request was correctly authorized.
9. So conditional requests are not merely a bandwidth saving. On this API they are free calls. A poller that ignores ETags burns its quota on data it already has.
10. Step four, handling the failures rather than assuming success. Here is the shape every calling script should have:

```
code=$(curl -s -o body.json -w '%{http_code}' \
  --connect-timeout 5 --max-time 30 \
  -H "Authorization: Bearer $GH_TOKEN" \
  https://api.github.com/repos/kedbyte/book)

case "$code" in
  200) jq -r '.full_name, .stargazers_count' body.json ;;
  304) echo "unchanged, using cached copy" ;;
  401) echo "token missing or invalid"; exit 1 ;;
  403|429) echo "rate limited; see reset header"; exit 2 ;;
  404) echo "no such repo, or token cannot see it"; exit 3 ;;
  5*)  echo "server side; retry with backoff" ; exit 4 ;;
  *)   echo "unexpected status $code"; exit 5 ;;
esac
```

11. Note 404. On GitHub a private repository you cannot see returns 404, not 403, on purpose, so that the API does not leak the existence of private things. Your handler must not treat 404 as "definitely does not exist".
12. Step five, pagination in a loop. GitHub returns a `Link` header holding the next page's address, and you follow it until there is no next.

```
url="https://api.github.com/repos/kedbyte/book/issues?per_page=100"
while [ -n "$url" ]; do
  curl -sS -D head.txt -o page.json \
```

```
    -H "Authorization: Bearer $GH_TOKEN" "$url"
  jq -r '.[] | "\(.number) \(.title)"' page.json
  url=$(tr -d '\r' < head.txt | grep -i '^link:' \
    | tr ',' '\n' | grep 'rel="next"' \
    | sed -e 's/.*<\(.*\)>.*/\1/')
  sleep 1
done
```

13. Three things make that loop safe. It follows the server's own link instead of building page numbers itself. It stops when there is no next link. And it pauses a second between pages, which is politeness in one word.
14. The same call in `httpie`, which is friendlier to type and colours the output, though the plumbing is identical:

```
https api.github.com/repos/kedbyte/book \
  Accept:application/vnd.github+json \
  Authorization:"Bearer $GH_TOKEN"
```

15. And the secret. Look at the two examples above: the token came from an environment variable, `$GH_TOKEN`, never from a literal string in the file.
16. That habit is what stops a token being committed to git, pasted into a bug report, or shipped inside a web page.

36.13.4 PLAIN - what is really happening inside

1. A conditional request is a small conversation about versions.
2. On the first response the server sends an `ETag`, which is just an opaque label for the exact bytes it sent. You store it beside the data.
3. Next time you send `If-None-Match` with that label. The server compares. If the current label matches, it stops right there and sends 304 Not Modified with a status line, some headers, and no body at all.
4. The saving is the body, which is usually 99 percent of the bytes, plus, on some APIs, the rate-limit charge.
5. There is a second form, `If-Modified-Since` with a date, which is older and less precise because it only has one-second resolution. Prefer ETags.
6. The same machinery, used the other way round, prevents lost updates. You send `If-Match` with the ETag on a write, and the server refuses with 412 Precondition Failed if somebody else changed the thing meanwhile.
7. Pagination has two families, and knowing which you are in matters.
8. Offset paging asks for "rows 200 to 299". It is simple and it is wrong under change: if a row is inserted at the top while you are paging, everything shifts down by one and you see one item twice and miss another.
9. Cursor paging asks for "carry on after this marker". The marker encodes the position in a stable sort order, so insertions elsewhere do not disturb you.
10. This is why big APIs moved to cursors, and why the `Link` header pattern hands you a complete next address instead of a page number to compute.
11. Now the secret. When you put a key in browser JavaScript, the browser must be able to read it in order to send it. So the user can read it too.
12. Minifying does not hide it. It is in the network tab of the developer tools the moment the request is made. In a mobile app it is in the binary, and `strings` on the file finds it in seconds.
13. The correct shape is a small server of your own in the middle. The browser calls your server with the user's own session. Your server holds the third party's key and calls the third party. The key never leaves your machine.
14. There is one legitimate exception, and it is a different kind of value: a publishable key designed to be public, such as a Stripe publishable key or a Maps key restricted by referring domain. These are

identifiers, not credentials, and the vendor says so in the documentation. If the documentation does not say so, it is a secret.

36.13.5 TECHNICAL - the engineer's version

1. Tooling, and what each is genuinely for:

Tool	Strength	Weakness
curl	Exact bytes, everywhere	Verbose to type
httpie	Readable, JSON-aware	Extra install
Postman	Collections, teams	Heavy, cloud-linked
jq	Filtering JSON in shell	Own syntax to learn

2. `curl` flags worth memorizing: `-i` include response headers, `-D file` dump headers to a file, `-o file` body to a file, `-s` silent, `-S` still show errors, `-w '%{http_code}'` print the status, `-L` follow redirects, `--fail-with-body` return a non-zero exit code on 4xx and 5xx while still printing the body, `--compressed` request gzip, `-v` full trace, and `--connect-timeout` with `--max-time` for deadlines.
3. `--fail-with-body`, added in curl 7.76 (March 2021), is the flag that fixes shell scripts which silently succeed on an error response. Plain `-f` discards the body, which throws away the reason.
4. Conditional requests are specified in RFC 9110, June 2022, sections 8.8 and
   13. `ETag` may be strong, `"abc"`, or weak, `W/"abc"`. Weak means semantically equivalent but not byte-identical, which is why weak validators are valid for caching but not for `If-Match` on writes.
5. Caching headers you should actually read: `Cache-Control` with `max-age`, `no-cache` (revalidate, do not blindly reuse), `no-store` (do not keep at all), and `private` versus `public`. RFC 9111 is the caching specification. `no-cache` and `no-store` mean different things and are constantly confused.
6. Rate-limit handling, in the order GitHub itself recommends: if a `retry-after` header is present, wait that long; else if `x-ratelimit-remaining` is zero, wait until `x-ratelimit-reset`; else wait at least one minute and then back off exponentially. GitHub states plainly that continuing to call while limited may get your integration banned.
7. Concurrency is where well-behaved clients go wrong. Two parallel workers share one quota, so eight workers exhaust a 5,000-per-hour budget eight times faster. Rate-limit your own client, do not rely on being refused.
8. Pagination in the wild:

Style	Request looks like	Trouble
Page number	`?page=3&per_page=100`	Shifts as data changes
Offset	`?offset=200&limit=100`	Same shift, slow deep
Cursor	`?after=Y3Vyc29yOjIwMA`	Opaque, cannot jump
Link header	follow `rel="next"`	Must parse the header

9. The `Link` header is RFC 8288, October 2017, Web Linking. GitHub sends `rel="next"`, `rel="prev"`, `rel="first"` and `rel="last"`, and its guidance is to follow the header rather than construct page URLs yourself.
10. Client-side secrets, precisely. A single-page app, a mobile binary, a desktop Electron app and a browser extension are all **public clients** in OAuth terms: they cannot keep a secret. RFC 9700, the January 2025 Best Current Practice for OAuth 2.0 Security, requires public clients to use the authorization code flow with PKCE and forbids the implicit grant, exactly because there is nowhere safe to put a client secret.
11. Detection and cleanup: `gitleaks` and `trufflehog` scan repositories, GitHub push protection blocks known secret formats at push time, and GitHub secret scanning notifies partner providers so leaked

keys can be revoked automatically. Rotating a leaked key is not optional; git history is permanent, and removing the file does not remove the commit.

12. Server-side proxy pattern, in three rules: the browser authenticates to your server with the user's own session; your server holds the third-party credential in a secrets manager or environment variable; and your server enforces its own rate limit and authorization, because otherwise you have simply published the upstream API with your key attached.
13. Error handling that survives production: distinguish retryable from terminal (see section 36.10), log the request identifier the API returns, such as GitHub's `x-github-request-id`, and quote it in support tickets. Never log the `Authorization` header, and be careful with `-v`, which prints it in full.
14. Testing your client without the network: record and replay with VCR-style libraries, run a mock server generated from the OpenAPI document, or point at the vendor sandbox. Test the failure paths on purpose: 429, 500, timeouts, and a truncated response body.
15. Observation: `curl -w` with a format string prints `time_namelookup`, `time_connect`, `time_appconnect`, `time_starttransfer` and `time_total`, which separates DNS from TCP from TLS from server think time, and turns "the API is slow" into a specific accusation.

36.13.6 WORDS - remember these

1. Conditional request — asking only if it changed — a request carrying `If-None-Match` or `If-Modified-Since`, answered with 304 when unchanged.
2. ETag — a label for one version of a resource — an opaque validator, strong or weak, compared by the server rather than interpreted by the client.
3. 304 Not Modified — nothing new, keep what you have — a bodyless response confirming the client's cached copy is current.
4. Offset pagination — ask for rows N to M — index-based paging that skips or repeats items when the underlying data changes mid-scan.
5. Cursor pagination — carry on after this marker — keyset paging using an opaque stable position, immune to concurrent insertions.
6. Link header — the server tells you the next address — RFC 8288 web linking with relations such as `next`, `prev`, `first` and `last`.
7. Public client — software that cannot keep a secret — any client whose code or binary is in the user's hands, per RFC 6749 and RFC 9700.
8. Backend for frontend — your own small server in the middle — a proxy holding third-party credentials so the user's device never sees them.

36.14 APIs as a business

36.14.1 PLAIN - in simple words

1. So far an API has been an engineering thing. It is also a commercial thing, and some of the largest companies alive are shaped by that fact.
2. A **product** is something people use. A **platform** is something people build on. An API is how a product becomes a platform.
3. When other people build on you, they do work you did not pay for, and their customers become reachable by you. That is why a company gives data away.
4. **API-first** means designing the interface before writing the software, and making your own teams use the same interface everyone else uses.
5. That sounds like a small internal rule. In one famous case it reshaped an entire company and then created a new industry. Section 36.14.5 gives the dates.
6. Money comes from APIs in several ways: charging per call, charging by tier, giving it away to sell something else, or taking a share of what flows through it.

7. Then the uncomfortable part. If your business runs on somebody else's API, you do not control your own product. They can change the price, change the rules, or switch it off.
8. This is not a theoretical worry. Real companies have been closed by an announcement from a platform they depended on, and section 36.14.3 names them with years.
9. The lesson is not "never build on an API". Almost everything is built on somebody's API. The lesson is to know exactly how exposed you are, and to price that risk instead of ignoring it.

36.14.2 PLAIN – a picture in your head

1. Think of a railway company that has built a line between two cities.
2. It could run only its own trains. It carries its own goods, it keeps all the revenue, and the size of the business is the size of its own trade.
3. Or it can sell track access. Now other companies run trains on its rails. Each of them invests in carriages and customers that the railway never had to pay for.
4. The railway earns a fee from every train, and the line becomes more valuable the more operators use it, because the destinations are better served.
5. That is the platform move. The track is the API.
6. Now notice the power that comes with owning track. The railway sets the access charge, the timetable slots, and the safety rules.
7. An operator who has bought fifty carriages and hired three hundred staff cannot easily leave when the access charge triples. Their investment is welded to that particular line.
8. And if the railway decides tomorrow to run only its own trains again, those fifty carriages are scrap metal.

Where this comparison breaks:

9. Track takes years to build and is regulated by governments. An API can be changed in an afternoon by one team, and almost nowhere regulates it.
10. A railway operator can see the rails and count the trains. An API user often cannot even tell how many other users there are, or whether they are a rounding error to the platform or its main source of value.

36.14.3 PLAIN – a worked example

1. Here are real cases, with dates. Read them as case law, because that is effectively what they are.

Year	Platform	What changed
2014	Facebook Graph API	Friend data cut off
2014	Netflix	Public API closed
2018	Google Maps	Price up about 14 times
2023	Twitter, now X	Free access ended
2023	Reddit	Charged for API calls

2. Netflix, 2014. On 13 June 2014 Netflix announced that its public API would close, and it did close on 14 November 2014. Applications built on it, including catalogue browsers and recommendation tools, simply stopped.
3. Facebook, 2014 to 2015. Graph API v2.0 arrived on 30 April 2014 and removed the broad access to a user's friend list that many social apps had been built on. Version 1.0 was fully retired on 30 April 2015. A generation of "find your friends who also use this" features died in that change.
4. Google Maps, 2018. Announced in early May 2018 and effective on 16 July 2018, the pricing moved from about 0.50 US dollars per 1,000 dynamic map loads to about 7.00 US dollars per 1,000, a rise of roughly fourteen times.
5. The free allowance changed shape too: from 25,000 map loads per day to a 200 US dollar monthly credit, worth roughly 28,000 loads per month.

6. Read those two numbers together. A site doing 25,000 loads a day was inside the free tier before, and afterwards was about 27 times over the monthly credit. Many small mapping sites moved to OpenStreetMap-based providers or closed the feature.
7. Twitter, 2023. On 19 January 2023 the developer agreement was changed to ban third-party clients. Tweetbot and Twitterrific, which had existed since 2007 and 2008, shut down; Twitterrific had helped popularize the word "tweet" and the bird icon.
8. In February 2023 free API access ended. On 29 March 2023 the new tiers arrived: a free tier limited to 1,500 posts per month and login, a Basic tier at 100 US dollars per month, and an Enterprise tier reported at around 42,000 US dollars per month.
9. Academic research that had depended on cheap Twitter data largely stopped. That was not a side effect anybody had budgeted for.
10. Reddit, 2023. Announced on 18 April 2023 and effective 30 June 2023. Christian Selig, the developer of the popular client Apollo, calculated that the new rate would cost him about 20 million US dollars per year.
11. Apollo, Sync, BaconReader and rif is fun all shut down on 30 June 2023. Between 12 and 14 June 2023, thousands of subreddits went private in protest, and many stayed dark afterwards.
12. Reddit's chief executive, Steve Huffman, gave the reason openly at the time: the data was valuable, and giving it free to the largest companies in the world was no longer the plan. That is the honest commercial logic behind most of these changes, and it arrived alongside the demand for training data for language models.
13. Now the pattern across all five. In every case the platform was inside its rights, the change was announced, the notice was months not years, and the businesses that died were the ones with no second source.

36.14.4 PLAIN - what is really happening inside

1. Why would a company give its data away at all? Five reasons, and they are not charity.
2. Distribution. Every app built on your API is a salesperson. Slack, Stripe and Twilio grew mostly this way.
3. Lock-in through investment. Once a customer has written six months of code against your interface, moving costs them six months.
4. Data. Every call tells you what people want, in aggregate, before your competitors know.
5. Standards. If your shape of the problem becomes the shape everyone uses, the default choice is yours. Amazon's S3 interface became the shape of object storage, and rivals now advertise being compatible with it.
6. Complements. If maps are cheap, more apps need location, and location services are what you actually sell.
7. Then the money models:

Model	You pay	Typical example
Free, funded elsewhere	Nothing	Search, social login
Freemium quota	Above a limit	Maps, weather
Per call	Each request	Messaging, translation
Tiered subscription	Per month	Data and analytics
Revenue share	A cut of value	Payments, marketplaces

8. Revenue share is the strongest of these, because the API provider's income grows with the customer's success rather than with their request count. It is why payment APIs are such durable businesses.
9. Now the risk, from the caller's side. Ask four questions about any API you build on, before you build.
10. What fraction of my product stops working if this disappears tomorrow?
11. Is there a second supplier with a similar interface, and how many days of work is the switch?

12. Am I paying? Free users are the first to be cut, because they are a cost with no contract.
13. Is my usage growing towards a tier boundary that changes the price by a factor rather than a percentage?
14. The mitigations are unglamorous and they work: put an interface of your own in front of theirs so the dependency touches one file rather than fifty; cache what you are permitted to cache; export your own data regularly; read the terms of service for the notice period; and keep a written estimate of the switching cost.
15. The one that people skip is the first. If every part of your codebase calls the vendor's SDK directly, then swapping vendors means editing everything. If they all call your own thin wrapper, it means editing one adapter.

36.14.5 TECHNICAL - the engineer's version

1. The Bezos mandate. Around 2002, Jeff Bezos issued an internal directive at Amazon that all teams must expose their data and functionality through service interfaces, must communicate only through those interfaces, and must design them to be externalizable, with termination as the stated consequence for non-compliance.
2. Be honest about the source. Amazon never published the memo. The wording everyone quotes comes from a long public post written by the engineer Steve Yegge in October 2011, recalling it from his time at Amazon. So treat the exact phrasing as recollection, not as a document.
3. The outcome, however, is checkable. Amazon Web Services launched publicly in 2006: Simple Storage Service in March 2006 and Elastic Compute Cloud in August 2006. A company that had already forced every internal team to speak only through network interfaces was in a position to sell those interfaces.
4. That is the extreme case of taking internal APIs seriously, and it is the subject of Chapter 43, which covers cloud computing, the mandate, and how AWS and Azure turned internal engineering discipline into an industry.
5. Metrics that API businesses actually run on, distinct from web metrics:

Metric	What it measures
Time to first call	Signup to first 200 response
Time to first value	Signup to first real result
Calls per active key	Depth of integration
Error rate by code	Where users get stuck
Version distribution	How much old shape to keep

6. Time to first call is the one that predicts adoption. Stripe's reputation was built substantially on making it a few minutes, with a working `curl` command on the front page of the documentation.
7. API-first as an engineering practice means: design and review the interface document before implementation, generate server stubs and client libraries from it, run contract tests in continuous integration, and treat a breaking diff as a build failure. Chapter 41 covers the continuous integration machinery this depends on.
8. The internal-external question is a genuine disagreement among architects. One camp says internal and external APIs should be the same interface, because dogfooding is the only reliable quality signal. The other says external APIs need stability guarantees, versioning, documentation and support that would paralyse internal development, so they should be a separate, deliberately narrower layer. Amazon's mandate takes the first position; many large companies quietly take the second.
9. Deprecation as a commercial act, not just a technical one. The pattern that holds up in court and in public opinion is: written notice period stated in the terms of service, public changelog entry with a date, `Deprecation` and `Sunset` headers from day one (RFC 9745 and RFC 8594), direct contact with accounts still calling the endpoint, and a migration guide.

10. Legal reality, briefly and carefully. In Google versus Oracle America, decided by the United States Supreme Court on 5 April 2021, the court held that Google's reuse of roughly 11,500 lines of Java SE API declaring code in Android was fair use. The court assumed without deciding that the declarations were copyrightable, so it did not settle whether API definitions can be copyrighted; it settled that this particular reuse was fair. Anyone telling you the case made APIs freely copyable is overstating it.
11. Practical consequence of that case for engineers: reimplementing somebody else's interface is a normal and long-standing practice, and it is how S3-compatible storage, PostgreSQL wire-protocol clones and POSIX reimplementations exist. It is not risk-free, and it is not settled law outside the United States.
12. Failure modes of an API business, in one list: pricing that punishes exactly the users who grow; a free tier that attracts users the business model cannot support; breaking changes without a version; documentation that lags the implementation; and no sandbox, so nobody can evaluate you without committing.
13. And the strategic trap on the other side: a platform that opens an API, lets an ecosystem grow, then builds the most profitable ecosystem product itself and restricts the interface that made the competitor possible. Every example in section 36.14.3 has some version of this in its history, and it is the strongest argument for owning your own second source.

36.14.6 WORDS - remember these

1. Platform — something other people build on — a product whose value grows with third-party integrations rather than only with its own features.
2. API-first — the interface is designed before the code — a practice where the specification is the source of truth and implementations are generated from or validated against it.
3. Ecosystem — the businesses living on your interface — the set of third-party products whose viability depends on a platform's continued access terms.
4. Lock-in — leaving is expensive — accumulated switching costs from code, data, staff knowledge and contracts tied to one provider.
5. Sunset, commercially — the announced end of an interface — the dated withdrawal of an API, governed by the terms of service notice period.
6. Second source — an alternative supplier ready in advance — an equivalent provider behind your own abstraction layer, bounding your switching cost.
7. Bezos mandate — talk only through interfaces — the roughly 2002 Amazon directive requiring all internal capability to be exposed as externalizable service interfaces, recounted publicly in 2011.

36.98 Common wrong ideas

1. Wrong: an API means a web endpoint. Right: an API is any published contract between software parts. `strlen`, `write` and Vulkan are APIs with no network anywhere near them.
2. Wrong: REST means JSON over HTTP. Right: REST is six constraints defined by Roy Fielding in 2000 and says nothing about JSON. Almost every "REST API" in production is really an HTTP JSON API, because it omits the hypermedia constraint.
3. Wrong: GraphQL replaces REST. Right: GraphQL solves over-fetching and round-trip count for client-driven queries, at the cost of caching, observability and query-complexity control. Many systems run both, for different callers.
4. Wrong: an API key is authentication. Right: a key identifies an account. It rarely expires, is rarely scoped, is often committed to git or logged in a query string, and proves nothing about who is holding it right now.
5. Wrong: versioning means adding `v2` to the URL. Right: versioning is a policy about how long you will run old behaviour and who pays for it. The URL, a header, a media type and a pinned date are four ways to express that policy, and none of them is the policy itself.
6. Wrong: a failed request means nothing happened. Right: a lost response is indistinguishable from a lost request at the client. The work may be done. That is partial failure, and the fix is idempotency, not

a better error message.

7. Wrong: retrying automatically is always safe. Right: retrying a non-idempotent request without an idempotency key can charge a customer twice, and retrying without jitter can turn a brief outage into a long one.
8. Wrong: a JWT is encrypted, so claims inside it are private. Right: a JWT is base64url-encoded and signed, not encrypted. Anyone holding it can read every claim. The signature proves integrity, not secrecy.
9. Wrong: the API key in my mobile app or single-page app is hidden. Right: any credential shipped to a user's device is public. Minification, obfuscation and native binaries all lose to developer tools and the `strings` command.
10. Wrong: my webhook handler is fine because it parses the JSON and then checks the signature. Right: the signature covers the exact bytes that arrived. Re-serializing changes the bytes and the check will fail, or worse, be skipped. Verify the raw body first.

36.99 Chapter summary in 20 lines

1. An API is a published contract about how one piece of software may ask another to do something: what you may ask, how, and what comes back.
2. It is not a technology. It exists at every scale, from `strlen` inside one file to a service on the other side of the planet, and the idea never changes.
3. An API has a surface, a shape and a behaviour. Beginners see only the surface; almost all real bugs live in the behaviour.
4. Nobody invented APIs in one moment. Subroutine libraries appeared around 1951, the phrase "application program interface" in print in 1968, remote procedure calls with Birrell and Nelson in 1984, CORBA in 1991, SOAP and Fielding's REST thesis around 2000, and GraphQL and gRPC from 2015.
5. Every generation that tried to hide the network behind a local-looking call eventually failed at the same place: partial failure and latency.
6. An API is not an ABI. An ABI fixes registers, calling conventions and struct layout, which is why code can compile and still crash at run time.
7. Semantic versioning states the promise in three numbers: patch fixes, minor adds compatibly, major breaks. The promise is social, not enforced.
8. Crossing a machine boundary through system calls costs a mode switch; crossing the planet costs about ten million times a local call.
9. The eight fallacies of distributed computing, from Sun Microsystems between 1994 and 1997, each name a specific cost you pay for believing them.
10. HTTP is a request line, headers, a blank line and an optional body, both ways. Methods differ in safety and idempotency; status codes are grouped 1xx to 5xx; HTTP/2 and HTTP/3 changed the framing, not the semantics.
11. REST is six constraints, and the one almost nobody implements is the uniform interface's hypermedia requirement. Say "HTTP JSON API" when that is what you mean.
12. GraphQL, gRPC, SOAP, JSON-RPC, WebSockets and server-sent events each win on a different axis: payload size, typing, streaming, tooling or reach.
13. Authentication asks who you are; authorization asks what you may do. API keys answer neither well. Bearer tokens expire, which is their main gain.
14. OAuth 2.0 with PKCE, now required for all client types by RFC 9700 of January 2025, lets an app act for you without ever seeing your password.
15. A JWT is a signed, readable token, not an encrypted one, and a scope limits the token rather than the user.
16. Reliability is six habits: rate limits with token or leaky buckets, retries with exponential backoff and jitter, idempotency keys, timeouts at every layer, circuit breakers, and graceful degradation.

17. Retrying a non-idempotent request without an idempotency key can double charge a real person, and retrying without jitter can extend an outage.
18. Webhooks invert the call: the server posts to you. Verify the HMAC over the raw body, deduplicate on the event identifier, expect any order, answer fast, and use a reverse tunnel to receive them behind NAT.
19. A usable API is consistent, has machine-readable error codes, an honest versioning and deprecation policy with `Deprecation` and `Sunset` headers, an OpenAPI description, generated SDKs and a sandbox.
20. Commercially, an API turns a product into a platform, and the shutdowns of Netflix in 2014, Google Maps pricing in 2018, Twitter and Reddit in 2023 show what a platform's decision can do to businesses built on it. Amazon's roughly 2002 internal mandate is the extreme case, and Chapter 43 follows it.

37

CHAPTER

KB

Git — Why It Works With No Network

37.0 What this chapter gives you

1. You will be able to say what version control is for, and name the six real problems it solves, rather than repeating that it "saves your work".
2. You will be able to state the difference between a centralized and a distributed version control system in one sentence, and defend it.
3. You will be able to explain why the reader could keep committing to branches for the whole length of their network outage, and push everything later.
4. You will be able to open a `.git` directory, name every entry inside it, and say what each one holds.
5. You will be able to say what "the whole history is local" really means, with real numbers: a 316 MiB `.git` next to a 60 MiB working tree.
6. You will be able to explain that `origin` is a nickname for a URL in a text file, that it has no authority, and that a repository may have none.
7. You will be able to explain that `origin/main` is a memory, not a live reading, and say exactly which commands refresh it and which never do.
8. You will be able to classify every common git command as local or remote, and explain why the split falls exactly where it does.
9. You will be able to state the honest costs of the distributed model, and say why large game studios and hardware firms still buy Perforce.
10. You will be able to say plainly that git and GitHub are different things, and point at the line between them.

37.1 What version control is for

37.1.1 PLAIN – in simple words

1. You are writing something over many days. Code, a report, a design.
2. It changes. Sometimes it gets better. Sometimes you break it.
3. A **version control system** is a program that remembers every state your files have ever been in, and lets you go back to any of them.
4. That one sentence hides six separate problems. Here they are.
5. **History.** What did this file look like three weeks ago, before the change that broke everything?
6. **Undo.** Put the whole project back exactly as it was on Tuesday, including files you deleted and files you renamed.
7. **Blame.** Who wrote this strange line, when, and what else did they change at the same moment, and why did they say they did it?
8. **Parallel work.** You want to try a risky idea without disturbing the working version, and abandon it cheaply if it fails.
9. **Backup.** Your laptop is stolen. The work is not gone, because a copy exists somewhere else.

10. **Collaboration.** Four people edit the same project at once and the system combines their work instead of one person silently erasing another.
11. Notice that only two of those six need another computer. Backup needs one. Collaboration needs one. The other four are questions about your own files.
12. Hold on to that. It is the whole answer to why the reader could keep working with no network.

37.1.2 PLAIN - a picture in your head

1. Before version control existed, people did the obvious thing. They copied the folder.
2. `report`, then `report-final`, then `report-final-2`, then `report-final-USE-THIS-ONE`, then `report-final-v2-jan14-shikhar`.
3. It works for about a week. Then it stops working, for reasons that are worth naming one at a time.
4. You cannot tell what changed between two copies without opening both and reading them side by side, line by line.
5. You cannot tell why anything changed. The folder name holds a date at best. It never holds a reason.
6. Disk use grows with the number of copies, not with the amount of change. Ten copies of a 200 MB project is 2 GB, even if you only edited one line.
7. Two people cannot work at once. If both copy the folder and both edit, you have two folders and no way to combine them except by hand.
8. And the names lie. `final` is never final. Everyone has met a `final-v3-really-final`.

Where this comparison breaks:

9. Dated folders are not useless. For a one-person, one-week job with no branching, they are genuinely fine, and pretending otherwise is dishonest.
10. Version control's real win is not that it saves copies. It is that it saves the *reasons* and the *relationships* between copies. A folder cannot say "this version came from that version, and here is why".
11. Also, git does not actually store a full copy per version, which is why ten versions of a big project cost far less than ten folders. That is 37.5.

37.1.3 PLAIN - a worked example

1. Here is a real bug hunt, done with folders, then done with version control.
2. The situation: a program worked on 1 April and is broken on 13 August. There were 400 changes in between.
3. With dated folders you have perhaps twelve snapshots. You open them one at a time until you find the first broken one. Then you diff two folders that are three weeks apart and stare at 6,000 changed lines.
4. With version control you run one command that does a binary search across all 400 changes.
5. Here it is running for real, in a sandbox, on the git project's own source repository, which holds 85,263 commits.

```
$ git bisect start
$ git bisect bad
$ git bisect good <first commit>
Bisecting: 2 revisions left to test after this (roughly 1 step)
[c35f063d2751d7340c2cdd4fe0046a137c9164c4] Offline commit 2
```

6. It halves the range every step. 400 changes needs about 9 tests, because 2 to the power 9 is 512.
7. At the end you do not get "somewhere in three weeks". You get one commit, one author, one timestamp, one message, and the exact lines it touched.
8. And every single step of that ran with no network at all. That matters later.

37.1.4 PLAIN - what is really happening inside

1. A version control system stores two different kinds of thing.
2. First, **content**: the bytes of your files at various moments.
3. Second, **metadata**: who, when, why, and which earlier state this one came from.
4. The second kind is what turns a pile of backups into a history. Backups are a set. A history is a chain.
5. Systems differ in how they store the content. There are three families.
6. **Full snapshots**: keep every version whole. Simple, fast to read, wasteful on disk unless you compress hard afterwards. This is git's model.
7. **Forward deltas**: keep the oldest version whole plus a list of edits to move forward. Cheap to store, slow to get the newest version, because you must replay every edit.
8. **Reverse deltas**: keep the newest version whole plus edits to walk backward. Fast for the common case, because you usually want the newest. This is what RCS chose in 1982.
9. Git looks like family one and behaves like family two after packing. It stores each version as a whole object, then later rewrites groups of objects into a compressed pack with deltas inside it. You never see this happen.

37.1.5 TECHNICAL - the engineer's version

1. The lineage of version control systems, with checkable dates.

System	Year	Origin
SCCS	1972	Rochkind, Bell Labs
RCS	1982	Tichy, Purdue
CVS	1986 / 1990	Grune, then Berliner
Subversion	2000	CollabNet
BitKeeper	2000	McVoy, BitMover
Git	2005	Torvalds

2. **SCCS**, the Source Code Control System, was written by Marc Rochkind at Bell Labs in late 1972, first for an IBM System/370 running OS/360.
3. He rewrote it in C for Unix on a PDP-11 in 1973. The first publicly available version, SCCS version 4, shipped on 18 February 1977 with the Programmer's Workbench edition of Unix.
4. SCCS stored **interleaved deltas**: all versions of a file woven into one file, with markers saying which lines belong to which revision.
5. **RCS**, the Revision Control System, came from Walter F. Tichy at Purdue University, first published in 1982.
6. RCS switched to **reverse deltas**. Tichy's argument was that recent revisions are read far more often than old ones, so the newest should be the cheap one to fetch.
7. Both SCCS and RCS versioned **one file at a time**. There was no concept of a project-wide change. A change touching four files was four unrelated events.
8. **CVS**, the Concurrent Versions System, began as shell scripts by Dick Grune in 1986. Brian Berliner's C rewrite started in April 1989, and CVS version 1.0 was submitted to the Free Software Foundation on 19 November 1990.
9. CVS was a front end over RCS. It added project-wide operation and a client or server split, but it still had no atomic commit: an interrupted commit could leave half the files updated.
10. **Subversion** was founded by CollabNet in 2000, explicitly to be "a better CVS". Version 1.0 was released in February 2004. It gave the world atomic commits, real directory versioning, and proper renames.
11. Subversion became an Apache Incubator project in November 2009 and a top-level Apache project on 17 February 2010.
12. Subversion is still centralized. That is the hinge of this chapter, and 37.2 takes it apart.

13. Tools to observe history in git: `git log`, `git log -p`, `git log --stat`, `git blame`, `git bisect`, `git show`, `git diff`, `git reflog`.

37.1.6 WORDS - remember these

1. Version control — a program that remembers every state of your files — a system that records content plus metadata and the parent relationships between revisions.
2. Commit — one saved point in the history — an immutable object naming a tree, zero or more parents, an author, a committer and a message.
3. Delta — the difference between two versions — a stored edit script that reconstructs one revision from another.
4. Snapshot — a whole copy of everything at one moment — a tree object recording the complete state of the project at one commit.
5. Blame — finding who last changed each line — per-line attribution computed by walking history backward through renames and copies.
6. Bisect — halving the search to find a breaking change — automated binary search over the commit graph with `git bisect`.

37.2 Centralized versus distributed

37.2.1 PLAIN - in simple words

1. This is the most important section in the chapter. Everything else follows from it.
2. There are two ways to build a version control system, and they differ in one question: **where does the history live?**
3. In a **centralized** system, the history lives on a server. Exactly one copy exists, in one place.
4. What is on your machine is a **working copy**. It is the files as they exist at one chosen revision, plus a little bookkeeping. It is not the history.
5. So when you save a change, that change has to travel to the server, because the server is the only thing that can record it.
6. Which means: **a commit is a network operation**. No network, no commit. The server is down, you cannot save your work as a version. You can only edit files and hope.
7. In a **distributed** system, every copy is a complete repository. Not a working copy. A repository, with the entire history in it.
8. Your machine has every commit, every version of every file, every branch, right there on your own disk.
9. So when you save a change, nothing has to travel anywhere. The change is recorded in a directory on your laptop.
10. Which means: **a commit is a local file operation**. No network needed. No server needed. No permission needed.
11. And that is the whole answer to the reader's question. During their outage they were writing to their own disk. There was nothing for the network to break.

37.2.2 PLAIN - a picture in your head

1. Centralized version control is a **library with one copy of each book**.
2. To read a book you go to the library. To add a chapter you go to the library, because the book is there and nowhere else.
3. If the library is shut, or the road is flooded, you cannot add a chapter. You can write on loose paper at home, but that writing is not in the book yet, and nothing keeps track of it.
4. Distributed version control is **everyone owning a full photocopy of every book, including all the old editions**.
5. You write a new chapter into your own copy immediately. It is a real chapter in a real book, dated and numbered, the moment you write it.

6. Later, when you next meet, you and a friend compare copies and exchange the chapters each of you is missing.
7. There is no library in this picture. There can be one by agreement, and there usually is, but the system does not require it.

Where this comparison breaks:

8. Photocopying every book sounds absurdly wasteful, and for paper it would be. Text compresses extraordinarily well, so in practice it is cheap. Real figures are in 37.5.
9. The picture also suggests that merging two copies is easy. It is not always. If you and your friend both rewrote page 40, someone has to decide. Git can combine most changes automatically and refuses when it genuinely cannot.
10. And the honest version: a distributed system does not remove the need for a shared place. Teams still nominate one. The difference is that the shared place is a **convention**, not a structural requirement.

37.2.3 PLAIN - a worked example

1. Two engineers, same task, same broken link. One uses Subversion. One uses git. Here is what each can do.

Action	Subversion	Git
Edit a file	works	works
See old version	needs server	works
Commit a change	needs server	works
Create a branch	needs server	works
See full log	needs server	works
Merge two branches	local, mostly	works
Get others' new work	needs server	needs server
Publish your work	needs server	needs server

2. In Subversion, five of those eight become impossible the moment the link dies. In git, six of the eight keep working.
3. The two that stop are exactly the two that involve another computer: receiving other people's work, and sending yours.
4. That is not a coincidence or a clever optimization. It is the direct, inevitable consequence of where the history is stored.
5. Here is the real demonstration, run in a sandbox with the remote deliberately made unreachable.

```
$ git commit -m "Committed during the outage"
COMMIT SUCCEEDED with no network

$ git fetch
fatal: Could not read from remote repository.

$ git ls-remote
fatal: Could not read from remote repository.
```

6. Same repository, same second, same broken remote. The local command succeeded. The two remote commands failed instantly and cleanly.

37.2.4 PLAIN - what is really happening inside

1. Let us be exact about what each system keeps on your disk.
2. A Subversion working copy has a `.svn` directory. Historically one per folder, and since Subversion 1.7 in 2011, a single one at the top.
3. Inside it is a record of which revision you checked out, and a **pristine copy** of each file at that revision, so that `svn diff` and `svn revert` can work without the server.
4. That is a genuine and often-forgotten point: Subversion is not fully online. Local diff and local revert work offline, because of those pristine copies.
5. What it does not have is any other revision. It cannot show you revision 900 if you checked out revision 1000. It has two states: yours and the one you checked out.
6. A git repository has a `.git` directory containing every object ever created in that project's history, and the pointers that organize them.
7. So `git log` is a walk over local files. `git diff HEAD~50` reconstructs a fifty-commit-old state from local files. `git checkout` of a two-year-old tag reads local files.
8. Here is the structural difference, drawn.

```
CENTRALIZED (Subversion, CVS, Perforce)

  [ server ]  <- the history lives here, one copy
      |
      | network required to commit
      |
  [ laptop ]  working copy = one revision + pristine files


DISTRIBUTED (Git, Mercurial)

  [ server ]  a repository, by agreement only
      |
      | network required only to sync
      |
  [ laptop ]  a repository = ALL commits, ALL versions
```

9. In the top picture, cutting the line stops work. In the bottom picture, cutting the line stops sharing.

37.2.5 TECHNICAL - the engineer's version

1. The formal distinction: in a centralized VCS the repository is a single authoritative store, usually addressed by revision numbers that are globally ordered. In a distributed VCS every clone holds the full object database, and revisions are named by content hash because no global ordering exists.
2. That last clause is the deep reason git uses SHA-1 hashes rather than numbers like `r4213`. With no central authority, nobody can hand out sequential numbers, so identity must be computed from content instead.
3. Subversion can number commits 1, 2, 3 because exactly one process assigns them. Git cannot, so it names a commit `e83c5163316f89bfbde7d9ab23ca2e25604af290`.

Property	Centralized	Distributed
History location	server only	every clone
Commit needs network	yes	no
Revision names	integers	content hashes
Branch cost	server-side copy	one 41-byte file
Offline log or diff	no	yes
Per-file locking	supported	not supported

4. Real timings, measured in a sandbox on the git project's own repository, 85,263 commits and 4,845 tracked files, git 2.43.0 on Linux.

Command	Kind	Time
git rev-parse HEAD	local	4 ms
git branch -a	local	3 ms
git log –oneline -1000	local	19 ms
git status	local	228 ms
git ls-remote origin	remote	751 ms
git fetch origin (no-op)	remote	456 ms

5. Read that table carefully. The local commands are not merely faster. They are in a different class, because their cost has no round-trip term in it.
6. And the sandbox had an excellent link to github.com. The reader, on home broadband in India reaching a Microsoft edge in Delhi, would see far higher remote numbers on a good day, and unbounded numbers on the bad day this book keeps returning to.
7. The distributed model is not new with git. GNU Arch appeared in 2001, Monotone in 2003, and Darcs in 2003. Git and Mercurial, both from April 2005, are the two that survived at scale.
8. Perforce, founded by Christopher Seiwald in 1995 and now sold as P4, formerly Helix Core, is centralized by design and remains widely used. 37.10 explains why that is a reasoned choice and not a mistake.

37.2.6 WORDS - remember these

1. Centralized VCS — the history lives on one server — a system with a single authoritative repository that all clients must contact to commit.
2. Distributed VCS — everybody has the whole history — a system where every clone contains the complete object database and refs.
3. Working copy — the files as they are right now — a checkout of one revision plus local bookkeeping, without the full history.
4. Repository — the store of all history — the object database plus refs, in git the contents of the `.git` directory.
5. Clone — your own full copy — a complete repository created from another, including all reachable objects, plus a remote entry pointing back.
6. Round trip — one there-and-back over the network — the request and response latency that every remote operation must pay at least once.

37.3 Why Git was made

37.3.1 PLAIN - in simple words

1. Git was not designed in a calm room over two years. It was written in a hurry, in April 2005, because something was taken away.
2. From 2002 the Linux kernel was developed using **BitKeeper**, a commercial distributed version control system made by a company called BitMover, run by Larry McVoy.
3. BitKeeper was not free software, but McVoy gave the kernel community free use of it. Some kernel developers objected loudly to depending on a proprietary tool. It was used anyway, and it worked well.
4. In April 2005 that arrangement ended. BitMover announced it would stop providing the free version to the community. The official cutoff was 1 July

2005.

5. So the largest and busiest software project in the world suddenly had no version control system, and about three months to find one.
6. Nothing available was good enough. CVS and Subversion were centralized and far too slow for the kernel's volume. The existing distributed tools could not handle the size.
7. So Linus Torvalds wrote a new one. He started on 3 April 2005. He announced it on 6 April. On 7 April it was storing its own source code in itself.
8. That first commit still exists and you can read it today.
9. Then he handed it over. Junio Hamano became the maintainer on 26 July 2005, and has led the project ever since. Version 1.0 was released on 21 December

2005.

37.3.2 PLAIN - a picture in your head

1. Imagine a factory of ten thousand workers that runs on one specialized machine, borrowed free from a neighbour.
2. One morning the neighbour says the machine goes back in ninety days.
3. Nothing on the market fits. The machines you can buy are built for factories of thirty people, and they jam when a thousand parts arrive at once.
4. So the foreman goes into the workshop and builds a replacement himself, from the simplest possible parts, choosing crude and fast over clever and fragile, because the deadline is real.
5. Four days later it is running. Not finished. Running, and holding its own blueprints.
6. That is what happened, and it explains git's personality. The parts are very simple. The joins are visible. The commands were designed for the machine first and the human second.

Where this comparison breaks:

7. Git in 2026 is not the tool of April 2005. Thousands of contributors have worked on it for twenty-one years. The friendly commands, `git switch` and `git restore`, arrived in version 2.23 on 16 August 2019.
8. And the "ten days" story is often told as though one person wrote all of git in ten days. He did not. He wrote a working core in days and led it for under four months. What you type today is mostly other people's work.

37.3.3 PLAIN - a worked example

1. Here is the first commit of git, read out of a real clone of the git project's own repository, in a sandbox, today.

```
$ git log --reverse --max-parents=0 --format='%H%n%an <%ae>%n%ad%n%s'

e83c5163316f89bfbde7d9ab23ca2e25604af290
Linus Torvalds <torvalds@ppc970.osdl.org>
Thu Apr 7 15:13:13 2005 -0700
Initial revision of "git", the information manager from hell
```

2. Every fact in that block is checkable by anyone with git installed and a network connection.
3. The date confirms the timeline: four days after he started, git was already the thing storing git.
4. The email domain `osdl.org` is the Open Source Development Labs, which employed Torvalds at the time and later merged into the Linux Foundation in

2007.

5. The machine name `ppc970` is a PowerPC 970 processor, which is what was inside an Apple Power Mac G5 of that era.
6. The commit message is a joke about how unfriendly the tool was. It was accurate.

37.3.4 PLAIN - what is really happening inside

1. Torvalds stated his goals for the new system. They are worth taking one at a time, because each one shows up in the design you use today.
2. **Speed.** He wanted patching to take no more than three seconds on the kernel. This is why git stores things in a way that is cheap to read and why almost everything is local.
3. **Simple design.** The core is a content-addressed store: a key-value database where the key is the hash of the value. Chapter 38 covers it. The simplicity is why the format has survived twenty-one years unchanged.
4. **Strong support for non-linear development.** Thousands of parallel branches. In git a branch is a file containing a hash. Creating one costs 41 bytes and no network. This is why git users branch casually.
5. **Fully distributed.** Every clone complete. This is the property this whole chapter is about.
6. **Able to handle large projects efficiently**, in both speed and data size. The kernel was already enormous in 2005.
7. He added a further requirement that is often left out: **very strong safeguards against corruption, accidental or malicious**. This is why every object is named by the hash of its own content, and why a commit's hash covers its parent's hash, so tampering with old history changes every hash after it.
8. And one more thing shaped it: the kernel has thousands of contributors and no central authority who may write to everyone's tree. The design had to assume that trust is granted person to person, not by a server.

37.3.5 TECHNICAL - the engineer's version

1. Verified timeline of April to December 2005.

Date	Event
4 May 2000	BitKeeper first public release
2002	Kernel adopts BitKeeper
April 2005	BitMover ends free version
3 April 2005	Git development begins
6 April 2005	Git announced on the list
7 April 2005	Git self-hosting, commit e83c516
19 April 2005	Mercurial announced
26 July 2005	Junio Hamano becomes maintainer
21 December 2005	Git 1.0.0 released
9 May 2016	BitKeeper open-sourced

2. The trigger was a dispute over reverse engineering. Larry McVoy alleged that Andrew Tridgell, the author of Samba and rsync, had reverse engineered the BitKeeper protocol to build a tool called SourcePuller. BitMover withdrew the free community licence in response.
3. BitKeeper itself was eventually released as open source on 9 May 2016, under the Apache License version 2, eleven years too late to matter.
4. **Mercurial** was announced by Olivia Mackall on 19 April 2005, for the same reason and in the same month. Both projects were candidates to replace BitKeeper for the kernel. Git won.
5. Git's version history, taken from tags in the real repository, gives useful anchors for what is old and what is new.

Version	Released	Notable
v1.0.0	21 Dec 2005	first stable
v2.19.0	10 Sep 2018	partial clone
v2.23.0	16 Aug 2019	switch, restore

Version	Released	Notable
v2.29.0	19 Oct 2020	SHA-256 experimental
v2.43.0	20 Nov 2023	used in this chapter
v2.55.0	29 Jun 2026	current at writing

6. The object format has been stable since 2005. A repository created with git 1.0 in December 2005 can be read by git 2.55 in 2026 without conversion. That is a rare property and it was deliberate.
7. Git remains a C project with a small core and a very large surface. The original design is described in the `Documentation/technical` directory of the source, which ships with every clone.
8. The first Linux kernel release managed entirely with git was 2.6.12 in June 2005, about two months after git's first commit.

37.3.6 WORDS - remember these

1. BitKeeper — the commercial tool the kernel used before git — a proprietary distributed VCS by BitMover whose free community licence ended in April 2005.
2. Self-hosting — a tool storing its own source in itself — the point at which a system is complete enough to manage its own development.
3. Non-linear development — many people working on many branches at once — a commit graph that is a directed acyclic graph rather than a straight line.
4. Content-addressed store — you look things up by what they are, not where they are — a key-value store where the key is a cryptographic hash of the value.
5. Maintainer — the person who decides what goes in — the individual holding the canonical tree and applying or rejecting contributions.
6. Directed acyclic graph — a web of points with one-way arrows and no loops — the DAG formed by commits and their parent pointers.

37.4 What is actually inside a .git directory

37.4.1 PLAIN - in simple words

1. When you run `git init`, git makes one hidden directory called `.git` and puts a handful of files in it.
2. That directory is the repository. Everything else you see is just your files as they currently are.
3. Delete `.git` and you have a plain folder with no history. Copy `.git` somewhere else and you have carried the entire project's past with you.
4. There is no database server. No background process. No registry entry. No cloud account. Files in a directory.
5. This is worth pausing on, because it is the reason for everything in this chapter. If a thing is a local file, reading it does not need a network.
6. Here is what a real `.git` looks like immediately after `git init` and one commit, listed in a sandbox with git 2.43.0.

```
$ ls -a .git
COMMIT_EDITMSG   description   index    objects
HEAD             hooks         info     refs
branches         config        logs
```

7. Eleven entries. That is a complete version control system.

37.4.2 PLAIN - a picture in your head

1. Think of `.git` as a **warehouse with a whiteboard by the door**.
2. `objects` is the warehouse itself. Every crate ever received is in there, stored by a label computed from its contents. Nothing is ever moved or edited, only added.

3. `refs` is the whiteboard. It has a short list of names, and next to each name a crate label. "main: crate 9902ff97". "v0.1: crate b398d1b0".
4. `HEAD` is a sticky note on the whiteboard saying which of those names you are currently standing at.
5. `index` is the loading bay: things you have carried out of the aisles and set down, ready to be made into the next crate, but not yet crated.
6. `config` is the notice board with local rules on it, including the phone numbers of other warehouses you exchange crates with.
7. `logs` is the security camera footage of the whiteboard: every time a name was pointed at a different crate, and when.
8. `hooks` are the little printed cards saying "before accepting a delivery, check this". By default they are all examples and none are active.

Where this comparison breaks:

9. In a real warehouse, taking a crate out removes it. In `objects` nothing is removed by reading, and almost nothing is ever removed at all until a garbage collection runs, which needs no network either.
10. And a real warehouse can run out of aisles. Git periodically rewrites thousands of loose crates into one enormous compressed container, called a pack. The contents are identical, the storage is a different shape.

37.4.3 PLAIN - a worked example

1. Here is a genuine `.git` from a repository created in a sandbox, one file committed, printed in full. Nothing is edited or shortened.

```
$ cat .git/HEAD
ref: refs/heads/main

$ cat .git/refs/heads/main
9902ff97b5e8ac502e8ac752d950f15257c19335

$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[user]
    name = Reader
    email = reader@example.com
```

2. Read `HEAD` first. It contains the text `ref: refs/heads/main`. It does not contain a commit. It contains the *name* of a place that contains a commit.
3. Now read that place. `.git/refs/heads/main` contains 40 hexadecimal characters and a newline. 41 bytes. That is a branch.
4. A branch in git is a 41-byte file whose contents are a commit's name. That is the entire implementation. There is nothing else.
5. This is why branching is instant and free, and why it needs no network. Creating a branch writes 41 bytes to your own disk.
6. And `config` is an INI-style text file you can open in any editor. There is no binary blob, no encrypted store, no service account.
7. Here is what `git init` prints, verbatim, so you can see how little happens.

```
$ git init initdemo
Initialized empty Git repository in /tmp/gitdemo/initdemo/.git/
```

8. One directory made. No server contacted. It works on a plane.

37.4.4 PLAIN - what is really happening inside

1. Now every entry, one at a time, with what it is for.
2. **objects/** — the object database. Every commit, every directory listing, every file version, stored under its own hash. Chapter 38 is entirely about this. It is where nearly all the bytes are.
3. **refs/** — human names for commits. `refs/heads/` holds your branches, `refs/tags/` holds tags, `refs/remotes/` holds remote-tracking refs, which are section 37.7 and the heart of the reader's question.
4. **HEAD** — where you are now. Normally a symbolic reference: text saying `ref: refs/heads/main`. If it holds a raw hash instead, you are in the state git calls detached HEAD.
5. **config** — settings for this repository only. Your identity, your remotes, and any per-repository preferences. Text, editable, version 0 format.
6. **index** — the staging area. A binary file listing exactly what will go into the next commit, with each path, its mode, its blob hash and cached file metadata so `git status` can skip unchanged files. Chapter 39 covers it.
7. **hooks/** — scripts git will run at certain moments. On a fresh repository every one is a `.sample` file, and a `.sample` file never runs. Nothing is active until you rename one and make it executable.
8. **logs/** — the reflog. A line-per-change record of every value each ref has held, with who and when. This is your safety net after a bad reset.
9. **info/** — extra bits. `info/exclude` is a private ignore list that is not committed and not shared, unlike `.gitignore`.
10. **description** — used only by the old `gitweb` viewer. On a normal repository it is dead weight and you can ignore it.
11. **COMMIT_EDITMSG** — a scratch file holding the message you typed last. Not part of the history. Purely a convenience.
12. **branches/** — an obsolete shorthand directory from very early git. It is still created for compatibility and is always empty. Ignore it.
13. **packed-refs** — appears once refs are compressed into one file. See below.
14. The honest version: the exact set of files is an **implementation detail**, not a standard. It has changed over time and can differ with configuration. What is guaranteed is the on-disk *object format* and the general layout.

37.4.5 TECHNICAL - the engineer's version

1. Real `du` output for the `.git` of a full clone of the git project itself, git 2.43.0, cloned in a sandbox on 13 August 2026.

Entry	Size	Purpose
objects	316 MiB	all history
index	452 KiB	staged tree, 4845 files
packed-refs	104 KiB	2025 refs in one file
hooks	68 KiB	14 inactive samples
logs	32 KiB	reflogs
refs	28 KiB	loose refs
config	4 KiB	settings
HEAD	4 KiB	current position

2. The object store is 99.9 percent of it. Everything else is bookkeeping.
3. **packed-refs**: once a repository has many refs, git writes them into one sorted text file rather than one file per ref, because thousands of tiny files are slow on every filesystem. Real head of that file:

```
# pack-refs with: peeled fully-peeled sorted
165e5ad3169d0fd26637da3383a4514f1a9d1e72 refs/remotes/origin/bisect
426ec014a6630fb78c5a7334461bc33568f07743 refs/remotes/origin/jch
e9019fcafe0040228b8631c30f97ae1adb61bcdc refs/remotes/origin/maint
745601a9a94110d74769ab605ccd4f61339758d2 refs/remotes/origin/master
```

4. A loose ref file always wins over a `packed-refs` entry for the same name. That is the lookup rule.
5. **index** format: a binary file beginning with the four bytes `DIRC`, short for "dircache", then a version number and an entry count.

```
$ head -c 12 .git/index | od -c
0000000   D   I   R   C  \0  \0  \0 002  \0  \0  \0 002
```

6. Those twelve bytes read: signature `DIRC`, version 2, two entries. Index versions 2, 3 and 4 exist; version 4 adds path prefix compression.
7. `git ls-files --stage` decodes it for you:

```
100644 bd9f1115522d0cca0ac0e7b22f7d256e08287438 0   README.md
100644 31489c9ee8c455a4964beabcf6dd91ca1cea9ad4 0   src/main.c
```

8. Columns: file mode, blob hash, merge stage number, path. Stage 0 means no conflict. Stages 1, 2 and 3 appear during a conflicted merge and are Chapter 39.
9. The fourteen sample hooks in git 2.43.0 are `applypatch-msg`, `commit-msg`, `fsmonitor-watchman`, `post-update`, `pre-applypatch`, `pre-commit`, `pre-merge-commit`, `pre-push`, `pre-rebase`, `pre-receive`, `prepare-commit-msg`, `push-to-checkout`, `sendemail-validate`, `update`.
10. Useful inspection commands: `git rev-parse --git-dir`, `git count-objects -vH`, `git for-each-ref`, `git cat-file -p HEAD`, `git ls-files --stage`, `git symbolic-ref HEAD`, `git config --list --show-origin --show-scope`.
11. A **bare** repository, made with `git init --bare`, has these same entries at the top level with no working tree and no `index`. That is what a server stores. It is the same data with the checkout removed.

37.4.6 WORDS - remember these

1. Object database — the store of everything ever committed — the `.git/objects` directory holding loose and packed objects keyed by hash.
2. Ref — a name that points at a commit — a file under `.git/refs` or an entry in `packed-refs` containing a 40-character object ID.
3. HEAD — where you are standing now — a symbolic ref, usually `ref: refs/heads/<branch>`, or a raw object ID when detached.
4. Index — the list of what goes in the next commit — a binary `DIRC` file of staged path, mode, blob ID and cached stat data.
5. Reflog — the record of where each name used to point — per-ref append-only logs under `.git/logs`, used to recover lost commits.
6. Bare repository — a repository with no working files — a `.git` layout at the top level, with `core.bare` true and no index or checkout.
7. Hook — a script git runs at a set moment — an executable in `.git/hooks` with an exact name, disabled by default and never cloned.

37.5 What "the whole history" really means

37.5.1 PLAIN - in simple words

1. When we say a clone contains the whole history, that phrase is doing a lot of work. Let us make it exact.
2. It means every commit ever made on any branch that was fetched.
3. It means every version of every file, including files that were deleted years ago and no longer exist in your folder.
4. It means every commit message, every author name, every date, every parent link, and every tag.
5. All of it, on your disk, right now, readable with no network.
6. The obvious worry is size. If a project has 85,000 commits, and each one is a snapshot of the whole project, surely the disk fills up?
7. It does not, and the reason is that source code is text, text is enormously repetitive, and git compresses it very hard.
8. Two mechanisms do the work. First, identical content is stored exactly once, because content with the same bytes gets the same name. If a file did not change between two commits, both commits point at the same stored copy.
9. Second, git periodically rewrites the store into a **pack**, which compresses everything and stores similar files as small differences from each other.
10. The result is that the whole past often costs only a few times the size of the present.

37.5.2 PLAIN - a picture in your head

1. Imagine photographing a whiteboard every day for five years. That is 1,800 photographs.
2. Stored naively, 1,800 full-resolution photographs is a lot of disk.
3. But most days almost nothing changes. So instead you keep one full photograph and, for each other day, a short note: “same as yesterday except the top-right box now says 42”.
4. Now 1,800 days of history costs slightly more than one photograph plus 1,799 short notes.
5. That is a pack file. Git finds files that resemble each other, picks one to store whole, and stores the others as instructions relative to it.
6. And it compresses everything on top of that, the way a zip file does.

Where this comparison breaks:

7. Git does not compare “yesterday to today” the way the photo story suggests. It compares any object to any similar object, regardless of date, using a heuristic based on file name and size.
8. Also a chain of differences cannot be infinitely long, or reading the newest version would require replaying thousands of steps. Git limits chain depth, by default to 50, so reads stay fast.
9. And binary files break the trick. A photograph, a compiled program or a video changes almost entirely with every edit and compresses badly. This is a real limitation and 37.10 returns to it.

37.5.3 PLAIN - a worked example

1. These are real measurements from a full clone of the git project’s own source repository, made in a sandbox on 13 August 2026.

Measure	Value
Commits in history	85,263
Files in working tree	4,845
Working tree on disk	60 MiB
.git directory	316 MiB
Objects stored	418,900
Pack files	1

2. So twenty-one years and 85,263 commits cost about 5.3 times the size of the current checkout. That is the answer to "surely it fills the disk".
3. Now the compression, measured object by object with `git cat-file --batch-all-objects`.

Object type	Count	Raw / On disk
blob (file data)	165,095	5364 / 194 MiB
tree (directories)	167,536	2453 / 74 MiB
commit	85,263	59 / 33 MiB
tag	1,006	0.7 / 0.6 MiB

4. Add them up: about 7,877 MiB of raw object content stored in about 302 MiB on disk. A ratio of roughly 26 to 1.
5. File data compresses best, 5,364 MiB down to 194 MiB, about 27 to 1, because most versions of most files are nearly identical to another version.
6. Commits compress worst, 59 MiB down to 33 MiB, about 1.8 to 1, because every commit contains different hashes, and hashes are random-looking text that does not compress.
7. That last line is a nice piece of evidence: compression works on repetition, and hashes have none by design.

37.5.4 PLAIN - what is really happening inside

1. Two storage shapes exist and both are always valid.
2. **Loose objects.** One file per object, at `.git/objects/ab/cdef...`, where `ab` is the first two hex characters of the hash and the rest is the file name. The content is compressed with zlib.
3. The two-character directory split exists because filesystems slow down badly with hundreds of thousands of files in one directory. It spreads them over 256 directories.
4. **Packed objects.** Many objects in one `.pack` file, with a `.idx` file beside it giving the byte offset of each object so lookup stays fast.
5. Inside a pack, an object may be stored whole, or as a **delta** against another object in the same pack. The delta is a tiny instruction list: copy these bytes from the base, then insert these new bytes.
6. Packing happens automatically. `git gc` runs when loose objects accumulate, and a fresh `git clone` always arrives as a pack because the sending side builds one for the transfer.
7. That is why a freshly cloned repository shows `count: 0` loose objects and one large pack.
8. Nothing about this needs a network. Packing, unpacking, delta resolution and garbage collection are all local CPU and disk work.

37.5.5 TECHNICAL - the engineer's version

1. Object naming: git 1.0 through the present names objects by SHA-1 of the header plus content. SHA-256 repositories became available as an experimental feature in git 2.29, released 19 October 2020, and interoperability with SHA-1 repositories is still incomplete as of git 2.55 in June 2026.
2. Default packing parameters, from `git config` documentation for git 2.43: `pack.depth` 50, `pack.window` 10, `core.compression` -1 meaning zlib default level 6, `gc.auto` 6700 loose objects.
3. Loose object path scheme: `.git/objects/<first 2 hex>/<remaining 38 hex>`. Real listing from a sandbox repository:

```
.git/objects/b1/a8becadd478624373132f066d206859ade455f
.git/objects/fb/560ef022402ab49df96a3b868aa7c540be1a57
.git/objects/e6/69826a4efde29351886aa6e636e14635a29827
.git/objects/c3/5f063d2751d7340c2cdd4fe0046a137c9164c4
```

4. **Shallow clone**: `git clone --depth N` fetches only the last N commits per branch. The repository records a `.git/shallow` file listing the commits whose parents are deliberately absent.

5. **Partial clone**: `git clone --filter=blob:none` fetches all commits and trees but no file contents, downloading blobs on demand. Introduced in git 2.19, 10 September 2018. It sets `remote.origin.promisor = true` and `remote.origin.partialclonefilter` in the config.
6. Real measurements of the same repository, four ways, same sandbox, same day.

Clone type	.git size	Clone time
full	316 MiB	34 s
blobless filter	130 MiB	38 s
treeless filter	51 MiB	25 s
shallow depth 1	14 MiB	4.6 s

7. What you lose, stated precisely.
8. Shallow: `git log` shows one commit, not 85,263. `git blame` cannot see past the cut. `git bisect` cannot search history you do not have. Pushing from a shallow clone works but has restrictions, and deepening requires the network.
9. Blobless: history commands work, but any command needing file *content* from an old commit, such as `git blame` or `git log -p`, silently triggers a network fetch. Offline, those commands now fail.
10. That is the sting. A partial clone quietly reintroduces the network into operations that were local. During an outage, a blobless clone behaves less like git and more like Subversion for exactly those commands.
11. Real config written by a blobless clone:

```
[remote "origin"]
    url = https://github.com/git/git.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    promisor = true
    partialclonefilter = blob:none
```

12. Guidance: use shallow clones for continuous integration runners that build once and are destroyed. Use full clones for machines where people work, especially machines that go offline.

37.5.6 WORDS - remember these

1. Blob — the stored contents of one file version — a git object holding raw file bytes with no name and no permissions.
2. Pack file — one big compressed container of many objects — a `.pack` with a companion `.idx` giving offsets, using delta chains internally.
3. Delta — a small "difference from another object" — copy and insert instructions against a base object within the same pack.
4. zlib — the common compression used inside git — DEFLATE compression, RFC 1950 and 1951, applied to every loose object and pack entry.
5. Shallow clone — a copy with only the newest commits — a clone with truncated history recorded in `.git/shallow`, created by `--depth`.
6. Partial clone — a copy that fetches file contents on demand — a promisor clone created with `--filter`, requiring the remote for missing objects.
7. Garbage collection — tidying the store — `git gc`, which packs loose objects and prunes unreachable ones after a grace period, default 14 days.

37.6 What "origin" actually is

37.6.1 PLAIN - in simple words

1. Almost every git user believes something slightly wrong about `origin`, so let us be blunt.
2. `origin` is a nickname for a URL, stored in a text file on your own computer.
3. That is the entire definition. There is nothing else to it.
4. It is not the master copy. It is not the real repository. It is not the authority. It does not own your work.
5. The name `origin` is not special to git. It is the name `git clone` happens to use by default. You may rename it to anything.
6. That default is a **convention**, not a **standard**. No specification requires it. It is a habit that became universal.
7. A repository can have zero remotes, one remote, or twenty remotes. All of them are equal. None outranks another.
8. Zero is not a broken state. A repository with no remotes is a completely normal, fully functional git repository. You can commit to it forever.
9. And the URL a remote points at does not have to be on the internet. It can be another folder on the same disk, or a USB stick.

37.6.2 PLAIN - a picture in your head

1. `origin` is an entry in your phone's contact list.
2. The contact is called "Mum". That name means nothing to the phone network. It is a label you chose, stored on your handset, mapping to a number.
3. Rename the contact to "Mother" and nothing anywhere else changes. The phone number still works. Your mother does not notice.
4. Delete the contact entirely and your phone still works perfectly. You simply have to type the number when you want to call.
5. You may have twenty contacts. None of them is in charge of your phone.
6. And your mother has her own contact list, in which you are an entry. Neither list is the true one.
7. That is `origin` exactly. A label on your machine mapping a short name to an address you would otherwise have to type in full.

Where this comparison breaks:

8. A phone contact is only a convenience. A git remote also carries a **refspec**, a rule saying which of their branches map to which of your remote-tracking names. That is real behaviour, not just a label.
9. And in a team, `origin` usually does have authority, because the team agreed it does and the hosting service enforces that agreement. That authority is social and administrative. It is not in the git data model.

37.6.3 PLAIN - a worked example

1. Here is a real `.git/config` immediately after a clone, printed verbatim from a sandbox.

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = /tmp/gitdemo/server.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
    remote = origin
    merge = refs/heads/main
```

2. Look at the `[remote "origin"]` block. Two lines. A URL and a fetch rule. That is what `origin` is made of.
3. Now rename it and watch what happens, and what does not.

```
$ git remote rename origin upstream
$ git config --get-regexp '^remote\.|^branch\.'
remote.upstream.url /tmp/gitdemo/server.git
remote.upstream.fetch +refs/heads/*:refs/remotes/upstream/*
branch.main.remote upstream
branch.feature/offline-work.remote upstream
```

4. Git rewrote the config, renamed the ref directory from `refs/remotes/origin/` to `refs/remotes/upstream/`, and updated every branch that referred to it.
5. No commit changed. No hash changed. The server was never contacted. It was a local text edit plus a directory rename.
6. Now remove every remote and keep working.

```
$ git remote remove origin
$ git remote
                        (empty: no remotes at all)
$ git commit -m "Commit in a repo with no remotes at all"
$ git log --oneline -1
4e2462c Commit in a repo with no remotes at all
```

7. Zero remotes. Commit succeeded. That is the proof that `origin` is optional.

37.6.4 PLAIN – what is really happening inside

1. A remote is three things stored together in `.git/config`.
2. **A name.** The text between the quotes in `[remote "origin"]`. Arbitrary.
3. **A URL.** Where to reach it. This may be `https://...`, `git@host:path` for SSH, `ssh://...`, `git://...`, or a plain filesystem path.
4. **A refspec.** The line `fetch = +refs/heads/*:refs/remotes/origin/*` reads: take everything under `refs/heads/` on their side, and store it under `refs/remotes/origin/` on mine. The leading `+` means overwrite even if the change is not a fast-forward.
5. Optionally a **pushurl**, if you fetch from one place and push to another.
6. Separately, each branch may record an **upstream**: the `[branch "main"]` block saying which remote and which branch it pairs with. This is what lets you type `git push` with no arguments.
7. When you run a remote command, git looks up the name, finds the URL, chooses a transport based on the URL scheme, and connects. Chapter 40 covers the transports and their ports.
8. Everything before the connect step is reading a text file.
9. The honest version: people say "push to origin" as though pushing were sending to a superior. It is not. It is one repository asking a peer to update some of its refs, and the peer may refuse. Chapter 40 covers refusal.

37.6.5 TECHNICAL – the engineer's version

1. Config precedence, highest wins: command line, then `.git/config`, then `~/.gitconfig` or `~/.config/git/config`, then `/etc/gitconfig`. Verify with `git config --list --show-scope --show-origin`.

```
system  file:/etc/gitconfig filter.lfs.required=true
global  file:/root/.gitconfig   user.name=Shikhar
global  file:/root/.gitconfig   user.email=shikhar@example.com
local   file:.git/config    remote.origin.url=...
```

2. Remote management commands, all of which are pure local config edits except the last two.

Command	Touches network
git remote -v	no
git remote add	no
git remote rename	no
git remote remove	no
git remote set-url	no
git remote show origin	yes
git remote update	yes

3. `git remote show origin` contacts the server, which surprises people. Use `git remote show -n origin` to stay offline and read only cached data.
4. A repository may hold many remotes and they are peers in the data model. The common three-remote fork workflow is:

```
$ git remote -v
mirror  /tmp/gitdemo/mirror.git (fetch)
mirror  /tmp/gitdemo/mirror.git (push)
origin  /tmp/gitdemo/server.git (fetch)
origin  /tmp/gitdemo/server.git (push)
```

5. Naming conventions widely used but not required by git: `origin` for the place you cloned from, `upstream` for the original project when `origin` is your fork, `fork` for a personal copy. All are **convention**.
6. The default remote name is set by `clone.defaultRemoteName`, added in git 2.30, January 2021. Setting it to `upstream` makes `git clone` name it that.
7. A remote URL of a local path is fully supported and often useful: a USB drive, a network mount, or a bare repository elsewhere on the same disk. The sandbox examples in this chapter use exactly that.
8. There is no field anywhere in git's data model marking one repository as canonical. Authority is entirely external: server permissions, branch protection rules and team agreement.

37.6.6 WORDS - remember these

1. Remote — a saved nickname for another repository's address — a named config section holding a URL and one or more refspecs.
2. origin — the default nickname `git clone` gives the place you cloned from — a conventional remote name with no special meaning to git.
3. Refspec — the rule mapping their branch names to yours — a `+src:dst` pattern, typically `+refs/heads/*:refs/remotes/origin/*`.
4. Upstream — the branch yours is paired with — the `branch.<name>.remote` and `branch.<name>.merge` config pair, shown as `@{upstream}`.
5. Bare repository — a repository with no checked-out files — what a server holds, so nobody's working tree is disturbed by a push.
6. Peer — an equal repository, not a superior — the correct mental model for every remote in a distributed VCS.

37.7 Remote-tracking refs and staleness

37.7.1 PLAIN - in simple words

1. This section is the reader's own case, so we will be very careful.
2. During the outage, their machine reported `origin/main` as `0a95cc8`.
3. It is natural to read that as "the server's main branch is at 0a95cc8 right now". That reading is wrong.
4. The correct reading is: "the last time this computer successfully talked to that server, its main branch was at 0a95cc8".

5. `origin/main` is a **memory**, not a **measurement**. It is a note your machine wrote to itself, at some past moment, and has not revised since.
6. It lives in a file on your disk. Nothing about it is live. Git will keep reporting that value cheerfully, forever, with the network cable cut.
7. And git will not warn you. There is no "possibly stale" marker in the normal output. It looks exactly the same when it is one second old and when it is six weeks old.
8. This is not a bug. It is the only thing a distributed system can do. Asking the server on every command would be exactly the centralized design git was built to escape.
9. So git gives you a fast, honest, local answer to a slightly different question than the one you thought you asked.

37.7.2 PLAIN - a picture in your head

1. You keep a note on your fridge that says: "Priya's flight lands at 18:40."
2. You wrote it on Tuesday, after speaking to Priya.
3. On Friday you look at the fridge and read 18:40. The note has not changed.
4. But the note is not the flight. It is what you were told on Tuesday. The flight may have moved to 21:10 and nobody has told the fridge.
5. The fridge cannot be wrong, because it is not claiming to know the flight. It is claiming to record what you wrote. It does that perfectly.
6. To learn the real time you must call the airline. That is `git fetch`.
7. If the phone line is down, you can still read the note. You just cannot refresh it. And the note gives you no hint of its own age.

Where this comparison breaks:

8. Git does keep a dated record of when the note changed, in the reflog, so you can find out how old it is. The fridge does not. Section 37.7.5 shows how.
9. And unlike a flight time, the true value of the remote's `main` can only move forward under normal rules. A stale `origin/main` is usually an *underestimate* of what the server now has, not a random wrong value.

37.7.3 PLAIN - a worked example

1. Here is the whole effect, produced in a sandbox by making the remote unreachable, exactly as an outage would.
2. Before the outage: clone, one commit, pushed. Both sides agree.

```
$ git rev-parse main
01d088e5131d972ed904171845efe601d994fea5
$ git rev-parse origin/main
01d088e5131d972ed904171845efe601d994fea5
```

3. Now the remote becomes unreachable, and three commits are made locally.

```
$ git rev-parse main
2c9e7c6d79ab36d4a7ca7b229c750fc0ce5dda53
$ git rev-parse origin/main
01d088e5131d972ed904171845efe601d994fea5
```

4. `main` moved. `origin/main` did not, and cannot, because moving it requires hearing from the server.
5. Here is the file itself, which is the point of the whole section.

```
$ cat .git/refs/remotes/origin/main
01d088e5131d972ed904171845efe601d994fea5
```

6. Forty-one bytes on your own disk. That is `origin/main`. It is not a connection. It is not a query. It is a file.

7. The reader's `0a95cc8` was the first seven characters of exactly such a file's contents, on their own machine, written the last time a fetch or push had succeeded.
8. Git will happily use it offline:

```
$ git log --oneline -1 origin/main
01d088e Initial project layout
```

9. That command ran with the remote unreachable and printed a confident answer.

37.7.4 PLAIN - what is really happening inside

1. Now the exact rules. Which commands update a remote-tracking ref and which never do.
2. **`git fetch`** updates them. That is its entire job: connect, download new objects, and move `refs/remotes/<remote>/*` to match what the server said.
3. **`git pull`** updates them, because a pull is a fetch followed by a merge or rebase. The fetch part does the updating.
4. **`git push`** updates them, for the branches it pushed. If your push succeeds, git knows the server now holds what you sent, so it moves the matching remote-tracking ref locally. This is a small but important detail.
5. **`git remote update`** and `git fetch --all` update them for all remotes.
6. Nothing else does. Not `git commit`, not `git status`, not `git log`, not `git branch -vv`, not `git merge`, not `git rebase`, not `git checkout`.
7. That last group includes every command that displays "ahead 3, behind 2". Those numbers are computed by comparing two local refs. Both are local, so both can be stale.
8. So "your branch is behind origin/main by 2 commits" means "behind the value I last heard, which was some time ago". It is not a live comparison.
9. Here is the sequence in a diagram.

```
      your disk                    the server
---------------------          ---------------------
refs/heads/main                refs/heads/main
   moved by: commit,              moved by: pushes
   merge, rebase, reset           from anyone

refs/remotes/origin/main
   moved ONLY by:
     fetch, pull, push
   otherwise frozen  <-- the reader's 0a95cc8
```

10. The left column can change every minute while you work. The middle entry changes only when a network operation succeeds. During an outage it is frozen by definition.

37.7.5 TECHNICAL - the engineer's version

1. Storage: remote-tracking refs live at `.git/refs/remotes/<remote>/<branch>` when loose, or as `refs/remotes/<remote>/<branch>` lines in `.git/packed-refs` once packed. Both are read the same way; loose wins on conflict.
2. They are ordinary refs in a reserved namespace. Git forbids committing onto them, which is why `git checkout origin/main` puts you in detached HEAD rather than on a branch.
3. **How to tell how stale it is.** The reflog for a remote-tracking ref records every update with a timestamp. Real output:

```
$ git reflog show origin/main --date=iso
d39bbb7 refs/remotes/origin/main@{2026-08-13 03:12:21 +0000}: update by push
01d088e refs/remotes/origin/main@{2026-08-13 03:09:20 +0000}: update by push
```

4. The top line's timestamp is the last moment this machine had confirmed knowledge of that branch. Everything since is unverified.
5. Note the reason strings: `update by push` here, `fetch` or `fetch origin` after a fetch. So the reflog also tells you *how* you learned it.
6. Other ways to measure staleness:

Command	What it tells you
git reflog show origin/main	when it last moved
stat -f %Sm .git/refs/…	file mtime, macOS
stat -c %y .git/refs/…	file mtime, Linux
git ls-remote origin main	the live value, needs net

7. The file modification time is a decent proxy but imperfect: a fetch that finds no change may not rewrite the file, and packing rewrites timestamps. Prefer the reflog.
8. `git ls-remote origin main` is the only entry in that table that answers the question you actually wanted. It contacts the server and prints the live value without downloading any objects. It is the correct tool for "is my `origin/main` current", and it needs the network.
9. `git status` output such as `## main...origin/main [ahead 3]` is computed by `git rev-list --left-right --count main...origin/main`, entirely locally.
10. `git fetch --dry-run` still contacts the server. It only skips writing.
11. Configuration worth knowing: `fetch.prune` or `git fetch --prune` deletes remote-tracking refs whose upstream branch no longer exists. Without it, `refs/remotes/origin/` accumulates names of branches deleted months ago, which is another form of staleness.
12. The honest version: `origin/main` is sometimes described as "the remote branch". It is not the remote branch. It is a local cache of the remote branch, with no expiry, no freshness metadata in the ref itself, and no invalidation. The remote branch is on the remote.

37.7.6 WORDS - remember these

1. Remote-tracking ref — your machine's note of where their branch was — a ref under `refs/remotes/<remote>/` updated only by fetch, pull or push.
2. Stale — out of date without saying so — holding a cached value with no freshness check and no invalidation signal.
3. Fetch — go and get their latest, without changing my work — download objects and update `refs/remotes/*`, leaving `refs/heads/*` untouched.
4. Ahead and behind — how many commits each side has that the other lacks — counts from `git rev-list --left-right --count A...B`, computed locally.
5. Prune — forget remote branches that no longer exist — `git fetch --prune`, deleting stale `refs/remotes` entries.
6. ls-remote — ask the server what it has right now — a network query printing the remote's refs without transferring objects.

37.8 Which git operations need the network

37.8.1 PLAIN - in simple words

1. Here is the whole answer in one line: a git command needs the network if and only if it must read from, or write to, another repository.
2. Everything else is reading and writing files inside `.git`, and files do not need a network.
3. That rule is not a rough guide. It is exact, and you can apply it to a command you have never seen.
4. Ask yourself: does this command need to know something only another machine knows, or change something only another machine holds?

5. If yes, it is a network command. If no, it is local.
6. `git log` reads commits. All commits are local. Local command.
7. `git branch` writes a 41-byte file. Local command.
8. `git merge` combines two commits you already have. Local command.
9. `git fetch` needs objects you do not have. Network command.
10. `git push` must change refs on another machine. Network command.
11. There is no third category and no exceptions worth memorizing, apart from a couple of traps listed at the end.

37.8.2 PLAIN - a picture in your head

1. Think of your repository as a personal notebook and the remote as a friend's notebook in another city.
2. Anything you can do by turning pages in your own notebook is instant and always available. Reading old pages. Copying a page. Rewriting a draft.
3. Anything requiring your friend's notebook needs a phone call. Learning what they wrote. Dictating your pages to them.
4. Nothing about a broken phone stops you turning your own pages.
5. And notice: dictating a page to your friend needs a call, but *writing* the page did not. The writing already happened, in your own notebook, days ago.
6. That separation between writing and sending is the whole distributed idea.

Where this comparison breaks:

7. A few git commands look local but quietly place a call. `git remote show origin` does. `git submodule update` does. In a partial clone, `git log -p` does. These are the traps, and 37.8.5 lists them.

37.8.3 PLAIN - a worked example

1. Every one of these ran successfully in a sandbox with the remote made unreachable, so the outage was real, not simulated by a flag.

```
git branch -a          rc=0   * feature/offline-work
git tag v0.1           rc=0
git diff HEAD~1 --stat rc=0   README.md | 1 +
git log --oneline -3   rc=0   d39bbb7 Committed during the outage
git blame -L 1,2 file  rc=0   ^01d088e (Reader 2026-08-13 ...)
git reflog -3          rc=0   d39bbb7 HEAD@{0}: commit: ...
git switch main        rc=0
git merge feature/...  rc=0
git bisect start/good  rc=0   Bisecting: 2 revisions left to test
```

2. Branching, tagging, diffing, logging, blaming, reflogging, switching, merging and bisecting. All with no reachable remote.
3. And in the same session, at the same second, these failed:

```
$ git fetch
fatal: Could not read from remote repository.

$ git ls-remote
fatal: Could not read from remote repository.
```

4. The failures are immediate and specific. Git did not hang or half-work. It attempted a connection, failed, and told you.

37.8.4 PLAIN - what is really happening inside

1. The definitive classification. Local commands first.

Command	Local	What it touches
git init	yes	creates .git
git add	yes	index
git commit	yes	objects, refs, logs
git status	yes	index, working tree
git diff	yes	index, objects
git log	yes	objects
git branch	yes	refs
git checkout	yes	index, working tree
git switch	yes	index, working tree
git restore	yes	index, working tree
git merge	yes	objects, refs, index
git rebase	yes	objects, refs, index
git reset	yes	refs, index
git stash	yes	objects, refs
git tag	yes	refs, objects
git blame	yes	objects
git bisect	yes	objects, refs
git cherry-pick	yes	objects, refs
git reflog	yes	logs
git gc	yes	objects
git show	yes	objects
git worktree	yes	.git, filesystem

2. Now the network commands.

Command	Direction	Purpose
git clone	in	copy a whole repository
git fetch	in	download new objects, refs
git pull	in	fetch then merge or rebase
git push	out	send objects, move their refs
git ls-remote	in	read their refs only
git remote show	in	query their state
git submodule update	in	clone or fetch sub-repos
git request-pull	in	reads the remote to compare
git archive –remote	in	ask server to build a tarball

3. Why the line falls exactly there. Local commands read and write four things: the object database, the refs, the index, and your working files. All four are directories on your disk.
4. Network commands do exactly one extra thing: they move objects and ref values between two object databases over a transport.
5. That is why the split has no fuzzy middle. A git command is a manipulation of a local file store, or it is a synchronization between two file stores.
6. The commit itself is proof. `git commit` writes some objects, moves one ref, appends to one log file, and stops. It has no idea a remote exists.

37.8.5 TECHNICAL - the engineer's version

1. The traps, stated precisely, because each one has caught working engineers.
2. **`git pull` is not `git fetch` plus nothing.** It is fetch plus merge, or fetch plus rebase if `pull.rebase` is true. Offline it fails at the fetch and does not perform the merge.
3. **`git remote show origin` contacts the server.** `git remote show -n origin` does not, and prints cached data only.
4. **`git fetch --dry-run` contacts the server.** Dry run means it does not write refs, not that it does not connect.
5. **In a partial clone, previously local commands become remote.** With `--filter=blob:none`, commands needing historical file content, such as `git log -p`, `git blame` on old commits, or `git checkout` of an old tag, fetch blobs on demand and fail offline.
6. **Submodules.** `git submodule update --init` clones each submodule. Offline it fails, even though the parent repository's own history is complete.
7. **Git LFS.** Large File Storage replaces big files with pointer text and fetches the real bytes on checkout. A repository using LFS is not fully offline-capable for those files. LFS was released by GitHub in April 2015.
8. **Credential helpers and signing.** A commit signed with a hardware key or an SSH agent may need that device present, though not a network. A commit signed via a remote signing service does need one.
9. **Hooks.** A `pre-commit` hook that runs a linter which downloads packages turns a local command into a network one. That is your hook's doing, not git's.
10. Measured cost difference, real numbers from the sandbox on the git project repository, so you can see why the split matters for speed as well as availability.

Operation	Class	Time
git rev-parse HEAD	local	4 ms
git log –oneline (85k)	local	928 ms
git checkout HEAD~100 – .	local	119 ms
git ls-remote origin	remote	751 ms
git fetch origin (no change)	remote	456 ms

11. Walking 85,263 commits locally took 928 ms. Asking the server a single question over a fast link took 751 ms. On the reader's link that day, it would never have completed at all.
12. Exit code discipline: remote commands fail with a non-zero status and a `fatal:` message on stderr. Scripts should check status rather than parse text, because the messages are not a stable interface.

37.8.6 WORDS - remember these

1. Local operation — something git does entirely on your own disk — a command touching only the object database, refs, index and working tree.
2. Remote operation — something needing another repository — a command opening a transport connection to synchronize objects or refs.
3. Transport — the way two repositories talk — HTTPS, SSH, the git protocol on port 9418, or a local filesystem path.
4. Dry run — do the checking but not the writing — `--dry-run`, which for fetch and push still performs the network connection.
5. Submodule — another repository nested inside this one — a gitlink entry recording a commit ID, with the sub-repository cloned separately.
6. Git LFS — a system for keeping huge files out of history — Large File Storage, storing pointers in git and content on a separate server.

37.9 The reader's outage, worked through

37.9.1 PLAIN - in simple words

1. Now we put the whole chapter onto the reader's actual day.
2. They were in a flat in India, on home broadband, on a Mac. Their router was at `192.168.0.1` and their DNS resolver was `1.1.1.1`.
3. `github.com` resolved to `20.207.73.82`, an address in a Microsoft-owned range, because GitHub has been owned by Microsoft since 2018 and fronts traffic through Microsoft's network edge.
4. `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and then timed out after 15 seconds with no response at all. No refusal, no error message from any router. Silence.
5. The same site loaded instantly on mobile data on the same phone, which showed the site itself was fine and the path from that flat was not.
6. So: GitHub was reachable from the world and unreachable from that sofa.
7. And the reader kept working for the whole outage. They edited files, ran `git add`, ran `git commit`, created branches, committed again.
8. Every one of those commands succeeded. Not partially. Not queued for later. Fully succeeded, with real commits and real hashes.
9. The reason is now, we hope, boring: none of those commands involved another computer. They wrote files into `.git` on that Mac's own disk.
10. Later, connectivity returned. One `git push` sent everything that had piled up, in a single operation.

37.9.2 PLAIN - a picture in your head

1. Think of a writer in a cottage during a storm that has taken out the phone line.
2. They cannot post anything. They cannot receive anything. The postbox at the end of the lane may as well not exist.
3. But the desk still works. The pen still works. The filing cabinet still works. They write chapters four, five and six, date each one, and file them.
4. When the line comes back they walk to the postbox once and post all three chapters together.
5. Nothing was lost, nothing had to be redone, and nothing had to be remembered in their head rather than on paper.
6. The one real cost is that during the storm they did not know what their co-author had been writing. They found that out afterwards, and might then have had to reconcile.

Where this comparison breaks:

7. The writer knows the line is down. Git does not always make it obvious, because `git status` and `git log` behave identically offline and online. The reader's `origin/main` looked like a fact and was a memory.
8. And posting three chapters at once is not always safe in git. If the co-author moved `main` while you were offline, your push may be rejected, and you must fetch and reconcile. That is Chapter 40.

37.9.3 PLAIN - a worked example

1. The full sequence, with the reader's own values where known, reconstructed step by step.

```
TIME   EVENT                              origin/main   local main
-----  ---------------------------------  -----------   ----------
t0     last successful fetch or push      0a95cc8       0a95cc8
t1     network path to GitHub goes dark   0a95cc8       0a95cc8
t2     edit files, git add, git commit    0a95cc8       (new #1)
t3     git switch -c feature/...          0a95cc8       (new #1)
t4     more commits on the branch         0a95cc8       (new #1)
t5     git status: "ahead 1"  <- cached   0a95cc8       (new #1)
t6     git fetch -> fatal, no route       0a95cc8       (new #1)
```

```
t7     connectivity returns              0a95cc8       (new #1)
t8     git push origin main              (new #1)      (new #1)
t9     git push -u origin feature/...    (new #1)      (new #1)
```

2. The `origin/main` column is frozen from t0 to t8. That column is a file on the reader's disk, and only a successful network operation writes to it.
3. Here is the equivalent moment reproduced in a sandbox, with real output.

```
before push: origin/main = 01d088e5131d972ed904171845efe601d994fea5
before push: main        = d39bbb76dc18fd2151d333cbf82d9405449342f8
commits waiting to be sent: 5

$ git push origin main
To /tmp/gitdemo/server.git
   01d088e..d39bbb7  main -> main

after push:  origin/main = d39bbb76dc18fd2151d333cbf82d9405449342f8
```

4. Five commits, accumulated with no network, sent in one operation, and only then did `origin/main` move.
5. And the second branch, created entirely offline, published in one go:

```
$ git push -u origin feature/offline-work
To /tmp/gitdemo/server.git
 * [new branch]  feature/offline-work -> feature/offline-work
branch 'feature/offline-work' set up to track
'origin/feature/offline-work'.
```

37.9.4 PLAIN - what is really happening inside

1. Let us be precise about what each offline command wrote.
2. `git add README.md` read the file, computed its SHA-1, wrote a blob object into `.git/objects` if that content was new, and updated `.git/index`.
3. `git commit -m "..."` built a tree object from the index, wrote a commit object naming that tree and the previous commit as parent, moved `.git/refs/heads/main` to the new commit's hash, and appended two reflog lines.
4. Total external dependencies of that sequence: none. Total network packets: zero.
5. `git switch -c feature/offline-work` wrote one new 41-byte file at `.git/refs/heads/feature/offline-work` and rewrote `.git/HEAD` to point at it. Two small writes.
6. Now what was **not** possible, stated honestly, because this matters as much as what was.
7. They could not fetch anyone else's new work. If a colleague pushed during those hours, the reader's machine had no way to learn it and no way to include it.
8. They could not know whether anyone had pushed at all. `origin/main` said `0a95cc8` and would have said `0a95cc8` if fifty commits had landed.
9. They could not open a pull request, review one, comment, merge one, or read an issue. All of those are GitHub features reached over the network, not git features. Section 37.11 draws that line.
10. They could not see continuous integration results, because those live on GitHub's servers.
11. They could not verify that their branch would merge cleanly into the real current `main`, only into the `main` they last saw.
12. So the honest summary: **authoring** was fully available, and **coordination** was fully unavailable. Git splits those two cleanly, and that split is the single most useful thing to remember from this chapter.

37.9.5 TECHNICAL - the engineer's version

1. Evidence classification for that session, keeping proof separate from inference, as this book does everywhere.
2. **Proven by observation.** DNS resolution succeeded, returning `20.207.73.82`. A TCP connection to port 443 at that address received no response of any kind within 15 seconds. The same request over a different access network succeeded immediately.
3. **Also proven.** Local git commands returned exit status 0 and produced new objects and refs during the same period.
4. **Suggested, not proven.** That traffic was being silently discarded somewhere on the path between the reader's ISP and Microsoft's edge. The absence of any RST and any ICMP unreachable is consistent with a silent drop and inconsistent with the server being down or actively refusing.
5. **Not established.** Which party dropped it, or why. Traceroute showed `* * *` from hop 13 onward, but many routers suppress ICMP time-exceeded replies, so silence at the end of a trace proves nothing on its own.
6. What the git behaviour adds as evidence: the failure was at the transport layer, not at git's application layer. `git fetch` failing with `Could not read from remote repository` after a long stall is what a TCP connect timeout looks like from inside git.
7. Recovery pattern for this situation, in order:

Step	Command	Why
1	git status	see local state
2	git log –oneline –graph	see what accumulated
3	git fetch –all	refresh the caches
4	git log origin/main..main	what you still owe
5	git log main..origin/main	what you missed
6	git push	publish

8. Step 3 is the one to insist on. Until a fetch succeeds, every "ahead" and "behind" number on the screen is arithmetic on a stale cache.
9. Note the asymmetry between steps 4 and 5. `origin/main..main` lists commits you have that the remote lacked. `main..origin/main` lists the reverse. The two-dot syntax is not symmetric and the order matters.
10. If the push had failed midway with `send-pack: unexpected disconnect while reading sideband packet`, the correct response is not to guess. It is to re-query the server, because a disconnect while reading the response tells you nothing about whether the write landed. Chapter 40 covers that case in full.
11. Long-outage tip: `git bundle create out.bundle main` writes a single file containing commits, which can travel on a USB stick and be cloned or fetched from on another machine. It is the offline transport of last resort and it is built into git.

37.9.6 WORDS - remember these

1. Outage — a period when the network path does not work — loss of reachability between two endpoints, distinct from either endpoint being down.
2. Silent drop — packets vanish with no error returned — a discard with no RST and no ICMP unreachable, leaving the sender to time out.
3. Accumulate — pile up locally until you can send — build a series of local commits with no intervening synchronization.
4. Authoring versus coordination — making your own work versus agreeing with others — the local-capable and network-required halves of a git workflow.

5. Bundle — a whole repository transfer in one file — `git bundle`, a pack plus refs, usable as a remote for clone and fetch.
6. Two-dot range — commits on one side and not the other — `A..B` meaning reachable from B but not from A, and not symmetric.

37.10 The trade-offs of the distributed model

37.10.1 PLAIN - in simple words

1. This book does not sell things, so here are the costs of the distributed model as honestly as the benefits.
2. The benefits first, briefly. You can work with no network. Operations are fast because they are local. Every clone is a backup of the whole history. No single machine's failure loses the project.
3. Now the costs.
4. **The mental model is harder.** In Subversion there is one history and one revision number. In git there are as many histories as there are clones, and they may disagree, and you must reason about which is which.
5. **You cannot lock a file.** In some teams, a designer needs to say "nobody else touch this 400 MB scene file while I work on it". Git has no way to enforce that, because there is no central place to hold the lock.
6. **Files that cannot be merged are a real problem.** Two people edit the same image, the same 3D model, the same compiled asset. Git will detect the conflict and then be unable to help. One person's work must be thrown away.
7. **Whole-history clones are large.** For source code that is fine. For a project with twenty years of large binary assets it can be hundreds of gigabytes, and every new team member downloads all of it.
8. **There is no single source of truth** unless the team agrees on one. Git does not provide it, so the team, the server permissions and the process must.

37.10.2 PLAIN - a picture in your head

1. Distributed version control is like everyone owning a car instead of sharing one bus.
2. You go when you like. You do not wait for the timetable. If the bus company strikes, you still get to work.
3. But now there are twelve cars to park, twelve to fuel, and no driver telling everyone which route is correct. If two of you must arrive together, you have to arrange it yourselves.
4. And a bus is genuinely better for some journeys. If the whole team must move one enormous object, one vehicle with one driver is the right design.
5. Nobody thinks the bus company is foolish for existing. They are solving a different problem.

Where this comparison breaks:

6. Cars and buses cost about the same to run per person. Git and Perforce do not: git is free software and Perforce is licensed per user. Cost is part of why git spread, and it should be named.
7. And unlike vehicles, git and Perforce can coexist. Some studios run Perforce for art assets and git for engine code in the same building.

37.10.3 PLAIN - a worked example

1. Concretely, why a game studio chooses Perforce. Take a project with 800 GB of assets and 300 people.

Problem	Git	Perforce
New joiner setup	clone all history	sync latest only
Lock a scene file	not possible	exclusive checkout
Merge a 2 GB binary	impossible	prevented by lock

Problem	Git	Perforce
Check out one folder	awkward	native, standard
Work offline	full	very limited
Cost	free	per-seat licence

2. Rows one to four favour Perforce for this workload. Row five favours git. Row six favours git.
3. For a 300-person studio where everyone sits in one building with a fast local network, row five is worth little and rows one to four are worth a great deal.
4. For a distributed open-source project with 4,000 contributors on five continents who have never met, the weighting inverts completely.
5. Neither team is confused. They are optimizing different things, and the correct engineering answer is "it depends on the workload", which is an unsatisfying sentence that happens to be true.

37.10.4 PLAIN - what is really happening inside

1. The locking problem deserves precision, because it is the clearest structural limit of the distributed model.
2. A lock is a statement that everyone must agree on, at the same time. "Only Priya may edit `level3.blend` right now."
3. Agreement at the same time requires a single place that answers the question. That is exactly what a distributed system does not have.
4. So the absence of file locking in git is not laziness or a missing feature. It is a direct consequence of the architecture. You cannot have both.
5. Git LFS adds a lock command, and it works by contacting the LFS server. That is the same admission: to lock, you must centralize something.
6. The binary merge problem is different and is about file formats. Git merges text line by line. A `.psd`, a `.fbx` or a `.docx` has no lines, so there is nothing to merge.
7. Git will happily store binary files and version them. It just cannot combine two divergent edits, and it cannot compress them well, so the repository grows roughly linearly with every binary change.
8. The size problem compounds because git history is immutable. A 500 MB file committed once by mistake in 2019 is in every clone forever, unless someone rewrites history and every clone is re-made.

37.10.5 TECHNICAL - the engineer's version

1. Honest comparison across the main systems still in use in 2026.

System	Model	Typical domain
Git	distributed	source code, everywhere
Perforce P4	centralized	games, chips, film
Subversion	centralized	legacy enterprise
Mercurial	distributed	some large monorepos
Plastic SCM	centralized+	Unity game teams

2. Perforce, founded 1995 by Christopher Seiwald, remains the standard in game development, semiconductor design and visual effects. It received the 2019 CEDEC Award for Engineering, an award from the Japanese game industry, which tells you where its users are.
3. Perforce's exclusive checkout is implemented with the `+l` file type modifier, which makes the server grant a write lock to one client at a time.
4. Perforce's other decisive feature for these teams is the **client workspace view**, a mapping that lets a user sync one subdirectory of a 5 TB depot. Git has partial workarounds, sparse-checkout and partial clone, but they are less complete and less familiar.

5. Where experts disagree: some argue git plus LFS plus sparse-checkout now covers the game studio case and Perforce persists mainly through inertia and tooling lock-in. Others argue locking and workspace views are structural and git will never match them. Both positions are held by serious practitioners and the disagreement is genuine.
6. Scaling work on the git side is real and ongoing. Microsoft's VFS for Git, released in 2017 as GVFS, and later Scalar, merged into git itself in version 2.38 in October 2022, exist to make very large repositories usable.
7. Mitigations for the size problem, all mainstream as of 2026.

Technique	Reduces	Cost
Git LFS	clone size	needs LFS server
partial clone	clone size	fetches on demand
shallow clone	clone size	loses history
sparse-checkout	working tree	not history
commit-graph file	traversal time	none, local
multi-pack index	lookup time	none, local

8. Note which column those mitigations sit in. The last two are free local speedups. The first three all trade offline capability or history for size. That trade is the theme of this section.
9. `git maintenance start`, added in git 2.31, March 2021, schedules background packing, commit-graph writing and prefetching, and is the modern replacement for manually running `git gc`.

37.10.6 WORDS - remember these

1. File locking — one person at a time may edit a file — a server-granted exclusive write lock, unavailable in a purely distributed model.
2. Binary file — a file with no lines to compare — content without a text structure, so it cannot be three-way merged or delta-compressed well.
3. Monorepo — one repository holding many projects — a single history covering multiple products, common at very large companies.
4. Sparse checkout — having only part of the project in your folder — a cone-mode pattern list restricting which paths are materialized.
5. Source of truth — the copy everyone agrees is correct — an external convention plus server permissions, not a property of git itself.
6. Depot — Perforce's name for the server-side store — the centralized repository from which client workspaces are mapped and synced.

37.11 Git is not GitHub

37.11.1 PLAIN - in simple words

1. This confuses more beginners than anything else in the subject, so it goes in plainly.
2. **Git** is a program on your computer. You install it. It makes and reads files in a `.git` directory. It is free software, first released in 2005.
3. **GitHub** is a company that runs servers. You make an account. It stores copies of repositories and adds features around them. It launched on 10 April 2008 and Microsoft bought it in 2018.
4. Git does not need GitHub. GitHub needs git.
5. You can use git for thirty years and never make a GitHub account.
6. You can host repositories on GitLab, on Bitbucket, on a Raspberry Pi in a cupboard, on a USB stick, or on a company server. Git does not care.
7. The names are similar because GitHub chose a name based on git. That naming choice has cost the world a great deal of confusion.

8. Here is the sharpest possible test, and the reader lived it: during the outage, git kept working and GitHub did not.
9. That is not a coincidence. It is the boundary made visible.

37.11.2 PLAIN - a picture in your head

1. Git is a **word processor**. GitHub is a **print shop and a noticeboard**.
2. The word processor runs on your laptop. It works on a train. It never asks anyone's permission.
3. The print shop takes your document, keeps a copy, lets others read it, lets people leave notes on it, and runs a checking machine over it.
4. If the print shop closes, your word processor is unaffected and your document is still on your laptop.
5. If your laptop dies, the print shop still holds the copy you dropped off.
6. Neither one is the other. They cooperate through one narrow channel: you carrying documents in and out.

Where this comparison breaks:

7. A print shop holds a dumb copy. A GitHub repository is a real git repository and could replace yours entirely, which is why it feels authoritative.
8. And some things blur the line deliberately. GitHub Actions can commit to your repository. The GitHub web editor makes real commits. But those are GitHub running git on its own machines, which does not change the boundary.

37.11.3 PLAIN - a worked example

1. The division of labour, feature by feature.

Feature	Git	GitHub
commit, branch, merge	yes	no
full local history	yes	no
pull requests	no	yes
code review comments	no	yes
issues and projects	no	yes
CI via Actions	no	yes
access control, teams	no	yes
releases page	no	yes
hosting a remote	partly	yes

2. The `partly` in the last row: git includes `git daemon` and can serve over SSH, so a plain server with git installed is a perfectly good remote. It simply has no web interface.
3. During the reader's outage, the left column all worked and the right column all failed, because every item in the right column is an HTTPS request to a server in that unreachable direction.
4. The clearest evidence in that session: plain git operations behaved as described in 37.8, and GitHub API calls failed. Same laptop, same minute, different dependency.

37.11.4 PLAIN - what is really happening inside

1. What git actually is: a set of programs, mostly C, plus some shell and Perl. On a typical install it is a `git` binary and a directory of helper executables, around a hundred subcommands.
2. It speaks two wire protocols for talking to other repositories: the smart HTTP protocol and the SSH protocol, plus the older `git://` daemon protocol on port 9418. Chapter 40 covers all three.
3. What GitHub actually is: a large web application and API, backed by many servers that hold bare git repositories, wrapped in authentication, permissions, a review system, an issue tracker and a CI system.

4. When you `git push` to GitHub, git on your laptop opens a connection, runs the standard protocol, and a program on GitHub's side receives the pack. That receiving program is `git-receive-pack`, part of git.
5. So the push path is pure git at both ends. GitHub's own features do not participate.
6. When you open a pull request, none of that happens. Your browser or the `gh` command makes an HTTPS API call to GitHub. There is no git protocol involved and nothing is written to your `.git`.
7. That is why one worked offline and the other did not. They are different systems reached by different means.
8. The honest version: people say "push to GitHub" and "my code is on GitHub" as if GitHub were where the code lives. Your code lives on your disk. GitHub has a copy, which is extremely useful and is not the original.

37.11.5 TECHNICAL - the engineer's version

1. Git the software: created 2005, maintained by Junio Hamano since 26 July 2005, licensed under GPL version 2, current release 2.55.0 dated 29 June
2026. Written mainly in C.
2. GitHub the service: launched 10 April 2008 by Tom Preston-Werner, Chris Wanstrath, P. J. Hyett and Scott Chacon. Microsoft announced acquisition on 4 June 2018 for 7.5 billion US dollars in stock. The deal closed on 26 October 2018.
3. Alternatives, all of which speak the same git protocols: GitLab from 2011, Bitbucket from 2008, Gitea, Forgejo, Sourcehut, AWS CodeCommit, Azure Repos, and a plain SSH account on any Linux box.
4. Interface boundary in one table.

Thing you do	Protocol	Offline
git clone, fetch, push	git over HTTPS/SSH	needs net
git commit, log, merge	none	works
open a pull request	GitHub REST/GraphQL	needs net
gh pr list	GitHub REST/GraphQL	needs net
view Actions run	GitHub web/API	needs net

5. The `gh` command line tool is a GitHub client, not a git client. It has nothing to do with the `git` binary and fails independently of it. During the reader's outage this distinction was visible directly.
6. GitHub-specific concepts that do not exist in git at all: pull request, fork relationship, issue, label, milestone, review, required status check, branch protection rule, `mergeStateStatus`, Actions workflow, release, Gist.
7. Git concepts GitHub merely displays: commit, tree, blob, tag, branch, ref, merge, rebase, cherry-pick, reflog. GitHub cannot show you a reflog, because a reflog is local to a working repository.
8. Chapter 40 covers remotes, transports, authentication, tokens and scopes, and the exact division of labour when a push both moves refs and triggers a GitHub workflow.

37.11.6 WORDS - remember these

1. Git — the version control program on your machine — a distributed VCS implemented as a suite of C programs, GPLv2, first released 2005.
2. GitHub — a company hosting git repositories with extra features — a web service owned by Microsoft since 2018, offering pull requests, issues and CI.
3. Forge — the general word for a code hosting service — GitHub, GitLab, Bitbucket, Gitea and similar, all built around bare git repositories.
4. Pull request — a request to merge, with discussion — a forge feature, not a git feature, implemented over the forge's own API.

5. git-receive-pack — the program on the server that accepts a push — the receiving half of git's push protocol, run by the hosting service.
6. API — a way for programs to talk to a service — here, GitHub's REST and GraphQL interfaces over HTTPS, entirely separate from the git protocol.

37.12 Getting started properly

37.12.1 PLAIN - in simple words

1. There are exactly two ways a git repository comes to exist on your machine.
2. **`git init`** makes a new empty one, here, now, from nothing. No network.
3. **`git clone <url>`** copies an existing one, including all its history, and sets up a remote called `origin` pointing back at where it came from.
4. That is the complete list. Everything else is one of these two, possibly done by a tool on your behalf.
5. Before your first commit, tell git who you are. It records a name and an email address in every commit, permanently.
6. That email does not have to be real or reachable. Nothing sends mail to it. It is a label. But it is baked into the commit hash and cannot be quietly changed later without changing every hash after it.
7. Then create a `.gitignore` file listing things git should not track: build outputs, dependency folders, editor files, secrets.
8. And learn one hard fact early, because it catches everybody: `.gitignore` does **not** untrack a file that is already tracked.

37.12.2 PLAIN - a picture in your head

1. `git init` is starting a new notebook. Blank, yours, immediate.
2. `git clone` is photocopying someone's finished notebook, cover to cover, including their crossings-out, and writing their address inside your front cover so you can find them again.
3. `.gitignore` is a note on the front of the notebook saying "do not copy receipts into this book".
4. And here is the catch that the note cannot fix: if you already glued a receipt onto page 4, the note does nothing. The receipt is on page 4. It will be on page 4 in every photocopy anyone ever makes.
5. To stop tracking it you must deliberately remove it, and even then it stays in the old pages, because old pages are never edited.

Where this comparison breaks:

6. A photocopy is a dead copy. A clone is fully alive: it can commit, branch and serve as the source for further clones.
7. And unlike a notebook, you can start with `git init` and later attach a remote, which is the normal path for a project that begins locally.

37.12.3 PLAIN - a worked example

1. The full first-time setup, exactly as it should be typed.

```
git config --global user.name "Shikhar Singh"
git config --global user.email "you@example.com"
git config --global init.defaultBranch main
git config --global pull.rebase false
git config --global core.autocrlf input   # false on Windows
```

2. Then a new project from nothing:

```
mkdir myproject && cd myproject
git init
printf 'node_modules/\n*.log\n.env\n' > .gitignore
```

```
git add .
git commit -m "Initial commit"
```

3. That commit exists now, fully, with no account and no network.
4. Now the `.gitignore` trap, demonstrated with real output from a sandbox. A secret file was committed by mistake, then added to `.gitignore`.

```
$ git ls-files
.gitignore
app.py
config.env          <- still tracked

$ printf 'CHANGED\n' >> config.env
$ git status --short
 M config.env       <- git is still watching it
```

5. The `.gitignore` had no effect at all, because ignore rules apply only to files git is not already tracking.
6. The fix, and its limit:

```
$ git rm --cached config.env
$ git commit -m "stop tracking config.env"
$ git ls-files
.gitignore
app.py              <- now untracked

$ git show HEAD~2:config.env
secret-key          <- STILL IN HISTORY, forever
```

7. Read that last line carefully. The secret is out of the current commit and still recoverable from an old one. If it was a real credential, it is compromised and must be rotated. Removing it from history requires rewriting every commit after it, and every clone must be remade.

37.12.4 PLAIN - what is really happening inside

1. `git init` creates the `.git` directory, writes a default `config`, writes `HEAD` containing `ref: refs/heads/main`, and installs the sample hooks. It contacts nothing.
2. Note that `refs/heads/main` does not exist yet at that point. `HEAD` points at a branch that has no file. That is the "unborn branch" state, and it is why `git status` on a fresh repository says "No commits yet".
3. `git clone` does more. It creates the directory, runs `init`, adds the `[remote "origin"]` config, fetches everything, writes the remote-tracking refs, creates a local branch matching the remote's default, and checks it out.
4. Real output, so you can see how little it says:

```
$ git clone /tmp/gitdemo/server.git clonedemo
Cloning into 'clonedemo'...
done.
$ git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/feature/offline-work
  remotes/origin/main
```

5. Notice `remotes/origin/HEAD -> origin/main`. That records which branch the server said was its default, so `git checkout` of a bare remote name works.
6. `.gitignore` mechanics: on `git add` and on `git status`, git checks each **untracked** path against the ignore rules. Tracked paths skip the check entirely. That is the whole reason for the trap in 37.12.3.

7. Ignore rules come from several files, most specific first: the path's nearest `.gitignore`, then parent directories, then `.git/info/exclude`, then the file named by `core.excludesFile`.
8. `.gitignore` is committed and shared. `.git/info/exclude` is private and never leaves your machine. Use the second for editor droppings that only you produce.

37.12.5 TECHNICAL – the engineer's version

1. Settings worth changing on day one, with what each does.

Setting	Value	Effect
user.name	your name	commit author
user.email	your address	commit author
init.defaultBranch	main	branch on init
pull.rebase	false or true	pull behaviour
push.default	simple	push safety
core.autocrlf	input or false	line endings
fetch.prune	true	drop dead refs
rerere.enabled	true	reuse conflict fixes
diff.algorithm	histogram	better diffs

2. `init.defaultBranch` was added in git 2.28, July 2020. Before that the default was `master`, hard-coded. Git still defaults to `master` if you do not set it, and prints a hint saying so.
3. `push.default = simple` has been the default since git 2.0 in 2014. It pushes the current branch to its upstream of the same name and refuses otherwise, which prevents accidental mass pushes.
4. `core.autocrlf` matters only across operating systems. On Windows, `true` converts to CRLF on checkout and back to LF on commit. On macOS and Linux, `input` or `false`. Better still, commit a `.gitattributes` with `* text=auto` and stop thinking about it.
5. `rerere` stands for "reuse recorded resolution". With it on, git remembers how you resolved a conflict and applies the same resolution if it sees the same conflict again. It is invaluable during a long rebase.
6. Verify what is set and from where:

```
$ git config --list --show-scope --show-origin
system  file:/etc/gitconfig filter.lfs.required=true
global  file:/root/.gitconfig   user.name=Shikhar
global  file:/root/.gitconfig   user.email=shikhar@example.com
```

7. `git check-ignore -v --no-index <path>` tells you which rule in which file at which line number is ignoring a path. Without `--no-index` it deliberately ignores tracked files, which is itself a source of confusion.

```
$ git check-ignore -v --no-index config.env
.gitignore:1:config.env config.env
```

8. For removing a secret from all history the current recommended tool is `git filter-repo`, which replaced the much slower `git filter-branch`. The git project has discouraged `filter-branch` since version 2.24, November

2019. Either way, every existing clone becomes invalid and the credential must be rotated regardless.

9. `.gitignore` pattern rules follow `gitignore(5)`: `#` is a comment, a trailing `/` matches directories only, a leading `/` anchors to the `.gitignore`'s own directory, `**` matches across directories, and a leading `!` negates an earlier rule. A negation cannot re-include a file if its parent directory is itself excluded, which is the most common pattern bug.

37.12.6 WORDS - remember these

1. init — start a new empty repository here — create a `.git` directory with default config, HEAD and hooks, contacting nothing.
2. Clone — take a full copy of an existing repository — init plus a remote plus a full fetch plus a checkout of the default branch.
3. .gitignore — a list of things git should not track — patterns matched against untracked paths only, per `gitignore(5)`, committed and shared.
4. Untracked file — a file git knows nothing about — a working tree path with no index entry, eligible for ignore rules.
5. Tracked file — a file git is watching — a path present in the index, immune to `.gitignore` until removed with `git rm --cached`.
6. Unborn branch — HEAD points at a branch with no commits yet — the state after `git init`, before the first commit creates the ref.
7. rerere — git remembering how you solved a conflict before — reuse recorded resolution, enabled with `rerere.enabled true`.

37.98 Common wrong ideas

1. Wrong: git needs a server. Right: git needs a directory. `git init`, `git add`, `git commit`, `git branch` and `git merge` all work on a machine that has never been connected to anything.
2. Wrong: `origin` is the real repository. Right: `origin` is a nickname for a URL stored in `.git/config`. It has no authority in git's data model. Rename it, remove it, or have twenty of them.
3. Wrong: `origin/main` tells you where the remote's main branch is. Right: it tells you where it was the last time a fetch, pull or push succeeded on this machine. It is a file on your disk and it never expires.
4. Wrong: cloning downloads the current version of the project. Right: cloning downloads every commit, every version of every file, every branch and every tag, then checks out one of them into your folder.
5. Wrong: git and GitHub are the same thing. Right: git is a program from 2005 that runs on your computer. GitHub is a company from 2008, owned by Microsoft since 2018, that hosts copies and adds pull requests, issues and CI.
6. Wrong: a commit is saved to the cloud. Right: a commit writes objects and one ref into `.git` on your own disk. Nothing leaves your machine until you push.
7. Wrong: `git pull` is the safe way to get up to date. Right: `git pull` is `git fetch` plus a merge or rebase that changes your branch. `git fetch` alone updates your knowledge and changes nothing of yours.
8. Wrong: adding a file to `.gitignore` removes it from the repository. Right: ignore rules apply only to untracked files. An already-tracked file keeps being tracked until `git rm --cached`, and stays in history forever.
9. Wrong: "ahead 3, behind 2" is a live comparison with the server. Right: it is arithmetic between two local refs, one of which is a cache that may be weeks old. Run `git fetch` before believing it.
10. Wrong: a shallow clone is just a smaller clone. Right: a shallow clone has no old history, so `git log`, `git blame` and `git bisect` cannot see past the cut, and deepening requires the network.

37.99 Chapter summary in 20 lines

1. Version control solves six problems: history, undo, blame, parallel work, backup and collaboration. Only the last two need another computer.
2. Dated folders fail because they record copies without recording reasons or relationships, and because two people cannot use them at once.
3. The lineage runs SCCS 1972, RCS 1982, CVS from 1986 and 1990, Subversion from 2000 with 1.0 in February 2004, then git in April 2005.
4. In a centralized system the history lives on a server and your machine holds a working copy, so a commit is a network operation.

5. In a distributed system every clone is a complete repository, so a commit is a local file write. This is why the reader kept committing during the outage.
6. Git exists because BitMover ended the kernel's free BitKeeper licence in April 2005. Torvalds started on 3 April and git was self-hosting on 7 April.
7. His stated goals were speed, simple design, strong support for non-linear development, fully distributed operation, and handling the Linux kernel.
8. A `.git` directory holds objects, refs, HEAD, config, index, hooks, logs, info, description and sometimes packed-refs. That is the entire repository.
9. A branch is a 41-byte file containing a commit hash. That is why branching is instant, free and offline.
10. In a real clone of the git project, `.git` was 316 MiB against a 60 MiB working tree, holding 418,900 objects and 85,263 commits.
11. Compression is why: about 7,877 MiB of raw object content stored in about 302 MiB, a ratio near 26 to 1, via zlib and pack deltas.
12. Shallow and partial clones trade history or content for size, and both reintroduce the network into operations that were previously local.
13. `origin` is a nickname for a URL in `.git/config`. It is a convention, not a standard, it carries no authority, and a repository may have zero remotes.
14. `origin/main` is a local file recording where you last saw the remote's main. It is a memory, not a measurement, and it never expires by itself.
15. Only `fetch`, `pull` and `push` update remote-tracking refs. `git reflog show origin/main --date=iso` tells you how stale yours is.
16. Local commands touch only objects, refs, index and working tree. Remote commands synchronize two object databases. The split has no fuzzy middle.
17. During the outage, authoring was fully available and coordination was fully unavailable: no fetching others' work, no pull requests, no CI results.
18. The costs of the distributed model are a harder mental model, no file locking, no useful merging of binaries, large clones, and no built-in truth.
19. That is why game studios, chip designers and visual effects houses still buy Perforce, founded 1995, for its exclusive checkout and workspace views.
20. Git is a program on your computer. GitHub is a company that hosts copies. During the outage git worked and the GitHub API did not. That is the line.

38

CHAPTER

KB

What a Commit Actually Is

38.0 What this chapter gives you

1. You will be able to say what content-addressed storage is: the name of a thing is worked out from the thing itself, so equal content always gets equal names.
2. You will be able to explain what a hash function is, name the five properties that matter, and say why SHA-1 gives 160 bits printed as 40 hexadecimal characters.
3. You will be able to state honestly that SHA-1 was broken in public on 23 February 2017 by Google and CWI Amsterdam, say what git did about it, and say why git's use of it was never purely a security control.
4. You will be able to name git's four object types, blob, tree, commit and tag, describe the exact bytes each one stores, and rebuild any object's hash by hand with one shell command.
5. You will be able to draw the complete object graph of a small repository, every blob, every tree, every commit and every arrow, and say exactly which objects are created and which are reused when one file changes.
6. You will be able to explain why history is a directed acyclic graph and not a line, what a merge commit with two parents looks like inside, and why the tip hash seals everything behind it.
7. You will be able to say what a branch really is, a small text file holding 40 hexadecimal characters, and why that makes branching almost free.
8. You will be able to read and repair the state of HEAD, including detached HEAD, and recover work that looked lost after a hard reset by using the reflog.
9. You will be able to use the plumbing commands that show the machinery: `git cat-file`, `git rev-parse`, `git hash-object`, `git ls-tree`, `git count-objects`, `git verify-pack` and `git rev-list`.
10. You will be able to explain the reader's own observation that `origin/main` was `0a95cc8`, and why a rebased branch inevitably came out with a different hash.

38.1 Content-addressed storage, the central idea

38.1.1 PLAIN – in simple words

1. Normally you give a thing a name and then put content inside it. A folder called `notes` holds whatever you put in it today.
2. Git does the opposite. You give it content, and git works out the name from the content.
3. The name is a long string of letters and digits, produced by feeding every byte of the content through a fixed piece of arithmetic.
4. This is called **content-addressed storage**. The address of a thing is computed from the thing.
5. Two consequences follow, and they are the whole reason git works the way it does.
6. First, the same content always gets the same name. So git can never store the same content twice. It looks up the name, finds it already there, and stops.

7. Second, if anyone changes the content even by one bit, the name no longer matches. So changing content without being noticed is not possible.
8. Everything else in this chapter is a consequence of those two sentences.

38.1.2 PLAIN - a picture in your head

1. Think of a very strange left-luggage office at a railway station.
2. You hand over a bag. The clerk does not give you a numbered ticket chosen from a pile.
3. Instead the clerk weighs the bag, measures it, catalogues every object inside it, and turns all of that into one long code.
4. That code is your ticket. It describes the bag rather than pointing at a shelf.
5. If you hand in an identical bag tomorrow, you get the identical ticket. The clerk says "I already have that one" and stores nothing new.
6. If somebody opens your bag and removes one sock, then re-runs the measurement, they get a different code. Your old ticket no longer matches anything. The swap is visible.

Where this comparison breaks:

7. A real clerk could be lazy and reuse a code for two nearly identical bags. A hash function cannot be lazy. It is arithmetic, and it runs the same way every time.
8. A real bag can be measured many ways. Git measures exactly one way, over exactly the bytes, in exactly one order, so the answer is never a matter of judgement.
9. And a left-luggage ticket does not tell you what is in the bag. A git hash does not either, but it is enough to prove that the bag you get back is the bag you handed in.

38.1.3 PLAIN - a worked example

1. Here is a real repository built for this chapter. Four files were created, three of them holding exactly the same line of text.
2. The command `git ls-tree -r HEAD` lists every file with the name git gave to its content.

```
100644 blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3    a/one.txt
100644 blob c9330452ddba7055160034759c594f78381fd24d    a/other.txt
100644 blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3    b/two.txt
100644 blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3    c/three.txt
```

3. Three files, three different paths, one identical name: `6fe3e87...`.
4. Now count how many pieces of file content git actually stored:

```
6fe3e87371433ed417646e3cb2354cfb9cff08d3 blob 22
c9330452ddba7055160034759c594f78381fd24d blob 18
```

5. Two. Not four. The three identical files share one stored copy of 22 bytes.
6. Nobody asked for that. Nobody ran a de-duplication tool. It falls out of naming things by their content.
7. Now rename `a/one.txt` to `a/renamed.txt` and commit. The listing changes, but the stored content list does not change at all.

```
100644 blob c9330452ddba7055160034759c594f78381fd24d    a/other.txt
100644 blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3    a/renamed.txt
100644 blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3    b/two.txt
100644 blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3    c/three.txt
```

8. The file name lives in the directory listing, not in the content. Renaming is free.

38.1.4 PLAIN - what is really happening inside

1. When you run `git add file.txt`, git reads every byte of the file.
2. It puts a small header in front of those bytes. The header is the word `blob`, a space, the length in bytes, and a zero byte.
3. It runs the header plus the content through the hash function. Out comes a 160-bit number.
4. It prints that number as 40 hexadecimal characters. That is the object's name.
5. It then looks on disk for a file at `.git/objects/` plus the first two characters, a slash, then the other 38 characters.
6. If that file already exists, git does nothing more. The content is already stored, by definition, because the name proves it.
7. If it does not exist, git compresses the header plus content with zlib and writes it there.
8. Reading works in reverse. Given a name, git knows the path, reads the file, decompresses it, and hands back the bytes.
9. Because the file is named after its own content, git can check itself at any time. Re-hash the content, compare with the file name, and any difference is damage or tampering.

38.1.5 TECHNICAL - the engineer's version

1. Git implements a content-addressable object store. The key space is the output of a cryptographic hash over `<type> <size>\0<content>`.
2. The store is also a **Merkle tree**, the structure Ralph Merkle described in his 1979 doctoral work: a tree in which every parent node contains the hash of its children.
3. That is why one hash at the top, the commit hash, transitively covers every byte in the tree below it.
4. Loose objects live at `.git/objects/ab/cdef...`. The two-character sub-directory exists because early filesystems degraded badly with hundreds of thousands of entries in one directory.
5. Here is a real loose object, on disk, byte for byte:

```
00000000  78 01 4b ca c9 4f 52 30  34 61 f0 cc 2b 29 ca 4f
00000010  29 4d 2e c9 cc cf d3 e3  02 00 54 63 07 5f
```

6. That file is 30 bytes. The leading `78 01` is a zlib header (RFC 1950, May 1996, by Jean-loup Gailly and Mark Adler). Decompressed it is 22 bytes:

```
b'blob 14\x00Introduction.\n'
```

7. So the stored object is `blob`, space, `14`, a NUL byte, then the 14 bytes of file content. The hash is taken over all 22 of those bytes.
8. Deduplication is exact-match only. Git does not find near-duplicates at store time. Similarity is exploited later, inside packfiles, and that is a separate mechanism covered in 38.11.

Property of the store	Value in git
Key derivation	SHA-1 of header+content
Key length	160 bits, 40 hex chars
Loose object path	`objects/xx/38-char-rest`
Object compression	zlib deflate, RFC 1951
Duplicate handling	write skipped if key exists

9. The honest version: content addressing gives integrity, not secrecy. Anyone with the object store can read every byte. Encryption is a separate concern that git does not provide.

38.1.6 WORDS - remember these

1. Content-addressed storage — the name comes from the content — key space is the image of a hash function over the stored bytes.
2. Blob — a stored piece of file content — a git object of type `blob` holding raw bytes with no name and no permissions.
3. Deduplication — storing one copy of identical things — exact-match coalescing by hash key at write time.
4. Merkle tree — a tree where each parent names its children by hash — hash tree, described by Ralph Merkle in 1979, giving transitive integrity.
5. zlib — the squeezing format git uses on disk — DEFLATE stream in a zlib wrapper, RFC 1950 and RFC 1951, both May 1996.

38.2 Hashing, and the honest truth about SHA-1

38.2.1 PLAIN - in simple words

1. A **hash function** takes any amount of data and produces a short fixed-size number that stands for it.
2. Five things matter about it.
3. It is **deterministic**: the same input always gives the same output, on any machine, in any year.
4. It has a **fixed size** output: one byte in or one gigabyte in, the answer is the same length.
5. It has the **avalanche effect**: change one bit of input and about half the output bits flip. There is no gentle drift.
6. It is **one-way**: given the output you cannot work backwards to the input, other than by guessing.
7. It is **collision resistant**: it should be impractical to find two different inputs with the same output.
8. Git used SHA-1, which produces 160 bits. Printed four bits per character in hexadecimal, that is 40 characters.
9. The honest version: the fifth property, collision resistance, is broken for SHA-1. That is not a rumour, it is a demonstrated public fact from 2017.

38.2.2 PLAIN - a picture in your head

1. Think of a machine that turns any document into a 40-character summary code.
2. Feed it a shopping list, get a code. Feed it the complete works of a novelist, get a code of exactly the same length.
3. Change one comma anywhere in the novel and the code changes beyond recognition, not slightly.
4. You cannot run the machine backwards. Given a code, there is no way to get the novel out of it.
5. Collision resistance is the promise that nobody can make two different documents that produce the same code.
6. That promise held for SHA-1 for about twenty-two years. Then it did not.

Where this comparison breaks:

7. A summary suggests something readable. A hash output is not readable and carries no meaning at all. It is only a label.
8. Also, a real summary of a longer document loses information gently. A hash loses nearly all information at once. From 160 bits you can prove a match, but you can never reconstruct anything.

38.2.3 PLAIN – a worked example

1. Here are real SHA-1 values computed in the sandbox used to write this chapter, on an Intel Xeon at 2.10 GHz.
2. Determinism. The same five letters, hashed three times:

```
$ printf 'hello' | sha1sum
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
```

3. Avalanche. Change the last letter from `o` to `p`:

```
hello  -> aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
hellp  -> d4baaad7a68a379b1e133e0fea0603051b0124ca
```

4. Of the 40 hexadecimal characters, only 3 happen to be equal. Of the 160 bits, 85 differ. The ideal for a good hash is 80. It is behaving correctly.
5. Fixed size. One byte in, and one million bytes in:

```
1 byte        -> 86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
1,000,000 B   -> bef3595266a65a2ff36b700a75e8ed95c68210b6
```

6. Both answers are 40 characters. The length of the input is invisible in the output.
7. Now the git-specific part. Git does not hash the file. It hashes a header plus the file. Here is the proof, done by hand:

```
$ printf 'Introduction.\n' | wc -c
14
$ printf 'blob 14\0Introduction.\n' | sha1sum
0e2a325c7de3369b164d48fcba823bca9c2cec4e
$ git hash-object docs/intro.md
0e2a325c7de3369b164d48fcba823bca9c2cec4e
```

8. Identical. `sha1sum` and git agree, because git's rule is exactly "hash the header plus the content".

38.2.4 PLAIN – what is really happening inside

1. SHA-1 works on the input in blocks of 64 bytes.
2. It keeps five 32-bit working registers, 160 bits in total, initialized to fixed constants written into the specification.
3. For each block it mixes the block into those registers through 80 rounds of addition, rotation and bit logic, then adds the result back into the running state.
4. When the input runs out, it appends a single 1 bit, then zero bits, then the original bit length as a 64-bit number, and processes the final block.
5. The five registers, printed end to end, are the answer.
6. The reason a one-bit change scatters the output is that the mixing is designed so every input bit influences every register within a few rounds.
7. A **collision** means two different inputs that end with the same five registers. Because inputs are unlimited and outputs are only 160 bits, collisions must exist. The question is only whether anyone can find one.
8. In 2017 somebody did, on purpose, with two files chosen in advance to be different in a way that mattered.

38.2.5 TECHNICAL – the engineer's version

1. SHA-1 is specified in FIPS PUB 180-1, published by the United States National Institute of Standards and Technology in 1995. It replaced SHA-0 from FIPS PUB 180 in 1993, which had a design flaw.

2. Output is 160 bits. Internal state is five 32-bit words. Block size is 512 bits. Rounds per block: 80.
3. On 23 February 2017 researchers at CWI Amsterdam and Google announced **SHAttered**, the first public SHA-1 collision: two different PDF files with the identical SHA-1 digest.
4. The published cost was approximately 2 to the power 63.1 SHA-1 evaluations, described as the equivalent of 6,500 CPU-years plus 110 GPU-years, and about 100,000 times cheaper than a brute-force birthday search.
5. On 5 January 2020 Gaetan Leurent and Thomas Peyrin announced **SHAmbles**, a chosen-prefix collision at about 2 to the power 63.4, costed at roughly 45,000 US dollars of rented GPU time. Chosen-prefix is the dangerous kind, because the attacker picks both meaningful beginnings.
6. NIST formally deprecated SHA-1 in 2011, disallowed it for digital signatures from 2013, and in 2022 set 31 December 2030 as the full phase-out date.
7. Here is what those numbers mean in the machine used for this chapter. It hashes at about 1,241 MB/s, which is about 1.95 x 10^7 blocks per second.

Work item	Evaluations	Time on this one machine
SHAttered, 2^63.1	9.9 x 10^18	about 16,000 years
Birthday bound 2^80	1.2 x 10^24	about 2 x 10^9 years
Full preimage 2^160	1.5 x 10^48	about 2 x 10^33 years

8. What git did. From **Git 2.13, released in May 2017**, git ships the collision-detecting SHA-1 implementation by Marc Stevens of CWI and Dan Shumow of Microsoft. The Git 2.13 release notes say it "has been integrated and made the default".
9. That code computes SHA-1 normally but also watches for the internal disturbance vectors that any known collision attack must use. If it sees them, git aborts rather than storing the object. The two SHAttered PDFs cannot be committed to a modern git.
10. Separately, git has been growing a SHA-256 object format. The design document dates from 2017 and 2018, experimental support began landing from **Git 2.19 in September 2018**, and `git init --object-format=sha256` arrived in **Git 2.29 in October 2020**.
11. It works. In the sandbox:

```
$ git init --object-format=sha256 sha256demo
$ git hash-object intro.md
22c6af3b0fccae1e36b21923804fd096c8636e54da038f04c424208
d2ecb1650
```

12. That is 64 hexadecimal characters, and it verifies by hand exactly the same way: `printf 'blob 14\0Introduction.\n' | sha256sum` gives the same value.
13. But it is still not the default in 2026, and git's own documentation says that until the wire protocol gains SHA-256 support, using SHA-256 storage on public-facing servers is strongly discouraged. Repositories in the two formats cannot yet talk to each other.
14. The honest version, and the part people argue about: Linus Torvalds has said publicly that git's hash was chosen mainly to guard against accidental corruption and to give a stable name, not as a signature scheme. Critics reply that in practice people do rely on commit hashes as identity, in deploy pipelines and in audit trails. Both points are true.
15. Where experts disagree: some argue the SHA-256 migration is overdue and the ecosystem cost is worth paying now. Others argue that with collision detection turned on, a real attack on a git repository is far harder than an attack on a bare hash, so the migration can proceed slowly. There is no settled answer.

38.2.6 WORDS - remember these

1. Hash function — arithmetic that turns any data into a short fixed code — a map from arbitrary-length input to fixed-length digest.

2. Avalanche effect — one small change scrambles the whole answer — a one-bit input change flips about half the output bits.
3. Collision — two different inputs with the same code — two distinct pre-images mapping to one digest.
4. Preimage resistance — you cannot work backwards from the code — finding any input for a given digest should cost about 2^160 for SHA-1.
5. SHAttered — the 2017 demonstration that SHA-1 collisions are buildable — CWI Amsterdam and Google, 23 February 2017, cost about 2^63.1.
6. sha1dc — git's hardened SHA-1 that spots attacks — the collision-detecting SHA-1 of Marc Stevens and Dan Shumow, default since Git 2.13, May 2017.

38.3 The four object types

38.3.1 PLAIN - in simple words

1. Git stores exactly four kinds of object. Everything in a repository is one of these four.
2. A **blob** is the content of one file. Just the bytes. It does not know its own name, its folder, or when it was made.
3. A **tree** is one directory listing. It maps names to blobs and to other trees, and records each entry's permission mode.
4. A **commit** is a snapshot marker. It names exactly one tree, names zero or more parent commits, records who wrote it and who recorded it with times, and carries a message.
5. A **tag object** is a named, described pointer, normally to a commit, with its own author and message, and optionally a signature.
6. That is the whole vocabulary. Four nouns.
7. Notice what is missing. There is no "file" object holding a name, no "change" object, no "diff" object, and no "branch" object. Those are all built from the four.

38.3.2 PLAIN - a picture in your head

1. Think about a warehouse that stores photographs of a whole office, taken every day.
2. A blob is one printed page in a filing cabinet. It has no label on it. It is just the paper.
3. A tree is an index card for one room. It says "the page called README is card number f6da25f, the cupboard called docs is index card 80b4ad5".
4. A commit is the front cover of one day's survey. It says "here is the index card for the whole office, taken at 10:00 on 5 January, by this person, for this reason, and yesterday's cover was that one over there".
5. A tag object is a brass plaque screwed on to one particular day's cover, saying "this one is version 1.0, blessed by this person on this date".

Where this comparison breaks:

6. In a warehouse, tomorrow's survey needs a whole new set of pages. In git, tomorrow's cover reuses every index card and every page that did not change. Only the changed cards are re-cut.
7. And in a warehouse the covers are dated by the clerk. In git the cover's own identity is computed from its contents, including the date, so you cannot alter the date and keep the identity.

38.3.3 PLAIN - a worked example

1. Here is a real repository. Three commits, two directories, five files by the end. Every hash below came out of the sandbox.
2. A blob. `git cat-file -t` says the type, `-s` says the size in bytes, `-p` pretty-prints the content:

```
$ git cat-file -t 0e2a325
blob
$ git cat-file -s 0e2a325
```

```
14
$ git cat-file -p 0e2a325
Introduction.
```

3. Notice: no file name anywhere. This blob does not know it is `intro.md`.
4. A tree. This is the top directory of the first commit:

```
$ git cat-file -t b39d894
tree
$ git cat-file -p b39d894
100644 blob f6da25f44e6006699370496c1d1033320df8802b    README.md
040000 tree 80b4ad5b31db187c032b03fca7a3b9a0cbdc2747    docs
040000 tree ed46dd19f6204c92947780231982716d4016b9bb    src
```

5. Three entries. One blob and two sub-directories. The names live here, in the tree, not in the blobs.
6. A commit. This is the third commit:

```
$ git cat-file -t b53a9f4
commit
$ git cat-file -p b53a9f4
tree 19f6e3d84c4955832c7f9d39e5205c273dd15d91
parent fb17f98cd37f546dfedb2f91e1b7a9762bf45d17
author KedByte Reader <reader@example.com> 1767594600 +0530
committer KedByte Reader <reader@example.com> 1767594600 +0530

Third commit: add src/util.c
```

7. That is the entire commit. One tree line, one parent line, two people lines with timestamps, a blank line, then the message.
8. A tag object, made with `git tag -a v1.0 -m "First release of the demo"`:

```
$ git cat-file -t v1.0
tag
$ git cat-file -p v1.0
object b53a9f46d432aa53e687328841c6dd2b89963809
type commit
tag v1.0
tagger KedByte Reader <reader@example.com> 1767598200 +0530

First release of the demo
```

9. The tag object is a separate object with its own hash, `d54ba1826a59225bc14a2b55e832536cb7a8e4ac`, pointing at the commit.

38.3.4 PLAIN – what is really happening inside

1. Every one of the four objects is stored the same way: a header, then the content, hashed and compressed together.
2. The header is the type word, a space, the content length in bytes, then a single zero byte.
3. So the whole rule of the object store is one line: `hash = SHA1(type + " " + size + "\0" + content)`.
4. Let us prove it for all four types with real numbers.
5. Blob. Content is 14 bytes.

```
$ printf 'blob 14\0Introduction.\n' | sha1sum
0e2a325c7de3369b164d48fcba823bca9c2cec4e
```

6. Tree. Content is 34 bytes of binary, so we pipe git's own raw bytes back in:

```
$ git cat-file tree ed46dd19 | wc -c
34
```

```
$ { printf 'tree 34\0'; git cat-file tree ed46dd19; } | sha1sum
ed46dd19f6204c92947780231982716d4016b9bb
```

7. Commit. Content is 208 bytes:

```
$ git cat-file commit d9cbb74 | wc -c
208
$ { printf 'commit 208\0'; git cat-file commit d9cbb74; } | sha1sum
d9cbb74f1fb9bce240d20db0c8e545caf68c802b
```

8. Tag. Content is 156 bytes:

```
$ { printf 'tag 156\0'; git cat-file tag v1.0; } | sha1sum
d54ba1826a59225bc14a2b55e832536cb7a8e4ac
```

9. Four for four. There is no secret step. You can reproduce every hash in a git repository with `printf` and `sha1sum`.
10. One warning about the tree. `git cat-file -p` shows you a friendly version. The bytes on disk are not that. Here is the truth:

```
00000000  31 30 30 36 34 34 20 52  45 41 44 4d 45 2e 6d 64
00000010  00 f6 da 25 f4 4e 60 06  69 93 70 49 6c 1d 10 33
00000020  32 0d f8 80 2b 34 30 30  30 30 20 64 6f 63 73 00
00000030  80 b4 ad 5b 31 db 18 7c  03 2b 03 fc a7 a3 b9 a0
00000040  cb dc 27 47 34 30 30 30  30 20 73 72 63 00 ed 46
00000050  dd 19 f6 20 4c 92 94 77  80 23 19 82 71 6d 40 16
00000060  b9 bb
```

11. Each entry is: the mode in ASCII, a space, the file name in ASCII, a zero byte, then the 20 raw bytes of the hash. Not 40 hex characters. 20 binary bytes.
12. Also notice the directory mode is stored as `40000`, five characters, even though `git ls-tree` prints `040000`. The printed leading zero is cosmetic.

38.3.5 TECHNICAL - the engineer's version

1. Object format, canonical for both SHA-1 and SHA-256 repositories: `<type> SP <size in bytes, decimal ASCII, no leading zeros> NUL <content>`.
2. Tree entry format, repeated and concatenated with no separator: `<mode ASCII> SP <name> NUL <20-byte raw object id>`.
3. Tree entries are sorted by name, with directory names compared as if they ended in a slash. This ordering is part of the format, not a convention: two git implementations must agree or the hashes differ.
4. Only five modes are legal in a tree. All five were produced in the sandbox:

```
040000 tree 0479003445f4e5a5ff25360c607ca79ffe4e4ea1    d
120000 blob dab8c79946b1756dcd7db770a986ad40d00c07f4    link.txt
100644 blob b9bca019c83a65e6d717d0b6da86215f45dde1b3    plain.txt
100755 blob 4163036efa65bd4a469e752267498f01ea36a55c    run.sh
160000 commit 0f0c4412ec5e57f084c7a39a254a1f9d67f3e965  sub
```

Mode	Meaning
100644	regular file
100755	executable regular file
120000	symbolic link
040000	directory, a sub-tree
160000	gitlink, a submodule commit

5. Git records only the executable bit. It does not record read or write permission, owner, group, or modification time. Those are deliberately outside the model.
6. A symbolic link is stored as a blob whose content is the target path. The blob above holds exactly the nine bytes `plain.txt`.
7. Commit object grammar, in order: one `tree` line, then zero or more `parent` lines, then `author`, then `committer`, then optional headers such as `gpgsig` and `encoding`, then a blank line, then the message.
8. Timestamps are two fields: seconds since the Unix epoch of 1 January 1970 UTC, then the author's local UTC offset as `+HHMM`. The offset is stored so the original wall-clock time can be reproduced. It does not affect ordering.
9. Two universal object ids you will meet in real work, and they are the same in every SHA-1 repository on earth:

```
$ git hash-object -t tree /dev/null
4b825dc642cb6eb9a060e54bf8d69288fbee4904
$ git hash-object -t blob /dev/null
e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

10. The first is the empty tree, useful for diffing a first commit against nothing. The second is the empty file.
11. You can construct all of this by hand. The plumbing path from bytes to a checked-out commit, run in the sandbox:

```
$ B=$(printf 'Hello from plumbing.\n' | git hash-object -w --stdin)
$ printf '100644 blob %s\thello.txt\n' "$B" | git mktree
ae0afea72a045c4653880d3b19ffd7a4979d8d0d
$ git commit-tree ae0afea -m "Made entirely by plumbing"
f0803fab3f189b2ed4f674bf2c63655cc7d7cfde
$ git update-ref refs/heads/main f0803fa
$ git log --oneline
f0803fa Made entirely by plumbing
```

12. No `git add`, no `git commit`. The high-level commands are convenience over exactly these four steps.

38.3.6 WORDS - remember these

1. Blob — stored file content with no name — an object of type `blob`, raw bytes, addressed by SHA of `blob <size>\0<bytes>`.
2. Tree — one directory listing — an object mapping sorted names to modes and object ids, using 20 raw bytes per id.
3. Commit — a snapshot marker with a message and a parent — an object naming one tree, zero or more parents, author, committer and message.
4. Annotated tag — a named marker that is itself a stored object — an object of type `tag` with tagger, message and optional signature.
5. Mode — the small number in front of each tree entry — the six-digit file mode; only 100644, 100755, 120000, 040000 and 160000 are valid.
6. Gitlink — a pointer to a commit in another repository — a tree entry with mode 160000 recording a submodule's commit id.

38.4 How the four types compose

38.4.1 PLAIN - in simple words

1. The four object types stack into one picture.
2. A commit points down at one tree. That tree points at blobs and at more trees. Those point at more blobs.
3. The whole content of the project at one moment hangs off that single commit hash.

4. A commit also points sideways, at its parent commit, which has its own tree, which shares most of the same blobs and sub-trees.
5. When you change one file, git does not copy the project. It creates one new blob for that file, one new tree for the directory containing it, one new tree for each directory above that, and one new commit.
6. Everything else is pointed at again, unchanged.
7. So the cost of a commit is proportional to the depth of the change, not to the size of the project.

38.4.2 PLAIN – a picture in your head

1. Imagine a company organization chart printed on a wall, with a card for each department and a card for each person.
2. Somebody changes their phone number. You do not reprint the wall.
3. You print one new card for that person, one new card for their team naming the new person card, one new card for their division naming the new team card, and one new card for the whole company.
4. Every other card on the wall is still valid and is still pointed at by the new cards.
5. Now you have two complete company charts on the wall, yesterday's and today's, and they share almost all their cards.

Where this comparison breaks:

6. Cards on a wall are pointed at by position. Git objects are pointed at by hash, so a "new card" for identical content is impossible. If two people ended up with the identical card, there would be exactly one card.
7. And a wall chart has to be walked from the top. Git can jump directly to any card, at any depth, if it knows the hash.

38.4.3 PLAIN – a worked example

1. Here is the complete object graph of the real three-commit repository, with every object and every arrow. Fifteen objects: 3 commits, 7 trees, 5 blobs.

```
COMMITS  (each points left to its parent)

 d9cbb74 <----- fb17f98 <----- b53a9f4
   C1             C2             C3
   |              |              |
   | tree         | tree         | tree
   v              v              v

ROOT TREES

 b39d894        5b9f378        19f6e3d
  |  |  |        |  |  |        |  |  |
  |  |  +--src-->|  |  +--src-->|  |  +--src--+
  |  +--docs-+   |  +--docs-+   |  +--docs-+  |
  +--README-+|   +--README-+|   +--README-+|  |
             ||             ||             ||  |
             vv             vv             vv  v
```

2. Rather than draw crossing lines, here is the same information written out. Each root tree has exactly three entries:

```
b39d894  (root tree of C1)
  README.md -> blob f6da25f
  docs      -> tree 80b4ad5
  src       -> tree ed46dd1

5b9f378  (root tree of C2)
```

```
  README.md -> blob f6da25f      SAME as C1
  docs      -> tree dbec85e      NEW
  src       -> tree ed46dd1      SAME as C1

19f6e3d  (root tree of C3)
  README.md -> blob f6da25f      SAME as C1 and C2
  docs      -> tree dbec85e      SAME as C2
  src       -> tree 069dcc5      NEW
```

3. And the sub-trees, with their blobs:

```
80b4ad5  docs at C1        dbec85e  docs at C2 and C3
  intro.md -> 0e2a325        intro.md -> bce6753

ed46dd1  src at C1 and C2  069dcc5  src at C3
  main.c -> 78f2de1          main.c   -> 78f2de1
                             util.c   -> df76ea6
```

4. The five blobs, with their real sizes:

Blob	Bytes	It is the content of
f6da25f	25	README.md, all commits
0e2a325	14	docs/intro.md at C1
bce6753	38	docs/intro.md at C2, C3
78f2de1	29	src/main.c, all commits
df76ea6	22	src/util.c at C3

5. Now the important part. Commit 2 changed exactly one file, `docs/intro.md`. Here is what that cost.

```
EDIT ONE FILE IN ONE DIRECTORY: C1 -> C2

NEW objects (4)              REUSED objects (4)
------------------           --------------------
blob   bce6753  intro.md     blob f6da25f README.md
tree   dbec85e  docs/        tree ed46dd1 src/
tree   5b9f378  root         blob 78f2de1 src/main.c
commit fb17f98               blob 0e2a325 old intro.md
                             (kept, still in C1)
```

6. One blob for the new content. One tree for the directory it is in. One tree for the root above it. One commit. Four objects.
7. The `src` directory was not touched, so tree `ed46dd1` is pointed at again by the new root tree. Not copied. Pointed at.
8. That is the rule: new objects equal one blob, plus one tree per directory level from the change up to the root, plus one commit.
9. Adding `src/util.c` in commit 3 cost the same shape: new blob `df76ea6`, new `src` tree `069dcc5`, new root tree `19f6e3d`, new commit `b53a9f4`.

38.4.4 PLAIN - what is really happening inside

1. When you run `git commit`, git works from a staging area called the index, which holds the intended next snapshot as a flat list of paths and blob ids.
2. Git writes the deepest directories first. For each directory it builds the tree bytes, hashes them, and writes the tree object if it is not already there.
3. Because it hashes before writing, an unchanged directory produces exactly the same tree bytes as last time, so exactly the same hash, so no write happens at all.

4. Then it builds the parent directory's tree, which now contains the just-computed child hashes, and so on up to the root.
5. Finally it builds the commit object: the root tree hash, the current branch tip as parent, your identity and the time, and your message.
6. It hashes that, writes it, and moves the branch file to the new commit hash.
7. Nothing was diffed. Nothing was compared with the previous commit. The reuse is automatic because equal content produces equal names.
8. And here is the proof that git stores whole content, not changes. The second version of `docs/intro.md` is stored complete:

```
$ git cat-file -s bce6753
38
$ git cat-file -p bce6753
Introduction.
Now with a second line.
```

9. Thirty-eight bytes: the full new file, both lines. Not "add one line at the end".
10. The diff you see in `git show` is worked out at the moment you ask, by comparing two trees:

```
$ git diff-tree -r d9cbb74 fb17f98
:100644 100644 0e2a325... bce6753... M  docs/intro.md
```

11. Git compared the two trees, saw one entry whose hash differed, and only then computed the line-by-line difference for display.

38.4.5 TECHNICAL - the engineer's version

1. The object graph is a Merkle DAG. Nodes are objects, edges are embedded object ids, and it is acyclic because an object's id depends on the ids it contains, so a cycle would require a hash to contain itself.
2. Write cost of a commit is $O(d + 1)$ new objects, where d is the number of directory levels from the deepest change to the root, plus one commit and one blob per changed file.
3. That is why deep directory hierarchies cost slightly more per commit than flat ones, and why a repository with one enormous directory costs more per commit than you might expect: the whole tree object for that directory is rewritten whenever any entry in it changes.
4. Real figures from the sandbox repository at the three-commit point:

Measure	Value
Distinct objects	15
Commits / trees / blobs	3 / 7 / 5
Files in final snapshot	4
Tree objects if no reuse	9 instead of 7

5. Only two tree objects were saved in this tiny example. In a repository with thousands of directories and a hundred thousand commits, the saving is the difference between a workable repository and an impossible one.
6. Verify the reuse yourself with `git rev-parse` on paths:

```
$ git rev-parse d9cbb74^{tree}:src
ed46dd19f6204c92947780231982716d4016b9bb
$ git rev-parse fb17f98^{tree}:src
ed46dd19f6204c92947780231982716d4016b9bb
```

7. Identical, from two different commits. One stored object.

8. Implementation detail, not part of the model: git also keeps a cache-tree inside `.git/index` so that unchanged directories do not even need their tree bytes rebuilt during commit. That is a speed optimization; the resulting hashes are identical either way.

38.4.6 WORDS - remember these

1. Snapshot — the complete state of the project at one commit — the transitive closure of objects reachable from a commit's root tree.
2. Structural sharing — new versions reusing old parts — persistent data structure behaviour, where unchanged sub-trees are referenced not copied.
3. Index — git's staging area — `.git/index`, a binary file listing paths, modes, blob ids and stat data for the next commit.
4. Merkle DAG — a graph where every node names its children by hash — a directed acyclic graph with hash-derived edges, giving whole-graph integrity.
5. diff-tree — the command that compares two snapshots — plumbing that walks two trees and reports entries whose object ids differ.

38.5 Why a commit points at its parent

38.5.1 PLAIN - in simple words

1. Every commit except the very first one records the hash of the commit that came before it.
2. That single line, `parent <hash>`, is what turns a pile of snapshots into a history.
3. Follow the parent links backwards and you reach the beginning. That is what `git log` does.
4. It is not a line, though. It is a **graph**. A commit can have two parents, which is what a merge is. And two different commits can share one parent, which is what a branch is.
5. It is **acyclic**, meaning you can never walk forwards and end up where you started. That is guaranteed by the hashes, not by a rule someone enforces.
6. Because a commit's hash covers its parent's hash, and that parent's hash covers its own parent, one hash at the tip covers the entire history behind it.
7. Change anything anywhere in the past and every hash from that point forward changes. History is therefore tamper-evident.
8. Tamper-evident is not the same as tamper-proof. Anyone can rewrite history and hand you the new version. They just cannot do it invisibly if you already know the old hash.

38.5.2 PLAIN - a picture in your head

1. Think of a family of numbered wax seals.
2. Each seal is made by melting together the day's document and a copy of yesterday's seal.
3. So today's seal depends on yesterday's, which depended on the day before, all the way to the first day.
4. If somebody quietly edits a document from three years ago, they have to make a new seal for that day. That changes the next day's seal, and every seal after it.
5. You hold only the most recent seal in your hand. That one seal proves the whole chain.

Where this comparison breaks:

6. A wax seal can be forged by someone with the same stamp. A hash cannot be forged without solving arithmetic nobody can currently solve.
7. But the comparison also flatters git. Your seal only proves things if you wrote down what the seal looked like. If you never recorded it, someone can hand you a completely different chain of seals and you cannot tell.
8. And unlike a family seal, git history can legitimately fork. Two people can both build on yesterday's seal and both be right.

38.5.3 PLAIN – a worked example

1. Here is a real history from the sandbox with a real merge, drawn by git itself:

```
$ git log --graph --oneline --all
*   7fc6cf8 Merge feature into main
|\
| * 7c7506f Fourth commit: add docs/feature.md on feature
* | 25478c1 Fifth commit: extend README on main
|/
* b53a9f4 Third commit: add src/util.c
* fb17f98 Second commit: edit docs/intro.md
* d9cbb74 First commit: readme, source and docs
```

2. That is a diamond, not a line. Two commits both have `b53a9f4` as parent, and one commit has both of them as parents.
3. Look inside the merge commit. It has two `parent` lines:

```
$ git cat-file -p 7fc6cf8
tree b704d3e37ec1cb98ac5712b56206dbe28b5aac6e
parent 25478c16e00d681e80a0db23a6c7bca204e07404
parent 7c7506f6a3b659163ac6f5d9f155a1a9b89f298e
author KedByte Reader <reader@example.com> 1767681000 +0530
committer KedByte Reader <reader@example.com> 1767681000 +0530

Merge feature into main
```

4. The order of the parent lines matters and is not arbitrary. The first parent is the branch you were on. The second is the branch you merged in.
5. That is what the ^ numbers select:

```
$ git rev-parse HEAD^1
25478c16e00d681e80a0db23a6c7bca204e07404
$ git rev-parse HEAD^2
7c7506f6a3b659163ac6f5d9f155a1a9b89f298e
```

6. And the very first commit has no parent line at all. Compare:

```
$ git cat-file -p d9cbb74
tree b39d894c54aa6e9c3c2545eef059431d7b208ebf
author KedByte Reader <reader@example.com> 1767587400 +0530
committer KedByte Reader <reader@example.com> 1767587400 +0530

First commit: readme, source and docs
```

7. No `parent` line. That is a **root commit**.

38.5.4 PLAIN – what is really happening inside

1. The word "ancestor" has an exact meaning here. Commit A is an ancestor of commit B if you can get from B to A by following parent links, in any number of steps.
2. "Descendant" is the same relation read the other way.
3. Merging looks for the **merge base**, the closest commit that is an ancestor of both sides. In the example above that is `b53a9f4`.
4. Immutability is not a rule that git enforces with a lock. It is a consequence of naming.
5. If you want to change a commit's message, git cannot edit the object, because the object's name is derived from its bytes including the message. Changing the message produces a different object with a different name.

6. So `git commit --amend` does not amend anything. It builds a brand new commit object and moves the branch file to point at it. The old object is still on disk.
7. The same is true of rebase, of `git filter-repo`, and of every other "history rewriting" tool. They all create new objects.
8. The sealing works like this. Suppose an attacker wants to change one character in the first commit's README.
9. That changes the README blob's hash. That changes the root tree's hash. That changes commit 1's hash. Commit 2 contains commit 1's hash, so commit 2's hash changes. And so on to the tip.
10. If you wrote down the tip hash yesterday and it still matches today, then every byte of every file in every commit behind it is unchanged.

38.5.5 TECHNICAL - the engineer's version

1. Git history is a directed acyclic graph, a DAG, with commits as vertices and parent references as edges. Merge commits have out-degree 2 or more.
2. Octopus merges with three or more parents are legal. The default merge strategy in modern git, `ort`, refuses to resolve conflicts in an octopus merge, so they are rare in practice.
3. Acyclicity is structural. A commit's id is a hash over content that includes its parents' ids, so creating a cycle would require finding a hash preimage, which is why nobody argues about whether cycles are possible.
4. `git merge-base A B` computes the lowest common ancestor. With criss-cross histories there can be several, and the `ort` strategy handles that by recursively merging the candidate bases.
5. Topological order is a partial order, not a total one. `git log` defaults to showing commits in reverse commit-date order, which can mislead when clocks disagree. Use `git log --topo-order` when the shape matters.
6. Signed commits. A commit signature is stored as a `gpgsig` header inside the commit object itself, so the signature is covered by the commit hash. Here is a real one, created and verified in the sandbox with an SSH key:

```
$ git cat-file -p HEAD
tree 4588a5c622c7b494a501343b782d2d928c5e447f
author KedByte Reader <reader@example.com> 1767846600 +0530
committer KedByte Reader <reader@example.com> 1767846600 +0530
gpgsig -----BEGIN SSH SIGNATURE-----
 U1NIU0lHAAAAAQAAADMAAAALc3NoLWVkMjU1MTkAAAAgrLzsfF...
 ...
 -----END SSH SIGNATURE-----

A signed commit
```

7. The base64 lines are trimmed here to fit the page. The unsigned commit object was 208 bytes; the signed one is 492 bytes.
8. Verification, again real output:

```
$ git verify-commit HEAD
Good "git" signature for reader@example.com with ED25519 key
SHA256:32dP45eSMmVSt/G/CGvcxl/P+MO3Nwj9xeTh/GSA2wc
```

9. What a signature adds, precisely: it binds a key to that exact commit hash, and therefore to the whole history behind it. It answers "who asserted this tip", which the hash alone cannot.
10. What it does not add: it does not stop rewriting. An attacker with commit access rewrites history and signs the new version with their own key. Verification then fails only if you check the key against a list you trust.

11. This was visible in the sandbox. Before an allowed-signers file was configured, git reported the signature as cryptographically good but with "No principal matched". Good maths, unknown signer. Those are different questions and git keeps them separate.

Question	Answered by
Are the bytes unchanged?	the object hash
Is the history unchanged?	the tip commit hash
Who asserted this tip?	a valid signature
Should I trust that key?	your allowed-signers list

12. The honest version: git's tamper-evidence protects you against silent modification of a repository you already have a hash for. It does not tell you whether the history you were handed in the first place is the real one.

38.5.6 WORDS - remember these

1. Parent — the commit that came before — an object id recorded in a `parent` header inside the commit object.
2. Root commit — a commit with no parent — a commit object with zero `parent` headers; a repository may have several.
3. Merge commit — a commit joining two lines of work — a commit object with two or more `parent` headers, first parent being the branch merged into.
4. Ancestor — reachable by walking parents backwards — the transitive closure of the parent relation.
5. Merge base — the closest shared ancestor of two commits — the lowest common ancestor in the commit DAG, computed by `git merge-base`.
6. Tamper-evident — changes cannot be hidden — any modification alters the object id and propagates to every descendant hash.

38.6 What a branch really is

38.6.1 PLAIN - in simple words

1. A branch is a file. That is the whole answer.
2. The file lives at `.git/refs/heads/` followed by the branch name.
3. Inside that file are 40 hexadecimal characters and a newline. Forty-one bytes.
4. Those 40 characters are the hash of one commit: the tip of the branch.
5. A branch does not contain commits. It points at one commit. The commits it "contains" are simply the ones you reach by walking parents backwards from there.
6. Creating a branch writes one 41-byte file. Nothing is copied. No files are duplicated. That is why branching is nearly free.
7. Deleting a branch deletes one 41-byte file. The commits are untouched.
8. Committing means writing a new commit object and then overwriting those 41 bytes with the new hash.

38.6.2 PLAIN - a picture in your head

1. Think of a long shelf of books, in order, each book naming the one before it.
2. A branch is not a copy of the shelf. It is a sticky note on the spine of one book saying "main".
3. Adding a branch means writing one more sticky note. It takes a second and costs one note.
4. Moving a branch forward means peeling the note off one book and putting it on the next.
5. Removing a branch means throwing away a sticky note. Every book is still on the shelf.

Where this comparison breaks:

6. A sticky note can fall off and be lost. Git writes the move into a log, the reflog, so you can put it back. Section 38.9 covers that.
7. And a shelf is one physical thing. In git, two people can have sticky notes with the same name on completely different books, and neither is wrong until they try to agree.

38.6.3 PLAIN – a worked example

1. Real output from a fresh sandbox repository with two commits.

```
$ ls -l .git/refs/heads/
-rw-r--r-- 1 root root 41 Aug 13 03:18 main
$ cat .git/refs/heads/main
bc7fc422aec1054a108922e49253ce8271b1e957
```

2. Forty-one bytes. Forty hexadecimal characters and a newline, confirmed by dumping the raw bytes:

```
0000040   7   1   b   1   e   9   5   7  \n
```

3. Now create two more branches, one at the tip and one at the previous commit:

```
$ git branch topic
$ git branch spike HEAD~1
$ ls -l .git/refs/heads/
-rw-r--r-- 1 root root 41 main
-rw-r--r-- 1 root root 41 spike
-rw-r--r-- 1 root root 41 topic
```

4. Three files, 41 bytes each, 123 bytes in total. Their contents:

```
main   bc7fc422aec1054a108922e49253ce8271b1e957
topic  bc7fc422aec1054a108922e49253ce8271b1e957
spike  1dfe2f9b9521c9a46513a095c135831a8b54ef01
```

5. `main` and `topic` hold the same 40 characters. They are two names for one commit, and the repository is not one byte larger for it.
6. Deleting a branch does not delete the commit:

```
$ git branch -D spike
Deleted branch spike (was 1dfe2f9).
$ git cat-file -t 1dfe2f9
commit
```

7. The commit is still there, still readable, still has its tree and its blobs. Only the pointer went.
8. You can move a branch by hand, because it really is just a file:

```
$ git update-ref refs/heads/topic 1dfe2f9b9521c9a4...
$ git branch -v
* main  0f0c441 three
  topic 1dfe2f9 one
```

38.6.4 PLAIN – what is really happening inside

1. Sometimes you look for `.git/refs/heads/main` and it is not there. The branch still exists. It has simply been packed.
2. Git can move all its ref files into one file called `.git/packed-refs`, to avoid thousands of tiny files.
3. Watch it happen for real:

```
$ git pack-refs --all
$ ls .git/refs/heads/
(empty)
$ cat .git/packed-refs
```

```
# pack-refs with: peeled fully-peeled sorted
bc7fc422aec1054a108922e49253ce8271b1e957 refs/heads/main
1dfe2f9b9521c9a46513a095c135831a8b54ef01 refs/heads/spike
bc7fc422aec1054a108922e49253ce8271b1e957 refs/heads/topic
```

4. The directory is empty and everything still works, because git looks in both places.
5. The rule is: a loose file wins over the packed file. Commit once more and the loose file comes back for that branch only:

```
$ git commit -m "three"
$ cat .git/refs/heads/main
0f0c4412ec5e57f084c7a39a254a1f9d67f3e965
$ grep main .git/packed-refs
bc7fc422aec1054a108922e49253ce8271b1e957 refs/heads/main
```

6. Two different values for `refs/heads/main` exist on disk at once, and that is fine. The loose file is the current one. The packed entry is stale and will be tidied later.
7. This is why you should never read ref files yourself in a script. Ask git: `git rev-parse main`, or `git show-ref`, or `git for-each-ref`.
8. Contrast with a system where a branch is a copy. If branching meant copying the working tree, then in the sandbox's 200-commit repository each branch would cost 109,408 bytes of file content. A thousand branches would be about 104 MiB.
9. In git, a thousand branches cost 41,000 bytes of actual content.

38.6.5 TECHNICAL – the engineer's version

1. A ref is a name in a hierarchical namespace, `refs/heads/*` for local branches, that resolves to an object id.
2. Storage back-ends, as of Git 2.43: loose files under `.git/refs`, plus `.git/packed-refs`. A newer `reftable` back-end exists as an option in recent releases and is not the default.
3. Measured in the sandbox on a 200-commit repository:

Operation	Measured cost
Create 1,000 branches	2.28 s, 41,000 bytes
Disk blocks for those refs	4.0 MB (4 KiB per file)
After `git pack-refs --all`	one file of 56,996 bytes
Disk blocks after packing	28 KB

4. Note the gap between 41,000 bytes of content and 4.0 MB of disk. Each 41-byte file still occupies a whole filesystem block. That is the real reason `packed-refs` exists, and why repositories with tens of thousands of refs feel slow before it is used.
5. Ref updates are made atomic by writing a temporary file and renaming it over the target, and by taking a `.lock` file. This is why two concurrent commits in the same repository cannot half-write a branch.
6. Historical contrast, and it is worth knowing why git felt revolutionary in

2005. In CVS, released in 1990, a branch was recorded inside every single file's history, so branching a large tree touched every file. In Subversion, released in 2000, a branch is a cheap server-side copy, but it appears as a real directory path and switching a working copy is not free.

7. Git's design goal, stated by Linus Torvalds when he began the project on 3 April 2005 after the BitKeeper licence withdrawal, was that branching and merging should be cheap enough to do casually. Writing 41 bytes achieves that.
8. Convention, not standard: the default branch name. Git shipped `master` as the default from 2005. GitHub changed the default for new repositories to `main` in October 2020, and git itself added `init.defaultBranch` so the choice is configurable. Neither name means anything to the software.

38.6.6 WORDS - remember these

1. Ref — a name that points at a commit — an entry in the `refs/` namespace resolving to an object id.
2. Branch — a moving pointer to the tip of a line of work — a ref under `refs/heads/` that is advanced automatically when you commit.
3. packed-refs — one file holding many refs — `.git/packed-refs`, a sorted text file consulted when no loose ref file exists.
4. Loose ref — a ref stored as its own small file — a 41-byte file under `.git/refs/`, which takes precedence over the packed entry.
5. update-ref — the command that sets a ref by hand — plumbing that writes a ref value with locking and reflog recording.

38.7 HEAD

38.7.1 PLAIN - in simple words

1. `HEAD` is another file, at `.git/HEAD`, and it answers one question: where am I right now.
2. Normally it does not contain a commit hash. It contains the name of a branch.
3. Its whole content is the text `ref: refs/heads/main` and a newline. Twenty-one bytes.
4. That is called **attached** HEAD. You are "on" a branch. When you commit, git writes the new commit and then moves that branch forward.
5. Sometimes `HEAD` contains a commit hash directly instead of a branch name. That is **detached** HEAD.
6. Detached means you are looking at a commit without being on any branch. You can look, build, test, even commit.
7. It is not an error. It is a normal state. Git even prints a helpful notice when you enter it.
8. The danger is only this: if you commit while detached and then walk away, nothing points at your new commit, so it is easy to lose track of it.

38.7.2 PLAIN - a picture in your head

1. Think of the shelf of books again, with sticky notes for branch names.
2. `HEAD` is your finger. Normally your finger rests on a sticky note, not on a book. When the note moves, your finger goes with it.
3. Detached HEAD is when you take your finger off the notes and put it directly on a book.
4. You can still read that book, and you can still write new books after it. But no sticky note follows you.
5. If you then move your finger back to a note, the books you wrote are still on the shelf, but with nothing labelling them, they are easy to overlook.

Where this comparison breaks:

6. On a real shelf you would still see the new books. In git nothing is visible in `git branch` or `git log` unless something points at it.
7. But git is kinder than the shelf: it writes down where your finger has been. That log is the reflog, and it is how you find those books again.

38.7.3 PLAIN - a worked example

1. Attached state, real output:

```
$ cat .git/HEAD
ref: refs/heads/main
$ git symbolic-ref HEAD
refs/heads/main
$ git status -sb
## main
```

2. Now detach on purpose. Git prints a long, genuinely useful notice:

```
$ git checkout HEAD~2
Note: switching to 'HEAD~2'.

You are in 'detached HEAD' state. You can look around, make
experimental changes and commit them, and you can discard any
commits you make in this state without impacting any branches
by switching back to a branch.

If you want to create a new branch to retain commits you create,
you may do so (now or later) by using -c with the switch command.
Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

HEAD is now at b53a9f4 Third commit: add src/util.c
```

3. Look at the file now. No branch name. A raw hash:

```
$ cat .git/HEAD
b53a9f46d432aa53e687328841c6dd2b89963809
$ git symbolic-ref HEAD
fatal: ref HEAD is not a symbolic ref
$ git status -sb
## HEAD (no branch)
```

4. Commit something while detached and `HEAD` simply advances by itself:

```
$ git commit -m "Work done while detached"
$ git rev-parse HEAD
3a0a3d62188c8a51d4d914b5efd000ea8483695b
$ cat .git/HEAD
3a0a3d62188c8a51d4d914b5efd000ea8483695b
```

5. No branch moved, because no branch was involved.
6. Now leave. Git warns you, by name and by hash:

```
$ git checkout main
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

  3a0a3d6 Work done while detached

If you want to keep it by creating a new branch, this may be a
good time to do so with:

 git branch <new-branch-name> 3a0a3d6

Switched to branch 'main'
```

7. That warning contains everything you need to recover. Copy the hash and run `git branch keep-this 3a0a3d6`.

38.7.4 PLAIN - what is really happening inside

1. `HEAD` is a **symbolic reference**: a ref whose value is the name of another ref rather than an object id.
2. When git needs "the current commit", it reads `HEAD`. If the content starts with `ref:`, it follows that name and reads the branch file. Otherwise it uses the hash directly.
3. That one indirection is why committing on a branch moves the branch. Git updates whatever `HEAD` points to.
4. How you end up detached, in practice:
    1. `git checkout <sha>` or `git checkout <tag>` or `git checkout origin/main`.
    2. `git rebase` while it is replaying commits.
    3. A continuous integration job doing `git checkout $COMMIT_SHA`, which is why almost every CI build runs detached.
    4. `git bisect`, which walks you across commits.
5. How to recover, in order of preference:
    1. If you have not committed: `git switch -` or `git checkout main`. Nothing is at risk.
    2. If you have committed and want to keep it: `git branch <name>` right there, or `git switch -c <name>`.
    3. If you already left and lost the hash: use the reflog, section 38.9.
6. The commits you made while detached are ordinary objects. They are not marked, not special, not fragile. They are simply unreferenced, and unreferenced objects are eventually swept up by garbage collection.

38.7.5 TECHNICAL - the engineer's version

1. `.git/HEAD` is 21 bytes in the attached state and 41 bytes in the detached state. Both are plain text.
2. The plumbing commands are `git symbolic-ref HEAD` to read or write the symbolic form, and `git rev-parse HEAD` to resolve either form to an object id.
3. `git rev-parse --abbrev-ref HEAD` prints the branch name when attached and the literal string `HEAD` when detached. Scripts use this to decide.
4. Other pseudo-refs live in `.git/` alongside HEAD and appear during operations. All of these were observed in the sandbox:

Pseudo-ref	Present during / after
`ORIG_HEAD`	reset, merge, rebase
`FETCH_HEAD`	after `git fetch`
`MERGE_HEAD`	an unfinished merge
`CHERRY_PICK_HEAD`	an unfinished cherry-pick
`AUTO_MERGE`	ort strategy's merged tree

5. `advice.detachedHead` controls the long notice. Setting it to `false` silences it, which is standard practice in CI images where every checkout is detached by design.
6. A worktree created with `git worktree add` gets its own `HEAD`, stored under `.git/worktrees/<name>/HEAD`. That is why two worktrees of one repository can sit on different commits while sharing one object store.
7. The honest version: people describe detached HEAD as dangerous. It is not. It is exactly as safe as any commit that no branch points at. The risk is purely about visibility, and the reflog removes most of that risk.

38.7.6 WORDS - remember these

1. HEAD — the file saying where you are — a pseudo-ref at `.git/HEAD`, usually symbolic.
2. Symbolic ref — a ref pointing at another ref by name — content of the form `ref: refs/heads/<name>`.

3. Attached HEAD — you are on a branch — `HEAD` is symbolic and commits advance the named branch.
4. Detached HEAD — you are on a commit with no branch — `HEAD` holds an object id directly; new commits are unreferenced.
5. Pseudo-ref — a special one-off ref in the git directory — names such as `ORIG_HEAD` and `MERGE_HEAD`, written by particular operations.

38.8 The rest of the refs

38.8.1 PLAIN - in simple words

1. Branches are only one kind of ref. All refs live in the same namespace and all of them work the same way: a name that resolves to an object id.
2. `refs/heads/` holds your local branches.
3. `refs/tags/` holds tags. There are two kinds and they differ on disk.
4. A **lightweight tag** is just a ref file pointing straight at a commit. No extra object exists.
5. An **annotated tag** points at a tag object, and that tag object points at the commit. It carries a tagger, a date, a message and, optionally, a signature.
6. `refs/remotes/` holds remote-tracking refs, such as `origin/main`. These are your local cached memory of what a remote looked like the last time you spoke to it.
7. `refs/notes/` holds notes, a way of attaching text to a commit without changing the commit.
8. `refs/stash` holds stashed work. There are a few other special names too.

38.8.2 PLAIN - a picture in your head

1. Think of the shelf of books once more, and the sticky notes.
2. A branch note is one you move forward every day.
3. A lightweight tag is a note you stick on and never move.
4. An annotated tag is not a note at all. It is a small printed card in a sleeve, with a date, a signature and a sentence explaining why this book matters, and the card names the book.
5. A remote-tracking note is a note in a different colour saying "when I last visited the other library, their note called `main` was on this book".
6. It does not move when their note moves. It moves when you next visit.

Where this comparison breaks:

7. On a shelf you could check the other library instantly by looking across the room. In git, checking costs a network round trip, which is exactly why the cached note exists and exactly why it goes stale.

38.8.3 PLAIN - a worked example

1. The complete ref tree of the sandbox repository, exactly as git reports it:

```
$ git for-each-ref --format='%(objectname:short) %(objecttype) %(refname)'
7c7506f commit refs/heads/feature
70a5d62 commit refs/heads/fix
39bf0c8 commit refs/heads/main
cc1849e commit refs/notes/commits
b53a9f4 commit refs/remotes/origin/main
b53a9f4 commit refs/tags/v1-light
d54ba18 tag    refs/tags/v1.0
```

2. Look at the last two rows. Both are tags, both point at the same release, but the object types differ.
3. `refs/tags/v1-light` has type `commit`. The ref points at the commit directly. That is a lightweight tag.
4. `refs/tags/v1.0` has type `tag`. The ref points at a tag object, which in turn points at the commit. That is an annotated tag.
5. On disk, both are 41-byte files with 40 hex characters. The difference is what those characters name:

```
.git/refs/tags/v1-light -> b53a9f46...  (a commit)
.git/refs/tags/v1.0     -> d54ba182...  (a tag object)
```

6. And the tag object itself, which the lightweight tag simply does not have:

```
$ git cat-file -p v1.0
object b53a9f46d432aa53e687328841c6dd2b89963809
type commit
tag v1.0
tagger KedByte Reader <reader@example.com> 1767598200 +0530

First release of the demo
```

38.8.4 PLAIN – what is really happening inside

1. Remote-tracking refs are updated by exactly two things: `git fetch` and `git push`. Nothing else touches them.
2. So `origin/main` is not a live reading of the server. It is a note of what the server said last time. This matters enormously and section 38.12 returns to it.
3. `git ls-remote` is the command that asks the server right now, over the network, without changing anything locally:

```
$ git ls-remote origin
b53a9f46d432aa53e687328841c6dd2b89963809    HEAD
b53a9f46d432aa53e687328841c6dd2b89963809    refs/heads/main
b53a9f46d432aa53e687328841c6dd2b89963809    refs/tags/v1-light
d54ba1826a59225bc14a2b55e832536cb7a8e4ac    refs/tags/v1.0
b53a9f46d432aa53e687328841c6dd2b89963809    refs/tags/v1.0^{}
```

4. Notice the last line. The suffix `^{}` means "the object this tag finally resolves to after following it through". Servers advertise both, so clients need not fetch the tag object just to learn what it points at.
5. The `packed-refs` file records the same fact with a caret line:

```
d54ba1826a59225bc14a2b55e832536cb7a8e4ac refs/tags/v1.0
^b53a9f46d432aa53e687328841c6dd2b89963809
```

6. A note is stored as a whole commit and tree of its own, under `refs/notes/commits`. The tree maps a commit's hash, used as a file name, to a blob holding the note text.
7. That is why notes can be added and changed without altering the commit they describe. The commit does not know about them.
8. A stash entry is a merge commit with two or three parents: the commit you were on, a commit holding the index state, and if needed one holding untracked files.

```
$ git cat-file -p refs/stash
tree a5419a1446acbb442a6da16df29e43c605ebd018
parent 39bf0c8354d383aaf21ff05eaff4304acc24f346
parent 96a1b317649d3354ec613c5f61bcc7ba7eba40c1
```

9. So a stash is not a special storage area. It is ordinary commits with a ref pointing at them.

38.8.5 TECHNICAL – the engineer's version

1. The ref namespace, with the parts you will actually meet:

Namespace	Holds
`refs/heads/*`	local branches

Namespace	Holds
`refs/tags/*`	tags, both kinds
`refs/remotes/<r>/*`	remote-tracking refs
`refs/notes/*`	note trees

2. Beyond those: `refs/stash`, `refs/bisect/*` during a bisect, `refs/replace/*` for object replacement, and on GitHub the server-side `refs/pull/<n>/head` and `refs/pull/<n>/merge`, which are a hosting convention rather than part of git.
3. Lightweight versus annotated is a real operational difference, not a style choice:

Aspect	Lightweight	Annotated
Extra object	none	one tag object
Has message	no	yes
Has tagger and date	no	yes
Can be signed	no	yes
`describe` prefers it	no	yes

4. `git describe` ignores lightweight tags unless you pass `--tags`. Release engineering therefore uses annotated tags almost universally.
5. The default fetch refspec, written into `.git/config` by `git clone`, is `+refs/heads/*:refs/remotes/origin/*`. The leading plus means non-fast-forward updates are allowed for these refs, which is why a force-push upstream quietly rewrites your `origin/*` refs.
6. `git for-each-ref` is the scriptable interface. `git show-ref` is the older one. Both read loose and packed refs correctly, which hand-written scripts that `cat` ref files do not.

38.8.6 WORDS - remember these

1. Lightweight tag — a fixed pointer to a commit — a ref under `refs/tags/` whose value is a commit id.
2. Annotated tag — a tag with a message and author — a ref pointing at a `tag` object carrying tagger, date, message and optional signature.
3. Remote-tracking ref — your cached memory of a remote branch — a ref under `refs/remotes/`, updated only by fetch and push.
4. Refspec — the rule mapping their refs to yours — for example `+refs/heads/*:refs/remotes/origin/*`.
5. Peeled ref — the object a tag finally points to — shown as `<tag>^{}` and recorded with a caret line in `packed-refs`.
6. Note — text attached to a commit after the fact — a blob in a tree under `refs/notes/`, keyed by the commit's object id.

38.9 The reflog

38.9.1 PLAIN - in simple words

1. Every time a ref changes value, git writes a line in a log saying where it was and where it went.
2. That log is the **reflog**. There is one for `HEAD` and one for each branch.
3. It records the old hash, the new hash, who did it, when, and which command caused it.
4. This log is the reason "I destroyed my work with git" is almost always false.
5. A hard reset, a bad rebase, a deleted branch, an amended commit: in every case the old commit object is still on disk and the reflog still knows its hash.
6. The reflog is local and private. It is never pushed, never fetched, and never shared. Cloning a repository does not bring its reflog.

7. Entries do not last forever. By default reachable entries survive 90 days and unreachable ones 30 days.

38.9.2 PLAIN – a picture in your head

1. Think of a security camera pointed at the sticky notes on the shelf.
2. It does not record the books. It records every time a note moved, and to where.
3. If you come back and find a note in the wrong place, you rewind the tape, read off which book it used to be on, and put it back.
4. The books never went anywhere. Only the note moved. That is the whole trick.

Where this comparison breaks:

5. A camera records what everyone did. The reflog records only what happened in your copy, on your machine. Your colleague's reflog is a different tape.
6. And the tape is wiped on a schedule. After about 30 days, a commit that nothing points at can genuinely be collected and deleted.

38.9.3 PLAIN – a worked example

1. Here is a real disaster and a real recovery, done in the sandbox. The starting point is a repository whose `main` is at a merge commit.

```
$ git log --oneline -3
7fc6cf8 Merge feature into main
25478c1 Fifth commit: extend README on main
7c7506f Fourth commit: add docs/feature.md on feature
```

2. Now the mistake. A hard reset two commits back, which throws away the merge and everything after it:

```
$ git reset --hard HEAD~2
HEAD is now at b53a9f4 Third commit: add src/util.c
$ git log --oneline -3
b53a9f4 Third commit: add src/util.c
fb17f98 Second commit: edit docs/intro.md
d9cbb74 First commit: readme, source and docs
$ ls
README.md  docs  src
```

3. The merge is gone from the log. Files that only existed after it are gone from the working directory. This is the moment people panic.
4. It is not gone. The commit object is still there and still readable:

```
$ git cat-file -t 7fc6cf8
commit
$ git cat-file -p 7fc6cf8 | head -3
tree b704d3e37ec1cb98ac5712b56206dbe28b5aac6e
parent 25478c16e00d681e80a0db23a6c7bca204e07404
parent 7c7506f6a3b659163ac6f5d9f155a1a9b89f298e
```

5. And even without knowing that hash, the reflog tells you it:

```
$ git reflog
b53a9f4 HEAD@{0}: reset: moving to HEAD~2
7fc6cf8 HEAD@{1}: checkout: moving from 3a0a3d6... to main
3a0a3d6 HEAD@{2}: commit: Work done while detached
```

6. `HEAD@{1}` is where `HEAD` was one move ago: `7fc6cf8`. That is the merge.
7. The branch has its own reflog, which is usually the cleaner one to read:

```
$ git reflog show main
7fc6cf8 main@{0}: merge feature: Merge made by the 'ort' strategy
25478c1 main@{1}: commit: Fifth commit: extend README on main
b53a9f4 main@{2}: commit: Third commit: add src/util.c
```

8. The recovery is one command:

```
$ git rev-parse main@{1}
7fc6cf85594a850b727418867d4054af04412e88
$ git reset --hard main@{1}
HEAD is now at 7fc6cf8 Merge feature into main
$ git log --oneline -3
7fc6cf8 Merge feature into main
25478c1 Fifth commit: extend README on main
7c7506f Fourth commit: add docs/feature.md on feature
```

9. Everything is back. Total elapsed effort: one lookup and one command.

38.9.4 PLAIN - what is really happening inside

1. The reflog is a plain text file per ref. `HEAD`'s log is `.git/logs/HEAD`. A branch's log is `.git/logs/refs/heads/<name>`.
2. Each line has five fields, tab-separated at the end: old object id, new object id, the person, the timestamp with offset, then the reason.
3. Here is one real line, wrapped across two lines to fit this page:

```
fb17f98cd37f546dfedb2f91e1b7a9762bf45d17 b53a9f46d432aa53e6873
28841c6dd2b89963809 KedByte Reader <reader@example.com>
1767594600 +0530    commit: Third commit: add src/util.c
```

4. The very first line of a branch's log has all zeros as the old id, because the branch did not exist before:

```
0000000000000000000000000000000000000000 d9cbb74f1fb9bce240d20d
b0c8e545caf68c802b ... commit (initial): First commit ...
```

5. The `@{n}` syntax counts entries in that file. `main@{0}` is the current value, `main@{1}` is the value before the most recent change, and so on.
6. There is also a time form. `main@{2026-01-06}` asks what `main` pointed at on that date. If the log does not go back that far, git says so:

```
warning: log for 'main' only goes back to Thu, 13 Aug 2026 03:12:55
```

7. Notice `@{n}` and `~n` are completely different questions. `HEAD~1` means "the parent commit in the history". `HEAD@{1}` means "wherever HEAD was pointing one move ago". They frequently give different answers.
8. Reflogs are only kept if `core.logAllRefUpdates` is on, which it is by default for non-bare repositories. Bare repositories, the kind that sit on a server, do not keep reflogs by default. That is worth remembering before you rely on one.

38.9.5 TECHNICAL - the engineer's version

1. Expiry defaults, from git's own documentation:

Setting	Default
`gc.reflogExpire`	90 days
`gc.reflogExpireUnreachable`	30 days
`gc.pruneExpire`	2 weeks ago

Setting	Default
`gc.auto`	6700 loose objects
`gc.autoPackLimit`	50 packs

2. "Unreachable" here means the reflog entry names a commit that is no longer reachable from any ref. Those are the entries you most need after a mistake, and they are the ones that expire soonest.
3. Reflog entries keep their objects alive. `git gc` treats reflog-referenced objects as reachable, which is why a commit lost by reset is not collected the same afternoon.
4. Proof from the sandbox. Immediately after abandoning a detached commit, `git fsck --unreachable` reported nothing, because the reflog still held it. Adding `--no-reflogs` revealed the truth:

```
$ git fsck --unreachable --no-reflogs
unreachable tree bcb2c24a564e8ff3d64369f03559123b68b04b03
unreachable blob 9a5abed11a92dda94398c590ec782181a789f4c5
unreachable commit 3a0a3d62188c8a51d4d914b5efd000ea8483695b
```

5. The recovery toolkit, in the order you should reach for it:
   1. `git reflog` or `git reflog show <branch>` — find the old hash.
   2. `git reset --hard <sha>` — move the branch back, discarding current work.
   3. `git branch rescue <sha>` — safer: give the lost commit a name without touching your current branch.
   4. `git fsck --lost-found` — last resort, when the reflog was pruned or the repository is bare. Writes dangling objects into `.git/lost-found/`.
6. `git log -g` prints the reflog as commits, with full messages, which is often easier to read than `git reflog` when you need to identify work by content.
7. Danger commands and what saves you from each:

Command	Reflog entry that saves you
`git reset --hard X`	`HEAD@{1}` and `<branch>@{1}`
`git rebase`	`<branch>@{1}`, before the rebase
`git commit --amend`	`HEAD@{1}` holds the pre-amend commit
`git branch -D X`	the deletion message prints the hash
`git checkout` detach	the "leaving N commits behind" warning

8. What the reflog does not save: uncommitted changes in the working tree. `git reset --hard`, `git checkout -- <file>` and `git clean -fd` destroy work that was never turned into an object, and no log exists for it. Anything you staged with `git add` survives, as a dangling blob findable with `git fsck --lost-found`.

38.9.6 WORDS - remember these

1. Reflog — a local log of where each ref has been — per-ref append-only log in `.git/logs/`, recording old id, new id, actor, time and reason.
2. `@{n}` — the value a ref had n moves ago — reflog index selector, distinct from the `~n` ancestry selector.
3. Unreachable object — an object no ref points at — a candidate for pruning once its reflog entry expires.
4. Dangling object — an unreachable object with nothing pointing at it at all — reported by `git fsck` and recoverable with `--lost-found`.
5. Prune — permanently delete unreachable objects — `git prune`, or the pruning step inside `git gc`, honouring `gc.pruneExpire`.

38.10 The commands that reveal the machinery

38.10.1 PLAIN – in simple words

1. Git has two layers of commands, and it is worth knowing which is which.
2. **Porcelain** commands are the ones you use daily: `add`, `commit`, `log`, `merge`, `status`, `push`. They are built for people. Their output is meant to read nicely and is allowed to change between versions.
3. **Plumbing** commands are the low-level ones underneath: `cat-file`, `hash-object`, `ls-tree`, `rev-parse`, `rev-list`, `update-ref`. They are built for scripts. Their output format is stable on purpose.
4. The names are git's own, from its documentation, and they come from the idea that you see the porcelain of a bathroom while the plumbing does the work.
5. Rule for scripts: never parse porcelain output. Use plumbing, or use the porcelain commands' `--porcelain` flags, which are stable despite the confusing name.
6. This section is a tour of the plumbing, with real output for each command.

38.10.2 PLAIN – a picture in your head

1. A car has a dashboard and it has an engine.
2. The dashboard tells you the speed in a form you can act on. The designer may redesign it next year and nobody minds.
3. The engine has bolts of exact sizes that have not changed in a decade, because tools depend on them.
4. Porcelain is the dashboard. Plumbing is the bolts.
5. You drive with the dashboard. You diagnose with the bolts.

Where this comparison breaks:

6. In a car you rarely need the engine. In git the plumbing is genuinely useful during ordinary work, especially when something has gone wrong and you need an answer that is exactly true rather than nicely presented.

38.10.3 PLAIN – a worked example

1. `git cat-file` with its three most useful flags. `-t` gives the type, `-s` the size in bytes, `-p` the content:

```
$ git cat-file -t b53a9f4
commit
$ git cat-file -s b53a9f4
247
$ git cat-file -t b53a9f4^{tree}
tree
```

2. `git rev-parse` turns any way of naming a commit into the one true hash:

```
$ git rev-parse HEAD
39bf0c8354d383aaf21ff05eaff4304acc24f346
$ git rev-parse --short HEAD
39bf0c8
$ git rev-parse --short=12 HEAD
39bf0c8354d3
$ git rev-parse --abbrev-ref HEAD
main
$ git rev-parse --git-dir
.git
$ git rev-parse --show-toplevel
/tmp/kb38/demo
```

3. It also resolves paths inside a commit, which is how you get a blob's hash without looking at a tree:

```
$ git rev-parse HEAD:docs/intro.md
bce6753328d0de8aaa42692aba92b99182ded5ce
```

4. `git hash-object` computes a name without storing anything, and with `-w` stores it:

```
$ printf 'Introduction.\n' | git hash-object --stdin
0e2a325c7de3369b164d48fcba823bca9c2cec4e
```

5. `git ls-tree` reads a tree. Plain, recursive, and with sizes:

```
$ git ls-tree HEAD
100644 blob 6af3a5ad4b99c155602b8f78298d89fc15b61209    README.md
040000 tree 8c33b40fe5eb264b8d43c66f01dc437cc4c90aa5    docs
040000 tree 069dcc578e4a416937da34a3dcc4d209c1048c63    src

$ git ls-tree -r HEAD
100644 blob 6af3a5ad... README.md
100644 blob 76c95066... docs/feature.md
100644 blob bce67533... docs/intro.md
100644 blob 61a98438... docs/mainnote.md
100644 blob 78f2de10... src/main.c
100644 blob df76ea63... src/util.c
```

6. `git count-objects -v` tells you the shape of the object store:

```
$ git count-objects -vH
count: 15
size: 60.00 KiB
in-pack: 31
packs: 1
size-pack: 4.60 KiB
```

7. `git rev-list` lists commit hashes, and counts them:

```
$ git rev-list --count HEAD
7
$ git rev-list --count --all
9
```

38.10.4 PLAIN – what is really happening inside

1. The revision syntax deserves its own explanation because two of its forms look alike and mean different things.
2. `X~n` means "go back n steps, always taking the first parent". `HEAD~2` is the grandparent.
3. `X^n` means "the n-th parent of this one commit". `HEAD^2` is the second parent, which only exists on a merge commit.
4. On the real merge commit in the sandbox:

```
$ git rev-parse HEAD      # the merge
7fc6cf85594a850b727418867d4054af04412e88
$ git rev-parse HEAD^1    # first parent, the branch merged into
25478c16e00d681e80a0db23a6c7bca204e07404
$ git rev-parse HEAD^2    # second parent, the branch merged in
7c7506f6a3b659163ac6f5d9f155a1a9b89f298e
$ git rev-parse HEAD~1    # same as ^1
25478c16e00d681e80a0db23a6c7bca204e07404
$ git rev-parse HEAD~2    # two first-parent steps back
b53a9f46d432aa53e687328841c6dd2b89963809
```

5. `HEAD~1` and `HEAD^1` gave the same answer. `HEAD~2` and `HEAD^2` gave completely different answers. That is the trap.
6. And `branch@{1}` is a third thing again: not ancestry at all, but the reflog. In the sandbox `main@{1}` was `7fc6cf8` while `main~1` was a different commit entirely.
7. `git show` is porcelain that stitches several plumbing operations together. Given a commit it prints the metadata plus a computed diff. Given a blob it prints content. Given a tree it prints a listing.
8. `git log --graph --oneline --all` is the single most useful shape-revealing command. `--all` is the important part: without it you only see history reachable from HEAD.

38.10.5 TECHNICAL - the engineer's version

1. Command classification, with the stability promise:

Command	Layer	Output stability
`git status`	porcelain	may change
`git status --porcelain=v2`	porcelain	stable, versioned
`git cat-file`	plumbing	stable
`git rev-parse`	plumbing	stable
`git for-each-ref`	plumbing	stable, formattable

2. `git cat-file --batch-check` is the high-throughput form. It reads object names on standard input and writes name, type and size, avoiding one process per object:

```
$ git cat-file --batch-all-objects --batch-check
0e2a325c7de3369b164d48fcba823bca9c2cec4e blob 14
6af3a5ad4b99c155602b8f78298d89fc15b61209 blob 52
78f2de106c92b0d60772bd5aa6c1e6da7bf71005 blob 29
```

3. `git verify-pack -v` lists a packfile's contents. Columns are: object id, type, size, size in the packfile, offset in the packfile, and for deltified objects the chain depth and the base object id:

```
5087b1a6efba7b591bda30d3badbd5056b4fab4b blob 111810 15280 30437
23841aba9acef22ead7d8505ba90653bc3544934 blob 78 93 45717 1 5087
6578910fb9573331428c33593424cb14b02cb49a blob 38 52 45810 2 2384
```

4. Read that carefully. The first blob's real size is 111,810 bytes and it occupies 15,280 bytes in the pack. The second's "size" column reads 78, but that is the size of the delta, not of the object. Depth 1 means one hop to its base.
5. `git rev-list` is the engine under `git log`. It walks the DAG and emits object ids. `--objects` also emits the trees and blobs reachable from each commit, which is exactly what `git pack-objects` consumes during a push.
6. `git fsck` verifies the whole store: every object's hash against its content, every referenced object present, every ref resolvable. It is the command that turns content addressing into a usable integrity check.
7. Useful one-liners worth memorizing:

```
git cat-file -p HEAD^{tree}            # top directory of HEAD
git rev-list --count main ^origin/main # commits not pushed yet
git for-each-ref --sort=-committerdate refs/heads/
git log --graph --oneline --all --decorate
git rev-parse --verify --quiet <sha>   # test existence, no output
```

38.10.6 WORDS - remember these

1. Porcelain — the commands people use — the human-facing layer whose output format carries no stability promise.

2. Plumbing — the commands scripts use — the low-level layer with stable, machine-readable output.
3. `rev-parse` — turn a name into a hash — the revision-parsing plumbing that resolves every form of object naming.
4. ~ and ^ — steps back, versus which parent — `~n` is n first-parent steps, `^n` selects the n-th parent of one commit.
5. `--batch-check` — ask about many objects at once — `git cat-file` batch mode giving id, type and size without one process per object.

38.11 How git stores things efficiently

38.11.1 PLAIN - in simple words

1. Everything so far has described whole snapshots. That sounds wasteful, and on its own it is.
2. Git solves this in two stages.
3. Stage one, **loose objects**. Each object gets its own file, squeezed with zlib compression. Simple, fast to write, wasteful of disk blocks.
4. Stage two, **packfiles**. Git gathers thousands of objects into one big file and, inside that file, stores similar objects as differences from one another.
5. This is the important sentence of the whole section: those differences are a storage trick only. Git's idea of history is still whole snapshots. The deltas exist below that, and you never see them.
6. Git does the packing when you push, when you clone, and when housekeeping runs.
7. `git gc` is the command that does the housekeeping: pack loose objects, pack refs, trim reflogs, and delete objects nothing points at any more.

38.11.2 PLAIN - a picture in your head

1. Imagine keeping every draft of a long report in a filing cabinet, one folder per draft.
2. At first you photocopy the whole report each time. Two hundred drafts, two hundred thick folders. That is loose objects.
3. Then a clerk goes through the cabinet. She keeps one full copy of the report, and replaces the other 199 folders with slips of paper saying "same as the previous draft, but line 812 now reads this".
4. The cabinet drawer that was full is now nearly empty.
5. Crucially, when you ask for draft 137, she reconstructs it and hands you a complete report. You never see the slips.

Where this comparison breaks:

6. A clerk works from the previous draft. Git compares each object against a window of candidates and picks whichever gives the smallest difference, which is often not the chronological predecessor.
7. And the clerk's slips would be useless if the full copy were lost. Git guarantees every chain terminates at a complete object inside the same packfile.

38.11.3 PLAIN - a worked example

1. Here is a real repository built for this section: one text file of about 110 KB, edited and committed 200 times, one line changed each time.
2. Before any packing:

```
$ git count-objects -vH
count: 600
size: 6.25 MiB
in-pack: 0
packs: 0
$ du -sh .git/objects
7.2M    .git/objects
```

3. Six hundred loose objects: 200 blobs, 200 trees, 200 commits.
4. The total content, uncompressed, is much larger than what is on disk already:

```
$ git cat-file --batch-all-objects --batch-check='%(objectsize)'
  (summed)
22463780 bytes = 21.42 MiB
```

5. So zlib alone took 21.42 MiB of content down to about 4.6 MiB of file bytes, occupying 7.2 MB of disk blocks.
6. Now run housekeeping:

```
$ git gc
$ git count-objects -vH
count: 0
size: 0 bytes
in-pack: 600
packs: 1
size-pack: 100.87 KiB
$ du -sh .git/objects
140K    .git/objects
```

7. Read those numbers again. 21.42 MiB of content is now in one packfile of 85,423 bytes, with a 17,872-byte index beside it.

Stage	Bytes	Disk
Raw object content	22,463,780	n/a
Loose, zlib only	4,601,531	7.2 MB
Packed with deltas	85,423	140 KB

8. That is a reduction of about 263 to 1 against the raw content, and about 54 to 1 against the already-compressed loose form.
9. Where did it come from? Look inside the pack:

```
5087b1a6... blob 111810 15280 30437
23841aba... blob 78 93 45717 1 5087b1a6...
6578910f... blob 38 52 45810 2 23841aba...
2048ca7c... blob 44 58 45862 3 6578910f...
```

10. The first blob is stored whole: 111,810 bytes, compressed to 15,280. Every following version is stored as a delta of 38 to 78 bytes.
11. Of the 200 blobs, 198 are stored as deltas and only 2 are stored whole.

38.11.4 PLAIN - what is really happening inside

1. A packfile begins with the four ASCII bytes `PACK`, then a 4-byte version number, then a 4-byte object count. Here is the real header of the pack above:

```
00000000  50 41 43 4b 00 00 00 02  00 00 02 58 93 0e 78 9c
```

2. `50 41 43 4b` is `PACK`. `00 00 00 02` is version 2. `00 00 02 58` is 600 in hexadecimal, exactly the object count git reported.
3. The last 20 bytes of the file are a SHA-1 checksum over everything before them, and that checksum is what the file is named after. The pack above is `pack-700da35f...b936f4f9ea0477`, and the file's final bytes are `b9 36 f4 f9 ea 04 77`.
4. Beside it sits a `.idx` file starting with the magic bytes `ff 74 4f 63` and version 2. It is a sorted table letting git find any object's offset in the pack without scanning it.

5. Delta selection works like this. Git sorts objects by type, then by a path hint, then by size descending. It then slides a window over that order and, for each object, tries to express it as a difference against the others in the window.
6. Two limits control the result. The window size, default 10, is how many candidates are considered. The depth, default 50, is how many hops a chain may have.
7. The sandbox pack hit that limit exactly:

```
chain length = 48: 14 objects
chain length = 49: 24 objects
chain length = 50: 27 objects
```

8. Nothing has depth 51, because 50 is the ceiling. Deeper chains would compress better but cost more to read, because reading one object means applying every delta along the chain.
9. Note the direction. Git usually stores newer objects as deltas against older ones or the reverse, depending on size; the ordering heuristic prefers larger objects as bases. There is no rule that the base is the previous version in time.
10. `git gc` runs several jobs: repack loose objects into a pack, pack the refs into `packed-refs`, expire old reflog entries, prune unreachable objects older than the prune window, and update helper indexes such as the commit-graph.

38.11.5 TECHNICAL - the engineer's version

1. Packfile format version 2 has been the on-disk format since 2005 and is stable. It is also the wire format: what travels during clone, fetch and push is a packfile.
2. Two delta encodings exist inside a pack. `OBJ_OFS_DELTA` refers to its base by a relative offset within the same pack, and `OBJ_REF_DELTA` refers to it by full object id. Offsets are smaller, so modern git prefers them locally and uses reference deltas for thin packs on the wire.
3. A **thin pack** omits bases the receiver is known to have. The receiving side completes it with `git index-pack --fix-thin`. This is why a push of one small change is measured in kilobytes even in a large repository.
4. Measured configuration and results from the sandbox:

Item	Value
`pack.window` default	10 candidates
`pack.depth` default	50
Deltified blobs observed	198 of 200
Longest chain observed	50
Pack file / index size	85,423 B / 17,872 B

5. Delta compression is completely orthogonal to git's data model. Every object in a pack still has the same object id it had when loose, still hashes to the same value once reconstructed, and `git cat-file -p` gives identical output. You cannot tell from the model whether an object is deltified.
6. Consequences that do show through, and they matter in real work:
    1. Binary files that do not delta well, such as compressed images, video and compiled artefacts, cost their full size in every version forever.
    2. `core.bigFileThreshold`, default 512 MiB, makes git skip delta attempts and store such files whole.
    3. Repacking a very large repository is memory hungry. `pack.windowMemory` and `pack.threads` exist to control that.
7. `git gc --aggressive` re-computes deltas with a much larger window, by default 250. It can take hours on a large repository and typically yields a modest further reduction. It is rarely the right answer.
8. Modern maintenance uses `git maintenance run` with incremental repacking rather than periodic full `git gc`. That is a change of practice within the last several years, not a change to the format.

9. Historical note: packfiles were not in the first version of git. Linus Torvalds began the project on 3 April 2005 and committed the first version on 7 April 2005, with the message `Initial revision of "git", the information manager from hell`. Loose objects came first, and packing was added later in 2005 when the loose-object-only design proved too slow for the Linux kernel history. Junio Hamano took over as maintainer in July 2005 and released version 1.0 on 21 December 2005.

38.11.6 WORDS - remember these

1. Loose object — one object in one file — zlib-compressed file under `.git/objects/xx/`.
2. Packfile — many objects in one file — the `.pack` format, version 2, with a `.idx` lookup table beside it.
3. Delta — an object stored as a difference from another — `OBJ_OFS_DELTA` or `OBJ_REF_DELTA` inside a pack, purely a storage encoding.
4. Delta chain depth — how many hops to reach a whole object — controlled by `pack.depth`, default 50.
5. Thin pack — a pack that assumes you already have the bases — used on the wire, completed by `git index-pack --fix-thin`.
6. Garbage collection — tidying the object store — `git gc`, which repacks, packs refs, expires reflogs and prunes unreachable objects.

38.12 Reading the reader's own situation

38.12.1 PLAIN - in simple words

1. During the network outage described in this book, the reader's machine reported `origin/main` as `0a95cc8`.
2. Three separate facts are hidden in that one short line, and telling them apart is the whole skill.
3. First, `0a95cc8` is an abbreviated object id. It is the first 7 of the 40 hexadecimal characters of a commit's hash.
4. Second, `origin/main` is a remote-tracking ref. It is a small local file holding that hash.
5. Third, and this is the part people get wrong, that file was written the last time the reader successfully talked to GitHub. It was not read from GitHub at the moment it was displayed.
6. During an outage, nothing on your machine can tell you what the server currently holds. Your machine can only tell you what it remembers.
7. The reader also rebased a local branch onto a moved `main`, and the branch's hash changed afterwards. Given everything in this chapter, that was unavoidable.

38.12.2 PLAIN - a picture in your head

1. Imagine you write a friend's address in your notebook after visiting them.
2. Months later somebody asks where your friend lives. You read out the notebook.
3. You are not reporting where they live. You are reporting where they lived when you last checked.
4. If the phone lines are down, you cannot check. The notebook is all you have, and it is still the same notebook whether they moved yesterday or not.
5. `origin/main` is that notebook page. `git ls-remote` is the phone call.

Where this comparison breaks:

6. A person might tell you they moved. A git server never volunteers anything. It answers only when asked, and only over the network.
7. And your notebook page can be updated by two events, not one: a fetch, and also a successful push, because after a push you know what you just put there.

38.12.3 PLAIN - a worked example

1. Start with the string itself. Seven characters, `0a95cc8`. What can you learn?

```
$ git cat-file -t 0a95cc8
commit
$ git cat-file -s 0a95cc8
$ git cat-file -p 0a95cc8
```

2. Those three commands tell you the type, the size and the full contents: the tree, the parent, the author, the committer and the message.
3. If the object is not present locally, git says so plainly. Run against the sandbox repository, which has never seen that commit:

```
$ git cat-file -t 0a95cc8
fatal: Not a valid object name 0a95cc8
$ git rev-parse 0a95cc8
fatal: ambiguous argument '0a95cc8': unknown revision or path
not in the working tree.
```

4. That message is worth reading carefully. Git cannot tell the difference between "no such commit" and "a commit I have not fetched". Both look identical from here.
5. To expand the abbreviation into the full 40 characters:

```
$ git rev-parse 0a95cc8
$ git rev-parse --short=12 origin/main
```

6. To see what the remote-tracking ref currently says, and where that value physically lives:

```
$ git rev-parse origin/main
b53a9f46d432aa53e687328841c6dd2b89963809
$ cat .git/refs/remotes/origin/main
b53a9f46d432aa53e687328841c6dd2b89963809
```

7. And to ask the server, right now, over the network:

```
$ git ls-remote origin refs/heads/main
b53a9f46d432aa53e687328841c6dd2b89963809    refs/heads/main
```

8. Compare those two values. If they differ, your cached ref is stale. If `git ls-remote` fails, you have learned about the network and nothing at all about the server's contents.

38.12.4 PLAIN - what is really happening inside

1. Why a rebase changes the hash. Recall the rule from section 38.3: a commit's name is the hash of its bytes, and its bytes include the tree, the parent, both people, both timestamps and the message.
2. A rebase does not move a commit. It cannot. It builds a new commit with the same message and the same changes but a different parent.
3. Here is exactly that, done in the sandbox. A branch `fix` was created off `main`, then `main` moved on, then `fix` was rebased.
4. Before the rebase:

```
tree 8376d5522743401a5e02cb99e72435cb6930e5a6
parent 7fc6cf85594a850b727418867d4054af04412e88
author KedByte Reader <reader@example.com> 1767760200 +0530
committer KedByte Reader <reader@example.com> 1767760200 +0530

Fix: add docs/fix.md
```

Its hash was `26c500778e51ea3a240b916e1ca97c33f2c37f55`.

5. After the rebase:

```
tree 353596612537fa8a5e94b31dfa28bd33108ad139
parent 39bf0c8354d383aaf21ff05eaff4304acc24f346
author KedByte Reader <reader@example.com> 1767760200 +0530
committer KedByte Reader <reader@example.com> 1767767400 +0530

Fix: add docs/fix.md
```

Its hash is now `70a5d6220cc7944c8b3824701b4952027bc539af`.

6. Compare the two blocks line by line. The message is identical. The author line is identical, including the original author timestamp.
7. Three things changed: the parent, because it is now on top of `main`; the tree, because the snapshot now also contains `main`'s newer file; and the committer timestamp, because the commit was recorded again just now.
8. Any one of those three would have been enough to change the hash. All three changed.
9. The old commit still exists. Nothing was destroyed:

```
$ git reflog fix
70a5d62 fix@{0}: rebase (finish): refs/heads/fix onto 39bf0c8
26c5007 fix@{1}: commit: Fix: add docs/fix.md
7fc6cf8 fix@{2}: branch: Created from main
$ git cat-file -t 26c5007
commit
```

10. So "the SHA changed after a rebase" is not really true. A different commit now exists, and the branch file was rewritten to point at it. Both commits are on disk.

38.12.5 TECHNICAL – the engineer's version

1. Abbreviation length. Git's default is `core.abbrev=auto`, which picks a length long enough that a collision among the objects present is unlikely, with a floor of 7 characters. Small repositories therefore show 7.
2. A 7-character prefix is 28 bits, about 268 million values. By the birthday bound, ambiguity becomes likely once a repository holds a few tens of thousands of objects, which is why large repositories abbreviate to 10, 12 or more.
3. An abbreviation is only unique within one repository at one moment. Adding objects can make a previously unique prefix ambiguous. Never store an abbreviated hash as a durable identifier; store all 40 characters.
4. What actually updates `refs/remotes/origin/*`:

Event	Updates origin/main?
`git fetch`	yes
`git pull`	yes, it fetches first
Successful `git push`	yes
`git ls-remote`	no, reads only
Someone else pushing	no
Time passing	no

5. The session evidence, applied. During the outage the reader had cached `origin/main` as `0a95cc8`. That was a true statement about the past and carried no information about the present.
6. Two pushes in that session failed with `send-pack: unexpected disconnect while reading sideband packet`. The write may or may not have landed; a disconnect while reading the response tells you

nothing about the request. The correct next step is not to guess but to re-query with `git ls-remote origin refs/heads/<branch>` and compare hashes.

7. This is why verifying by hash beats verifying by name. A branch name is a pointer that anyone can move. A commit hash is the content itself. In the same session, continuous integration was checked by run identifier against the exact head SHA rather than by trusting a coloured tick next to a branch name, and that is the stronger check for exactly this reason.
8. Practical checklist for "is my local view of the remote correct":

```
git rev-parse origin/main              # what I remember
git ls-remote origin refs/heads/main   # what is true now
git fetch origin && git rev-parse origin/main
git log --oneline origin/main..main    # mine, not theirs
git log --oneline main..origin/main    # theirs, not mine
```

9. And after a rebase, the correct push is `git push --force-with-lease`, not `--force`. The lease form refuses if the remote moved since your last fetch, which converts a silent overwrite of someone else's work into an error you can read.

38.12.6 WORDS - remember these

1. Abbreviated object id — the first few characters of a hash — a unique prefix within one repository, length chosen by `core.abbrev`.
2. Remote-tracking ref — your cached copy of a remote branch pointer — a ref under `refs/remotes/`, updated only by fetch and push.
3. `ls-remote` — ask the server what it holds right now — a network query that lists the remote's refs without changing anything locally.
4. Rebase — replay commits onto a new base — creates new commit objects with new parents, therefore new object ids.
5. `--force-with-lease` — force push, but only if nothing moved — a compare-and- swap on the remote ref against your last known value.

38.13 Why this design is good, and what it costs

38.13.1 PLAIN - in simple words

1. The benefits all fall out of one decision: name things by their content.
2. **Deduplication across all of history.** The same file content, in any commit, on any branch, from any author, is stored once.
3. **Integrity for free.** Every read can be checked against the name. Damage and tampering are both detectable without any extra machinery.
4. **Cheap branching.** A branch is 41 bytes, so having twenty of them costs less than one photograph.
5. **Distributed operation with no coordination.** Two people can create commits at the same time, offline, with no server involved, and the results will never clash, because names are computed from content and not assigned by an authority.
6. That last point is why the reader could keep committing all through a network outage. Nothing in `git commit` needs a server.
7. The costs are real too.
8. Large binary files are handled badly. Rewriting history is confusing. And the whole model takes genuine effort to learn, which is why this chapter exists.

38.13.2 PLAIN - a picture in your head

1. Think of two libraries in two cities that never phone each other.
2. Each catalogues books by a code derived from the book's exact text.

3. When they finally exchange catalogues, matching codes mean identical books, with no argument and no committee.
4. Different codes mean different books, even if the titles match.
5. Neither library needed permission from the other to catalogue anything.

Where this comparison breaks:

6. Real libraries have to decide which copy is the official one. Git also has to decide, and it does not decide by content. A human decides, by choosing what `main` points at on the server.
7. Content addressing removes the need to coordinate naming. It does not remove the need to agree on what is authoritative.

38.13.3 PLAIN - a worked example

1. Deduplication, measured. In the sandbox, four files with three identical contents produced two blobs, not four.
2. Integrity, measured. A single bit was flipped in one loose object file:

```
$ git fsck
error: inflate: data stream error (incorrect data check)
error: unable to unpack header of .git/objects/6f/e3e873...
error: 6fe3e873...: object corrupt or missing
missing blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3
```

3. That one was caught by zlib's own checksum. So a more careful attack was tried: a perfectly well-formed object, holding different content, written under the old name.

```
$ git cat-file -p 6fe3e87
The EVIL three words.

$ git fsck
error: 8d71e45d309cb1328988a64591163f5c4f8f43f1: hash-path
mismatch, found at: .git/objects/6f/e3e873...
missing blob 6fe3e87371433ed417646e3cb2354cfb9cff08d3
```

4. Read that carefully, because it contains both the strength and an important limitation.
5. The strength: `git fsck` computed the content's true name, `8d71e45...`, found it stored under `6fe3e87...`, and reported the mismatch. The tamper is caught with certainty.
6. The limitation, and it is honest: `git cat-file -p` handed back the evil content without complaint. Git does not re-verify every loose object on every ordinary read. Verification happens on transfer, when configured, and when you ask with `fsck`.
7. Cheap branching, measured: 1,000 branches created in 2.28 seconds, using 41,000 bytes of content.
8. Distributed operation: two commits created independently, offline, will have different hashes unless they are byte-for-byte identical, in which case they are the same commit and it does not matter.

38.13.4 PLAIN - what is really happening inside

1. Now the costs, honestly.
2. **Large binary files.** Git stores whole content per version. A 50 MB video edited ten times is ten blobs. Compressed formats do not delta, so the packfile saves almost nothing.
3. Worse, a clone fetches all of history by default. The 500 MB stays in every clone forever, even after the file is deleted, because the old commits still reference it.
4. The workarounds are all partial: `git lfs`, which stores a pointer file in git and the bytes elsewhere; shallow clones with `--depth`; partial clones with `--filter=blob:none`; and rewriting history with `git filter-repo`, which changes every hash from the rewrite point onwards.

5. **History rewriting is confusing** precisely because it is not rewriting. It is creating parallel objects and moving pointers. The old objects linger, other people still have the old hashes, and any branch built on the old ones must itself be rebuilt.
6. **The model takes effort.** Most tools let you learn the commands without the data model. Git punishes that. Almost every confusing git error makes immediate sense once you know that commits are immutable snapshots and branches are movable pointers, and almost none of them make sense otherwise.
7. There are smaller costs too. Git tracks content, not files, so it has no record of a rename; it infers renames when displaying a diff, and the inference can be wrong. It records only the executable bit, so full permissions and file ownership need a separate tool. And empty directories cannot be represented at all, because a tree with no entries in it is simply not referenced.

38.13.5 TECHNICAL – the engineer's version

1. Benefits and their mechanisms, stated precisely:

Benefit	Mechanism
Global deduplication	key = hash of content
Integrity verification	re-hash and compare
O(1) branch creation	write one 41-byte ref
Offline commit	ids need no authority

2. The distributed property is worth stating exactly. Git needs no central allocator of identifiers because the identifier space is a hash of content, and independent parties producing identical content produce identical ids by construction. This is the same property that lets content-addressed systems such as IPFS and container image registries work.
3. Costs, with the mitigations and their prices:

Cost	Mitigation	Price of the mitigation
Big binaries	Git LFS	extra server, extra tooling
Deep history size	`--depth` clone	limited history locally
Blob bloat	`--filter=blob:none`	fetches on demand
Bad past commits	`filter-repo`	every later hash changes

4. `core.bigFileThreshold`, default 512 MiB, controls when git stops attempting delta compression. Below it, git still tries, and for compressed media it still fails to save anything.
5. Rename detection is a display-time heuristic controlled by `diff.renameLimit` and a similarity threshold, by default 50 percent. It is not stored, it is recomputed, and on very large changesets git may give up entirely and report the rename as a delete plus an add.
6. Where experts disagree: whether git's monorepo behaviour at very large scale is a flaw in the model or an implementation gap. Facebook chose Mercurial in 2013 partly over this, and Microsoft built a virtual filesystem layer for the Windows repository, later replaced by scalar and partial clone in git itself. Both camps agree the model is sound; they disagree about how far it scales without special tooling.
7. Established fact: the object model has not changed since 2005, other than the addition of the optional SHA-256 format. Active work: the SHA-256 transition and the reftable ref back-end. Marketing claim territory: any statement that one hosting provider makes git itself faster. Hosting affects the network path, not the object model.

38.13.6 WORDS - remember these

1. Monorepo — one repository holding many projects — a scaling pattern that stresses git's whole-history clone model.
2. Git LFS — a way to keep big files out of git — Large File Storage, which commits a small pointer file and stores the bytes on a separate server.
3. Partial clone — a clone that skips file contents until needed — enabled with `--filter=blob:none`, fetching blobs on demand.
4. Shallow clone — a clone with truncated history — `--depth <n>`, which omits older commits and their objects.
5. Rename detection — noticing a file moved — a similarity heuristic computed at diff time, never stored in any object.

38.98 Common wrong ideas

1. Wrong: a commit stores a diff, the changes you made. Right: a commit names one tree, which is a complete snapshot of every file. The second version of `docs/intro.md` in this chapter is stored as all 38 bytes of the new file, not as "add one line".
2. Wrong: a branch contains commits. Right: a branch is a 41-byte file holding one commit's hash. The commits you see are the ones reachable by walking parent links backwards from there, and they may be reachable from twenty other branches too.
3. Wrong: git compresses by storing diffs between versions of a file. Right: git compresses in two independent ways. zlib compresses each object separately, and packfiles store some objects as deltas against any similar object, chosen by a size and window heuristic, not by version order. Neither is part of the history model.
4. Wrong: deleting a branch deletes its commits. Right: it deletes one small file. The commits stay until nothing references them, their reflog entries expire, and garbage collection runs, which by default is at least 30 days.
5. Wrong: `git commit --amend` edits the last commit. Right: it builds a new commit object with a new hash and moves the branch to it. The original is still on disk and is listed in the reflog.
6. Wrong: detached HEAD is an error state. Right: it is a normal state meaning `HEAD` names a commit instead of a branch. Every CI checkout runs this way. The only risk is losing sight of commits you make there.
7. Wrong: `origin/main` shows what is on the server. Right: it shows what was on the server the last time you fetched or pushed. Only `git ls-remote` or a fresh `git fetch` asks the server.
8. Wrong: rebasing moves commits onto a new base. Right: it creates new commit objects with new parents, new trees and new committer timestamps, therefore new hashes. The originals remain until they are collected.
9. Wrong: because git uses SHA-1, and SHA-1 is broken, git repositories can be forged. Right: SHA-1 collision resistance is broken in general, but since Git 2.13 in May 2017 git uses a collision-detecting SHA-1 that refuses known attack patterns, and a useful attack must also survive review and transport. The risk is real but it is not "anyone can forge a commit".
10. Wrong: a tag is just a branch that does not move. Right: a lightweight tag is close to that, but an annotated tag is a separate stored object with its own hash, tagger, date, message and optional signature, and `git describe` treats the two differently.

38.99 Chapter summary in 20 lines

1. Git names every stored thing by hashing the thing itself. That single decision explains almost everything else.
2. The name is SHA-1 of `<type> <size>\0<content>`, printed as 40 hexadecimal characters, and you can reproduce it with `printf` and `sha1sum`.

3. SHA-1's collision resistance was publicly broken by CWI Amsterdam and Google on 23 February 2017, at about 2^63.1 evaluations.
4. Git responded in version 2.13, May 2017, by making the collision-detecting SHA-1 of Marc Stevens and Dan Shumow the default; a SHA-256 object format exists but is still not the default in 2026.
5. There are four object types. A blob is file content with no name. A tree is one directory listing with modes. A commit names one tree and zero or more parents. A tag object is an annotated pointer.
6. Tree entries store the mode in ASCII, the name, a zero byte, and 20 raw bytes of hash. Only five modes are legal, including 120000 for symlinks and 160000 for submodules.
7. Changing one file creates one new blob, one new tree per directory level up to the root, and one new commit. Everything else is pointed at again.
8. In the worked repository, editing `docs/intro.md` created 4 new objects and reused 4 existing ones.
9. A commit's hash covers its parent's hash, so one tip hash seals the entire history behind it. History is immutable because names are derived from content.
10. History is a directed acyclic graph, not a line. Merge commits have two or more parents, and `^1` and `^2` select between them.
11. A branch is a file of 40 hexadecimal characters and a newline. Creating one writes 41 bytes; 1,000 branches took 2.28 seconds and 41,000 bytes.
12. Refs may be packed into `.git/packed-refs`, in which case the individual files vanish. A loose file always wins over the packed entry.
13. `HEAD` normally holds `ref: refs/heads/main`. When it holds a raw hash you are detached, which is normal, not an error, and recoverable.
14. Other refs live under `refs/tags`, `refs/remotes` and `refs/notes`, plus pseudo-refs such as `ORIG_HEAD` and `FETCH_HEAD`.
15. The reflog records every ref movement locally, keeps reachable entries 90 days and unreachable ones 30, and turns a catastrophic `git reset --hard` into a one-line recovery with `git reset --hard main@{1}`.
16. Plumbing commands show the machinery: `cat-file`, `rev-parse`, `hash-object`, `ls-tree`, `rev-list`, `count-objects`, `verify-pack`. Porcelain commands are for people and their output may change.
17. Storage is loose objects first, then packfiles. In the worked example 21.42 MiB of object content became one packfile of 85,423 bytes, with 198 of 200 blobs stored as deltas up to depth 50.
18. Deltas are storage only. Git's model is still whole snapshots, and the object ids are identical either way.
19. The reader's `origin/main` of `0a95cc8` was a cached local memory, not a live fact; only `git ls-remote` or a fresh fetch asks the server.
20. The rebased branch had to get a new hash, because its parent, its tree and its committer timestamp all changed, and all three go into the hash.

39

CHAPTER

KB

The Index, Push, Fetch, Merge and Rebase

39.0 What this chapter gives you

1. You will be able to name the three places a file can live in git, working directory, index and object database, and say which command moves content between which pair.
2. You will be able to open `.git/index`, read its header by hand, and explain why it is a full snapshot with stat data and not a list of differences.
3. You will be able to prove that `git add` writes a real object to disk immediately, before any commit exists, and find that object yourself.
4. You will be able to describe the three steps `git commit` performs, write a tree, make a commit, move the branch ref, and do all three by hand with plumbing commands.
5. You will be able to choose correctly between `git restore`, `git reset` in its three modes, `git revert`, `git clean` and `git stash`, and say for each one whether it can lose work.
6. You will be able to describe exactly what crosses the network on a push: capability advertisement, `have` and `want` negotiation, one packfile, pkt-line framing, and three sideband channels.
7. You will be able to explain the reader's own failure, `send-pack: unexpected disconnect while reading sideband packet`, say which stage it happened at, and say why it tells you nothing about whether the write landed.
8. You will be able to state the rule for non-idempotent operations over unreliable transports, and apply it beyond git.
9. You will be able to explain merge base, three-way merge, fast-forward, conflict markers, and why a clean text merge can still produce broken code.
10. You will be able to explain why rebase changes a branch's SHA even though the changes are identical, and decide when rebasing is safe.

39.1 The three states and three places

39.1.1 PLAIN – in simple words

1. A file that git is tracking exists in three different places at once.
2. The first place is the **working directory**. That is the ordinary folder on your disk, holding the files you open and edit.
3. The second place is the **index**, also called the staging area. It holds the exact version of every file that your next commit will contain.
4. The third place is the **object database**, inside `.git/objects`. That is the permanent store of every version git has ever been told to keep.
5. A fourth thing sits alongside these three, called **HEAD**. It is a pointer saying which commit you currently have checked out.
6. The three copies of a file are often identical. When they are, git says the working tree is clean and prints nothing interesting.

7. When they differ, git tells you which pair differs. That is the whole of `git status`.
8. "Changes to be committed" means index differs from HEAD.
9. "Changes not staged for commit" means working directory differs from index.
10. "Untracked files" means the file is in the working directory and in neither of the other two places.

39.1.2 PLAIN - a picture in your head

1. Think of posting a parcel at a counter.
2. Your desk at home is the working directory. Things lie around, half sorted, in whatever state you left them.
3. The counter at the post office is the index. You put on the counter only the items you actually want to send.
4. You can put one item on the counter, go home, change your mind about a second item, and come back. The counter keeps what you already placed.
5. The sealed and stamped parcel is the commit. Once sealed, its contents are fixed forever.
6. The archive room behind the counter is the object database. Every parcel ever sealed is stored there.
7. The clerk's note saying which parcel you are currently working from is HEAD.

Where this comparison breaks:

8. At a real post office the item leaves your desk when you put it on the counter. In git, staging a file copies it. Your working file stays put.
9. A real counter holds items. The git index holds a complete list of every tracked file, not only the changed ones.
10. A real parcel is opened by the receiver. A git commit is never opened and changed. It is replaced by a new one.

39.1.3 PLAIN - a worked example

1. Here is a real repository with one change of every kind in it, and its real `git status` output.

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   TODO.md
    modified:   calc.py
    deleted:    notes.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes)
    modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    out.log
```

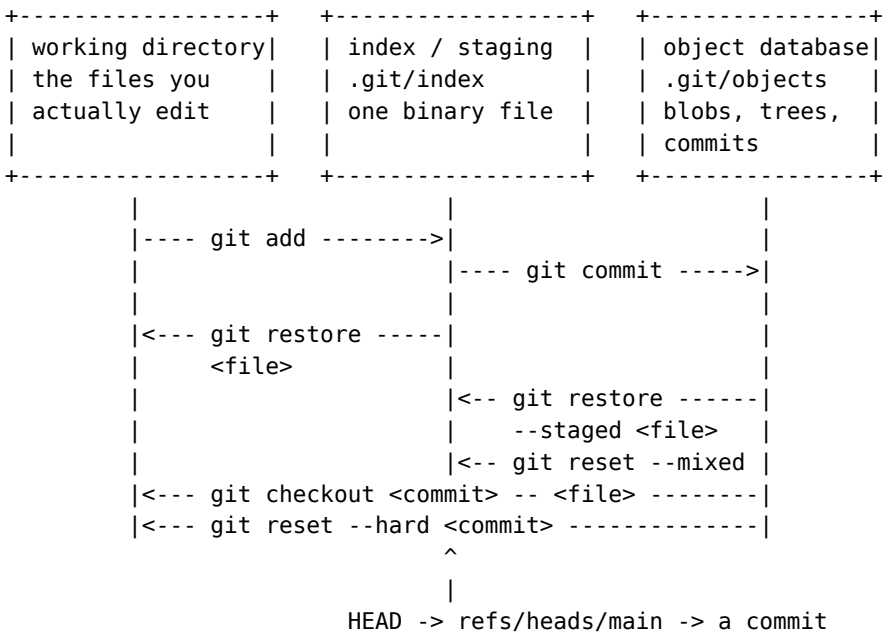
2. Map every line onto the three places.
3. `new file: TODO.md`. Not in HEAD. Present in the index. So index differs from HEAD, and the difference is an addition.
4. `modified: calc.py`. In HEAD with one content, in the index with another.
5. `deleted: notes.txt`. Present in HEAD, absent from the index.
6. `modified: README.md` under "not staged". Index and HEAD agree. The working file differs from both.
7. `out.log` under "Untracked". Not in HEAD, not in the index, only on disk.
8. The same state in short form, which is easier to read once you know it.

```
$ git status --short --branch
## main
 M README.md
A  TODO.md
M  calc.py
D  notes.txt
?? out.log
```

9. The two columns are index-versus-HEAD first, then working-versus-index.
10. So `M` means the index is clean and the working file changed. `M` means the index changed and the working file matches the index.

39.1.4 PLAIN - what is really happening inside

1. Here is the picture. The arrows are commands, and each one moves content in one direction between two of the three places.



2. `git add` reads a working file, stores its content in the object database, and records the resulting name in the index.
3. That is worth reading twice. `git add` already writes to the object database. The index only records the name of what was written.
4. `git commit` reads the index, builds tree objects from it, builds one commit object, and moves the branch that HEAD points at.
5. `git restore <file>` copies from the index back over the working file.
6. `git restore --staged <file>` copies from HEAD back into the index.
7. `git reset --hard <commit>` moves HEAD and the branch, then overwrites both the index and the working directory from that commit.
8. Nothing in this picture ever edits a stored object. Objects are written once and never modified.

39.1.5 TECHNICAL - the engineer's version

1. The three states are formally: modified, staged, committed. The three storage locations are the working tree, the index, and the object store.
2. The index is a single binary file at `$GIT_DIR/index`, by default `.git/index`. Its location can be overridden by `GIT_INDEX_FILE`.

3. `git status` is implemented as two diffs: `diff-index --cached HEAD` for the staged column and `diff-files` for the unstaged column, plus a directory walk for untracked paths.
4. The machine-readable form is `git status --porcelain=v2`, stable across versions and intended for scripts. Here is the same repository state.

```
$ git status --porcelain=v2 --branch
# branch.oid 66b84b4f79e5cce670a6a2c0f739d5433403423e
# branch.head main
1 .M N... 100644 100644 100644 1af06666... 1af06666... README.md
1 A. N... 000000 100644 100644 00000000... 1333ed77... TODO.md
1 M. N... 100644 100644 100644 4693ad3c... 6cddd28b... calc.py
1 D. N... 100644 000000 000000 fb188b9e... 00000000... notes.txt
? out.log
```

5. Columns after the XY code are: submodule state, mode in HEAD, mode in index, mode in worktree, object name in HEAD, object name in index, path. Hashes are abbreviated here to fit the page; the real output prints all 40 digits.
6. Note `TODO.md` has HEAD mode `000000` and HEAD hash all zeros. That is how git encodes "not present in HEAD".

Status word	Which two places differ	Direction
to be committed	index vs HEAD	staged
not staged	worktree vs index	unstaged
untracked	in worktree only	unknown to git
ignored	worktree, matched gitignore	suppressed

7. Commands that observe this: `git status`, `git diff` (worktree vs index), `git diff --cached` (index vs HEAD), `git diff HEAD` (worktree vs HEAD), `git ls-files -s`, `git diff-index`, `git diff-files`.

39.1.6 WORDS - remember these

1. Working directory — the files you edit — the checked-out worktree on the filesystem, outside `.git`.
2. Index — the counter where you put things to send — the binary staging file `.git/index` holding the next commit's full snapshot.
3. Object database — the archive of everything kept — `.git/objects`, holding loose and packed blob, tree, commit and tag objects.
4. HEAD — the note saying where you are — a symbolic ref, usually pointing at a branch ref under `refs/heads/`.
5. Staged — placed on the counter — present in the index with content differing from HEAD.
6. Untracked — git has never been told about it — a path in the worktree with no index entry and no HEAD entry.

39.2 What the index actually is

39.2.1 PLAIN - in simple words

1. The index is not an idea. It is a real file on your disk, and you can look at it.
2. It is at `.git/index`. It is binary, so opening it in a text editor shows rubbish, but its structure is documented and simple.
3. It holds one entry per tracked file. Not one entry per changed file. Every single tracked file, always.
4. Each entry stores the file's path, the name of its stored content, its permission bits, and a copy of what the operating system said about the file the last time git looked.
5. That last part is the clever bit. Git remembers the file's size, its last modified time, and its position on disk.

6. Next time you run `git status`, git asks the operating system for those same facts again. If they match, git assumes the file did not change and does not read it at all.
7. That is why `git status` is fast even in a project with tens of thousands of files. Most files are never opened.
8. The index stores whole snapshots, not differences. There is no "diff" anywhere in it.

39.2.2 PLAIN - a picture in your head

1. Think of a hotel cloakroom with a big register book.
2. Every guest's coat has a ticket. The register lists, for each coat, the ticket number, the peg it hangs on, its colour, and its weight.
3. When you come back, the clerk does not take every coat down and inspect it.
4. The clerk glances at the register and at the pegs. If the weights and colours still match, nothing has changed and the clerk says so instantly.
5. Only if something looks different does the clerk take a coat down and check it properly.
6. The register is the index. The weights and colours are the stat data. The coats themselves are the stored objects.

Where this comparison breaks:

7. A clerk could be fooled by two coats of identical weight and colour. Git can be fooled too, and this is real: if a file changes within the same second and keeps the same size, git can miss it. Git has a "racy timestamp" rule to handle that case, described later in this section.
8. A cloakroom register lists only coats currently held. The git index lists every tracked file, even ones you have not touched in years.

39.2.3 PLAIN - a worked example

1. Here is a small repository with five tracked files. First its size, then its contents.

```
$ ls -l .git/index
-rw-r--r-- 1 root root 479 .git/index

$ git ls-files -s
100644 1af06666...9965 0    README.md
100644 1333ed77...ee69 0    TODO.md
100644 6cddd28b...7159 0    calc.py
100644 b04e7afa...7b19 0    docs/guide.md
100644 11b15b1a...a0d5 0    src/main.py
```

2. The columns are: file mode, object name of the content, stage number, path. Stage 0 means "not conflicted". Hashes shortened here to fit the page.
3. Note `docs/guide.md` and `src/main.py`. The index has no directories. It has flat paths with slashes in them.
4. Now the stat cache. The `--debug` flag prints it.

```
$ git ls-files --debug | head -7
README.md
  ctime: 1786606646:156685960
  mtime: 1786606646:156685960
  dev: 65024    ino: 1049252
  uid: 0    gid: 0
  size: 13  flags: 0
TODO.md
```

5. `mtime` is when the content last changed. `ctime` is when the file's metadata last changed. `ino` is the inode number, the file's identity on the filesystem. `size` is 13 bytes.

6. Every one of those numbers came from a single `stat` call to the operating system, which is far cheaper than reading and hashing the file.
7. Here is the header of that binary file, byte by byte.

```
$ head -c 16 .git/index | od -A d -t x1z
0000000 44 49 52 43 00 00 00 02 00 00 00 05  >DIRC........<
```

8. `44 49 52 43` is the ASCII letters `DIRC`. Then `00 00 00 02` is version 2. Then `00 00 00 05` is five entries.

39.2.4 PLAIN - what is really happening inside

1. The file has three parts: a 12-byte header, then the entries, then optional extra sections, then a checksum at the end.
2. Each entry begins with 40 bytes of numbers copied from the operating system's `stat` call: two timestamps with nanoseconds, device, inode, mode, user, group and size.
3. Then 20 bytes holding the content's SHA-1 name. Then 2 bytes of flags, including the path length and the merge stage.
4. Then the path itself, followed by at least one zero byte, padded so the whole entry length is a multiple of eight.
5. The entries are sorted by path, as raw bytes. That sorting is what lets git find an entry by binary search instead of scanning.
6. Let us add it up for the five-file repository above. Fixed part is 62 bytes. `README.md` is 9 characters, so 62 plus 9 is 71, padded up to 72.
7. Doing that for all five paths gives 72, 72, 72, 80 and 80, which is 376.
8. Add the 12-byte header and the 20-byte trailing checksum and we have 408. The file is 479. The missing 71 bytes are one extension section.
9. That section is the cached tree, signature `TREE`, 63 bytes of data plus its own 8-byte header. It remembers tree objects already computed, so a commit does not have to rebuild trees for unchanged directories.
10. When `git status` runs, for each entry git calls `lstat` on the path. If size, mtime, ctime and inode all match, git marks the file unchanged without reading a single byte of it.

39.2.5 TECHNICAL - the engineer's version

1. The format is specified in the git documentation page `gitformat-index`. The signature is the four bytes `DIRC`, standing for "directory cache".
2. Supported versions are 2, 3 and 4. Version 3 adds a second flags word for skip-worktree and intent-to-add. Version 4 prefix-compresses path names.
3. All multi-byte numbers are big-endian ("network byte order"). The trailing checksum is a hash over everything before it.

Field group	Bytes	Contents
header	12	DIRC, version, entry count
entry, fixed part	62	stat data, OID, flags
entry, variable	var	path, NUL, pad to 8
trailer	20	SHA-1 of the whole file

4. Here is that exact five-entry index parsed by a short script, showing the real numbers.

```
total size: 479
signature: DIRC  version: 2  entries: 5
  README.md      mode=100644 size= 13 stage=0 bytes=72
  TODO.md        mode=100644 size=  5 stage=0 bytes=72
  calc.py        mode=100644 size= 65 stage=0 bytes=72
  docs/guide.md  mode=100644 size= 12 stage=0 bytes=80
```

```
  src/main.py    mode=100644 size= 15 stage=0 bytes=80
offset after entries: 388
extension: TREE size: 63
```

5. Only two file modes are legal for regular files, `100644` and `100755`. `120000` is a symbolic link and `160000` is a gitlink (a submodule).
6. The stat cache is a cache, so it can be wrong. The known hazard is the "racy index" problem: a file written in the same second as the index itself may have a matching mtime yet different content. Git handles this by treating any entry whose mtime is not strictly older than the index's own mtime as suspect, and re-reading it.
7. Real measurements, taken on a Linux sandbox with git 2.43.0, on a repository with 10,000 tracked files totalling 2,850,000 bytes:

Situation	git status time
stat cache valid	20 to 25 ms
every mtime invalidated	89 to 103 ms
hashing all 10,000 files	77 ms

8. The index for those 10,000 entries was 801,570 bytes, about 80 bytes per entry. The honest version: this repository is tiny, so the gap is only about four times. On a repository with large files the gap is far larger, because the cost of re-reading grows with total bytes while the stat calls do not.
9. Since git 2.36, released 18 April 2022, git ships a built-in filesystem monitor daemon, enabled with `git config core.fsmonitor true`. It replaces the per-file `lstat` scan with change notifications from the operating system, and matters most on macOS and Windows.
10. Sparse-checkout with a sparse index, `extensions.sparseIndex`, allows directory entries in the index with mode `040000`, so that a huge repository does not need an index entry per file.

39.2.6 WORDS - remember these

1. Index entry — one line in the register — a 62-byte fixed record plus a padded path, holding stat data, OID, mode and flags.
2. Stat data — what the filesystem says about a file — `st_mtime`, `st_ctime`, `st_dev`, `st_ino`, `st_size` and mode from `lstat(2)`.
3. Stat cache — remembering that so you need not re-read — the mechanism that makes `git status` skip unchanged files.
4. DIRC — the four letters at the start — the index file signature, short for "directory cache", present since git's first commit in April 2005.
5. Stage number — which side of a conflict — 0 for normal, 1 for merge base, 2 for ours, 3 for theirs.
6. Cached tree — the remembered directory hashes — the `TREE` index extension that lets `git write-tree` skip unchanged directories.

39.3 What `git add` really does

39.3.1 PLAIN - in simple words

1. Most people believe `git add` marks a file for later. It does much more than that, and it does it at once.
2. When you run `git add`, git reads the file, compresses its content, works out its name from the content, and writes that content into `.git/objects` straight away.
3. Then it writes an entry in the index recording that name against the path.
4. So the content is already safely stored before you type `git commit`. The commit does not copy your files; the copy has already happened.

5. This is why staging a file and then editing it again gives you three different versions at once: the committed one, the staged one, and the one on disk.
6. It also means you can stage part of a file. Git can build a version that exists nowhere on disk, store it, and put that in the index.
7. That partial staging is done with `git add -p`, which walks you through each chunk of change and asks yes or no.

39.3.2 PLAIN – a picture in your head

1. Imagine a photocopier next to your desk that also files things.
2. When you say "add this page", the machine does not put a sticky note on your page. It photocopies it immediately and files the copy in a cabinet.
3. It then writes in a ledger: "for the page called report.txt, use copy number eb2fc3ca".
4. You can now scribble all over your original. The copy in the cabinet is untouched.
5. If you later say "add this page" again, a new copy is made and the ledger line is updated to point at the new copy. The old copy stays in the cabinet, unreferenced.

Where this comparison breaks:

6. A photocopier makes a new copy every time. Git checks first: if a copy with that exact content already exists, it stores nothing and reuses it.
7. The ledger in this picture holds one line per page you added. The real index holds one line per tracked file, added or not.

39.3.3 PLAIN – a worked example

1. Start a completely empty repository and prove that `git add` writes to `.git/objects` before any commit exists.

```
$ git init -q .
$ find .git/objects -type f
(nothing)

$ printf 'hello index\n' > greeting.txt
$ git hash-object greeting.txt
eb2fc3ca2f129a710df1a6c0fd5ebfd088a10bfd

$ git add greeting.txt

$ find .git/objects -type f
.git/objects/eb/2fc3ca2f129a710df1a6c0fd5ebfd088a10bfd

$ git log --oneline
fatal: your current branch 'main' does not have any commits yet

$ git ls-files -s
100644 eb2fc3ca2f...bfd 0   greeting.txt
```

2. Read that carefully. There is no commit. There is no branch tip. Yet the content is already a permanent object in the database.
3. `git hash-object` printed the name before the add, without writing anything. It is the same name, because the name is computed from the content.
4. Now the three-version demonstration. Take a file with two separate changes and stage only the first.

```
$ git add -p
@@ -1,6 +1,6 @@
 import os
```

```
-PORT = 8080
+PORT = 9090
 DEBUG = False

(1/2) Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]? y
@@ -8,3 +8,4 @@ def start():
 def stop():
     print("stopping")
+    print("goodbye")
(2/2) Stage this hunk [y,n,q,a,d,K,g,/,e,?]? n
```

5. Now ask git for the name of `server.py` in all three places.

```
committed  : 6802ebd3762732a47322b13ed2d9510a9145a23f
index      : 345790221885e55f73d7e6c21d54fffc51998330
working dir: 154a7a3ee775a1746b842017a179c42cee649a30
```

6. Three different names, so three different contents. The middle one exists nowhere on your disk as a file. It is a real object in `.git/objects`.

39.3.4 PLAIN - what is really happening inside

1. `git add path` performs these steps, in order, for each path.
2. Call `lstat` on the path to get size, mode and timestamps.
3. Read the file's bytes. Apply any content filters that are configured, for example line-ending conversion or a clean filter.
4. Build the object header: the word `blob`, a space, the byte count in decimal, and one zero byte. Put the content after it.
5. Compute SHA-1 over header plus content. That 40-character result is the object name.
6. Check whether an object with that name already exists. If it does, do nothing more; the content is already stored.
7. If it does not, compress the header-plus-content with zlib and write it to `.git/objects/ab/cdef...`, where `ab` is the first two characters.
8. Write or update the index entry: path, the new object name, the mode, and the freshly captured stat data.
9. Write the whole index back out, atomically, by writing `.git/index.lock` and renaming it over `.git/index`.
10. For `git add -p`, git first computes the diff, splits it into hunks, and for each accepted hunk applies it to a copy of the staged version. The result of applying the accepted subset is what gets hashed and stored.

39.3.5 TECHNICAL - the engineer's version

1. The plumbing equivalents are `git update-index --add --cacheinfo` and `git hash-object -w`. The porcelain `git add` is a wrapper over both.
2. `git hash-object` without `-w` computes and prints the OID without writing. With `-w` it writes the loose object.
3. Loose objects are zlib-compressed with the default compression level set by `core.compression`, default -1 meaning zlib's own default of 6.
4. Useful `git add` variants:

Command	Effect
`git add -p`	interactive per-hunk staging
`git add -u`	stage changes to tracked files only
`git add -A`	stage everything, including deletions

Command	Effect
`git add -N`	record intent to add, no content
`git add -e`	edit the diff by hand before staging

5. `git add -N` sets the intent-to-add flag, which requires index version 3. It makes an untracked file appear in `git diff` output without staging content.
6. In `git add -p`, the keys are: `y` stage, `n` skip, `q` quit, `a` stage this and all remaining in the file, `d` skip this and all remaining in the file, `s` split the hunk, `e` edit the hunk, `j/k` move without deciding.
7. `git add --interactive` was written by Junio C Hamano and merged on 18 December 2006; it shipped in git 1.5.0 on 14 February 2007.
8. Filters that change content on the way in are set by gitattributes: `text` and `eol` for line endings, and `filter.<name>.clean` for custom transforms such as Git LFS. The honest version: with a clean filter configured, the blob stored is not the bytes on your disk.
9. To see what would be staged without staging it, use `git diff` for unstaged changes and `git add --dry-run -A`.

39.3.6 WORDS - remember these

1. Blob — the stored content of one file — a git object of type blob, containing raw bytes with no filename.
2. Hunk — one chunk of change in a diff — a `@@` section with its context lines, the unit `git add -p` operates on.
3. Loose object — one file per object — an object stored individually under `.git/objects/xx/`, as opposed to inside a packfile.
4. Intent to add — "this file is coming" — the index flag set by `git add -N`, registering a path with no content.
5. Clean filter — a transform applied on the way in — `filter.<name>.clean` from gitattributes, run before hashing.
6. Atomic index write — no half-written index — the lock-file-and-rename sequence git uses for `.git/index`.

39.4 What `git commit` really does

39.4.1 PLAIN - in simple words

1. `git commit` does exactly three things, in this order.
2. First, it turns the index into tree objects. One tree for the top folder, and one for each subfolder, each listing names, modes and object names.
3. Second, it creates one commit object. That object holds the name of the top tree, the name of the commit you were on, who you are, when it is, and your message.
4. Third, it changes one small text file, the branch file, so it holds the new commit's name.
5. That is all. No files are copied. The blobs were already written when you ran `git add`.
6. Because the branch file is the last thing to change, the commit is either fully there or not there at all.
7. If a step fails halfway, the objects written so far are simply unreferenced and will be cleaned up later. Nothing is corrupted.

39.4.2 PLAIN - a picture in your head

1. Think of publishing an edition of a newspaper.
2. All the articles are already typeset and stored, one file each. That happened when you staged them.
3. Making the edition means writing a contents page listing which article file goes on which page. That is the tree.
4. Then you write a cover sheet: edition number, date, editor's name, a sentence about what is in it, and the number of the previous edition. That is the commit.

5. Then you update the single sign on the wall that says “current edition”. That is the branch file.

Where this comparison breaks:

6. A newspaper’s contents page is written fresh each time. Git reuses tree objects for any folder whose contents did not change at all.
7. A newspaper edition can be recalled and reprinted. A commit object can never be edited. `git commit --amend` makes a completely new one.

39.4.3 PLAIN – a worked example

1. Here is a commit done by hand, one plumbing command per step, so you can see the three stages separately.

```
$ git add greeting.txt

$ git write-tree
14a7508225642503557d7347c657e485225656a1

$ git commit-tree 14a75082... -p HEAD -m "second line"
d5d96f0f76b0571dd21802be5196808779e4b9ba

$ git update-ref refs/heads/main d5d96f0f...

$ git log --oneline
d5d96f0 second line
8c96e11 add greeting
```

2. Step one made a tree from the index. Step two made a commit pointing at that tree and at the old HEAD. Step three moved the branch.
3. `git commit` is those three commands with a message editor bolted on.
4. Now watch the three objects appear for a first commit in an empty repository. Before the commit there is one object, the blob.

```
before commit:
  eb2fc3ca...  (blob)

after commit:
  856b2713...  (tree)
  8c96e114...  (commit)
  eb2fc3ca...  (blob)
```

5. The commit object itself is plain text. Here it is.

```
$ git cat-file -p HEAD
tree 856b271361271d5c57e0ac557a10cc6961ea6806
author Reader <reader@example.com> 1772429400 +0530
committer Reader <reader@example.com> 1772429400 +0530

add greeting
```

6. And the branch file went from not existing at all to holding 40 characters plus a newline.

39.4.4 PLAIN – what is really happening inside

1. Step one, `write-tree`, walks the index in sorted path order.
2. Whenever the path prefix changes, it knows a directory ended, so it builds the tree object for that directory and adds it as an entry in its parent.

3. If the cached tree extension says a directory is unchanged, git skips all of that and reuses the remembered tree name.
4. Step two builds the commit's bytes: the word `tree` and the tree name, then one `parent` line per parent, then `author`, then `committer`, then a blank line, then the message.
5. That text is hashed the same way a blob is, with a header saying `commit` and the byte count, and stored.
6. Step three writes the new commit's name into `.git/refs/heads/<branch>` and appends a line to `.git/logs/HEAD` and `.git/logs/refs/heads/<branch>`. Those log files are the reflog.
7. If HEAD is detached, meaning it holds a commit name directly rather than a branch name, step three writes to `.git/HEAD` instead.
8. Why does the index exist at all? Because it lets you decide what goes in a commit separately from what is on your disk.
9. Without it, either every commit contains everything you have changed, or the tool must ask you a list of files every single time.
10. The honest counter-argument: other systems manage without it. Mercurial has no staging area by default and uses `hg commit --interactive` for the same job. Subversion and Perforce have change lists instead. Fossil and Darcs ask about patches at commit time. So the index is a design choice, not a necessity, and reasonable people call it an unnecessary extra concept.

39.4.5 TECHNICAL - the engineer's version

1. The three plumbing commands are `git write-tree`, `git commit-tree` and `git update-ref`. Every porcelain commit path goes through equivalents.
2. `git commit --amend` does not modify the current commit. It builds a new commit with the same parents and moves the branch. The old commit remains reachable only through the reflog.
3. The commit object's canonical byte layout, in order: `tree <oid>`, zero or more `parent <oid>`, `author <name> <email> <unix-ts> <tz>`, `committer ...`, optional headers such as `gpgsig`, a blank line, message.
4. Author and committer differ whenever a commit is replayed. Rebase, cherry pick and `git am` all keep the author and rewrite the committer.
5. Ref updates go through the ref transaction machinery. With the files backend this is a `.lock` file plus rename per ref. An alternative `reftable` backend was integrated in git 2.45, released April 2024, and is selected at creation time with `git init --ref-format=reftable`.
6. Commit-related knobs worth knowing:

Setting or flag	Effect
`commit.gpgSign`	sign every commit
`git commit -a`	stage tracked modifications first
`git commit --amend`	replace the tip commit
`git commit --fixup X`	message that autosquash recognises
`core.hooksPath`	where pre-commit hooks live

7. Hooks that run: `pre-commit`, `prepare-commit-msg`, `commit-msg`, `post-commit`. A non-zero exit from the first three aborts the commit.
8. Historical note: the index was in git's very first commit. On 7 April 2005 Linus Torvalds committed `cache.h`, containing `#define CACHE_SIGNATURE 0x44495243 /* "DIRC" */`, and the object store then lived in `.dircache/objects`. The tools were called `update-cache`, `write-tree` and `commit-tree`. The names changed; the design did not.

39.4.6 WORDS - remember these

1. Tree object — a folder listing — a git object mapping names to modes and object IDs, one per directory.
2. Commit object — the cover sheet — a git object naming one tree, zero or more parents, author, committer and message.
3. Ref — the sign on the wall — a file under `.git/refs` (or a line in `packed-refs`) holding an object name.
4. Amend — replace the top edition — create a new commit with the old commit's parents and move the branch to it.
5. Committer versus author — who applied it versus who wrote it — two separate identity and timestamp fields in every commit.
6. Reflog — the local diary of ref movements — append-only files under `.git/logs`, local only, never pushed.

39.5 Undoing things, laid out clearly

39.5.1 PLAIN - in simple words

1. Git has many undo commands because there are many different things to undo.
2. The trick is to ask: which of the three places do I want to change?
3. `git restore <file>` changes the working directory only. It copies the staged version over your file, throwing your edits away.
4. `git restore --staged <file>` changes the index only. It copies the HEAD version into the index, so the file is no longer staged.
5. `git reset --soft <commit>` moves HEAD and the branch only. Index and working directory are untouched.
6. `git reset --mixed <commit>`, which is the default, moves HEAD and the branch and resets the index. Working directory untouched.
7. `git reset --hard <commit>` moves all three. Your uncommitted work is gone.
8. `git revert <commit>` changes nothing that already exists. It makes a new commit that undoes an old one. This is the safe undo for shared history.
9. `git clean` deletes untracked files. Nothing about them was ever stored, so they cannot be recovered.
10. `git stash` puts your uncommitted changes aside as hidden commits and gives you a clean tree.

39.5.2 PLAIN - a picture in your head

1. Think of a kitchen with a chopping board, a tray, and a fridge.
2. The chopping board is the working directory. The tray is the index. The fridge is the object database.
3. `git restore` wipes the board and re-copies from the tray.
4. `git restore --staged` empties the tray and refills it from the fridge.
5. `git reset --soft` only moves the label saying which meal you are making. Board and tray keep everything.
6. `git reset --mixed` moves the label and clears the tray.
7. `git reset --hard` moves the label, clears the tray, and scrapes the board into the bin.
8. `git revert` does not touch anything. It cooks a new dish that cancels out an old one, and puts that in the fridge too.

Where this comparison breaks:

9. Scraping the board into a bin is final. `git reset --hard` losing a commit is not final, because the reflog remembers where the branch used to point for 90 days. Losing uncommitted edits, though, really is final.

39.5.3 PLAIN – a worked example

1. Three commits, c1, c2, c3, where the file goes A, then A B, then A B C. Now watch each reset mode from the same starting point.

```
BASELINE  (at c3)
  HEAD    = c3   index = "A B C"   workdir = "A B C"   status: clean

git reset --soft HEAD~1
  HEAD    = c2   index = "A B C"   workdir = "A B C"
  status: M  f.txt      (staged modification)

git reset --mixed HEAD~1
  HEAD    = c2   index = "A B"     workdir = "A B C"
  status:  M f.txt      (unstaged modification)

git reset --hard HEAD~1
  HEAD    = c2   index = "A B"     workdir = "A B"
  status: clean         (the "C" line is gone)
```

2. Read the three status lines. Soft leaves the change staged. Mixed leaves it unstaged. Hard destroys it.
3. Now the other four, in one real session.

```
$ git status --short
A  staged.txt
 M work.txt
?? build.tmp

$ git restore --staged staged.txt
 M work.txt
?? build.tmp
?? staged.txt

$ git restore work.txt
?? build.tmp
?? staged.txt

$ git clean -n
Would remove build.tmp
Would remove staged.txt

$ git clean -f
Removing build.tmp
Removing staged.txt
```

4. Note the `-n` dry run. Always run that first. `git clean -f` deletes files that git has never stored, so there is nothing to recover from.

39.5.4 PLAIN – what is really happening inside

1. Here is the table. Read the middle column as "which of the three places does this touch".

Command	What it changes	Can lose work
`git restore <f>`	working dir only	yes, edits
`git restore --staged`	index only	no
`git reset --soft`	HEAD and branch	no
`git reset --mixed`	HEAD, branch, index	staged only
`git reset --hard`	HEAD, branch, index, wd	yes, badly

Command	What it changes	Can lose work
`git reset <f>`	index entry for one path	no
`git revert <c>`	adds a new commit	no
`git clean -f`	deletes untracked files	yes, forever
`git clean -fdx`	deletes ignored files too	yes, forever
`git stash`	wd and index into a stash	rarely
`git checkout <c>`	HEAD only (detaches)	no

2. Only two rows in that table can destroy something that git has never stored: `git reset --hard` and `git clean`. Everything else is recoverable.
3. `git stash` is not magic. It makes real commits. One commit records the index, another records the working tree, and a ref called `refs/stash` points at the second one.

```
$ git stash list
stash@{0}: On main: half finished idea

$ git log --oneline --graph refs/stash
*   69aa036 On main: half finished idea
|\
| * 77c7641 index on main: b068de2 three hours of important work
|/
* b068de2 three hours of important work
```

4. Now the recovery guide. Suppose you ran `git reset --hard HEAD~1` and threw away a commit.

```
$ git log --oneline
f74729c base                        <- the good commit is gone

$ git reflog
f74729c HEAD@{0}: reset: moving to HEAD~1
b068de2 HEAD@{1}: commit: three hours of important work
f74729c HEAD@{2}: commit (initial): base

$ git reset --hard 'HEAD@{1}'

$ git log --oneline
b068de2 three hours of important work
f74729c base
```

5. The reflog is a local diary of every value HEAD and each branch has held. It is written on every commit, checkout, reset, merge, rebase and pull.
6. Because the commit object was never deleted, only unreferenced, pointing a ref at it again brings it straight back.

39.5.5 TECHNICAL – the engineer's version

1. `git reset` has a fourth and fifth mode. `--merge` resets the index and updates worktree files that differ between HEAD and target while keeping unstaged changes. `--keep` is similar but aborts on conflict.
2. `git reset <paths>` never moves HEAD. Path form and commit form are different operations sharing a name, which is a known wart. `git restore` and `git switch`, introduced in git 2.23 on 16 August 2019, exist to split those roles. They were marked experimental at introduction and remain so in the documentation as of git 2.43.
3. Reflog retention defaults: `gc.reflogExpire` is 90 days for reachable entries, `gc.reflogExpireUnreachable` is 30 days. `gc.pruneExpire` is 2 weeks, so unreachable objects survive at least that long before deletion.

4. Recovery ladder, in the order to try:

Situation	Command
branch moved wrongly	`git reflog` then `reset`
commit lost, not in reflog	`git fsck --lost-found`
dropped a stash	`git fsck --unreachable`
bad rebase, want the old branch	`git reset --hard ORIG_HEAD`
bad merge, not committed	`git merge --abort`
bad rebase, mid-flight	`git rebase --abort`

5. `ORIG_HEAD` is written by merge, rebase, reset and pull before they move HEAD. `git reset --hard ORIG_HEAD` is the fastest undo for those four.
6. Since git 2.36 (April 2022), `git branch --recurse-submodules` and related plumbing exist, but stash and submodules still interact badly; `git stash` does not stash submodule changes by default.
7. `git stash push --staged`, added in git 2.35 (24 January 2022), stashes only what is in the index. `git stash -u` includes untracked files; `git stash -a` includes ignored files.
8. `git stash` was added to git on 30 June 2007 and shipped in git 1.5.3 on 2 September 2007.

39.5.6 WORDS - remember these

1. Soft reset — move the label only — move HEAD and branch, leave index and worktree.
2. Mixed reset — move the label and clear the tray — move HEAD and branch and reset the index; the default mode.
3. Hard reset — move everything and discard — overwrite index and worktree from the target commit.
4. Revert — undo by adding, not removing — create a new commit whose diff is the inverse of an existing commit.
5. ORIG_HEAD — where I was before that big operation — a ref written by merge, rebase, reset and pull.
6. Reflog — the local diary — per-ref append-only history of previous values, kept 90 days by default and never transferred over the network.

39.6 What actually crosses the wire on a push

39.6.1 PLAIN - in simple words

1. When you push, git does not send files. It sends objects, and it sends only the ones the other side does not already have.
2. The first thing that happens is a conversation. Your git starts a program on the server called `receive-pack`.
3. The server speaks first. It lists every branch and tag it has, with the object name each one points at, and a list of features it supports.
4. Your git compares that list with what you are pushing. It works out which objects the server is missing.
5. Your git bundles those objects into one compressed file called a packfile, and sends it, together with a line saying “move branch main from this old name to this new name”.
6. The server reads the packfile, checks that nothing is missing or corrupt, then updates the branch.
7. Then the server sends a short report: whether the pack unpacked, and whether each branch update succeeded.
8. All of that is one connection and, for the object transfer, one round trip. Twenty commits cost the same number of round trips as one.

39.6.2 PLAIN – a picture in your head

1. Think of sending documents to a records office by courier.
2. You phone first. The clerk reads out an index of everything already filed, and says which formats they accept.
3. You compare against your own shelf and put only the missing documents into one box. You do not send twenty separate envelopes.
4. On the box you write one instruction slip: "file these, then change the pointer for folder main from document 1c8a3b1 to document 9978003".
5. The courier carries the box. The clerk unpacks it, checks nothing is missing, files everything, updates the pointer, and phones you back.
6. That phone call back is the report. It is a separate thing from the box.

Where this comparison breaks:

7. A courier can tell you the box arrived. Over a network there is no such guarantee. If the return phone call is cut off, you have no idea whether the clerk filed the documents. That is the whole of section 39.7.

39.6.3 PLAIN – a worked example

1. This is a real push, traced with `GIT_TRACE_PACKET=1`. Lines are trimmed to fit the page; > means sent and < means received.

```
receive-pack> 9978003f... refs/heads/main\0report-status
              report-status-v2 delete-refs side-band-64k quiet
              atomic ofs-delta object-format=sha1 agent=git/2.43.0
receive-pack> 0000
push>         9978003f... e60afb23... refs/heads/main\0
              report-status-v2 side-band-64k quiet
              object-format=sha1 agent=git/2.43.0
push>         0000
   [ packfile bytes travel here ]
receive-pack> unpack ok
receive-pack> ok refs/heads/main
receive-pack> 0000
sideband<     \1000eunpack ok0017ok refs/heads/main0000
```

2. Line one is the advertisement. The server says what it has and what it can do. `0000` ends that list.
3. Line five is the command. Old name, new name, ref name, then the capabilities the client accepts.
4. Then the packfile is streamed.
5. Then the report: `unpack ok` and `ok refs/heads/main`.
6. Now look at the last line very carefully, because it shows the raw framing.
7. `\1` is the sideband channel number: channel 1, meaning real data.
8. `000e` is hexadecimal 14: this packet is 14 bytes long including its own 4-byte length. So 10 bytes of payload: `unpack ok` plus a newline.
9. `0017` is hexadecimal 23: 19 bytes of payload, `ok refs/heads/main` plus a newline.
10. `0000` is the flush packet, meaning "end of this stream".
11. Now the round-trip claim, measured. Twenty commits, each changing one file.

```
$ git rev-list --count origin/main..main
20
$ git rev-list --objects origin/main..main | wc -l
60
$ git push --progress origin main
Enumerating objects: 62, done.
Counting objects: 100% (62/62), done.
Delta compression using up to 2 threads
```

```
Compressing objects: 100% (20/20), done.
Writing objects: 100% (60/60), 3.95 KiB | 674.00 KiB/s, done.
Total 60 (delta 0), reused 0 (delta 0), pack-reused 0
To file:///tmp/gw/srv.git
   1c8a3b1..9978003  main -> main
```

12. Twenty commits became sixty objects, twenty commits plus twenty trees plus twenty blobs, in one packfile of 3.95 KiB, in one exchange.

39.6.4 PLAIN - what is really happening inside

1. Everything on the wire is wrapped in a frame called a **pkt-line**.
2. A pkt-line is four hexadecimal characters giving the total length, then that many bytes minus four of payload.
3. So `0009hello` is a complete packet: length 9, payload `hello` which is 5 bytes, plus the 4 length characters.
4. Three lengths are special. `0000` means flush, end of a section. `0001` is a delimiter used in the newer protocol. `0002` marks the end of a response.
5. On top of pkt-line sits the **sideband**. When both ends agree to it, the first byte of each packet's payload is a channel number.
6. Channel 1 is real data, the packfile or the report. Channel 2 is progress text, which your git prints with the prefix `remote:`. Channel 3 is a fatal error, after which the stream stops.
7. This is why, on a real clone, you see lines beginning with `remote:`. Those came over channel 2.

```
$ git clone https://github.com/octocat/Hello-World.git
Cloning into 'Hello-World'...
remote: Enumerating objects: 13, done.
remote: Total 13 (delta 0), reused 0, pack-reused 13 (from 1)
Receiving objects: 100% (13/13), done.
```

8. `remote: Enumerating objects` is the server talking on channel 2. `Receiving objects` is your own git counting what it reads from channel 1.
9. For fetching, the two sides negotiate. The client says `want <oid>` for what it needs and `have <oid>` for tips it already holds. The server replies `ACK` for the ones it recognises.
10. That is how the server learns the common ancestor without either side sending the whole history.

39.6.5 TECHNICAL - the engineer's version

1. Transports: SSH runs `git-upload-pack` or `git-receive-pack` as a remote command on port 22. Smart HTTP uses two requests on port 443. The `git://` daemon protocol uses port 9418 and has no authentication or encryption.
2. Real smart-HTTP trace against github.com, taken with `GIT_TRACE_CURL=1`, trimmed to the git-relevant headers.

```
=> GET /octocat/Hello-World.git/info/refs?service=git-upload-pack
=> User-Agent: git/2.43.0
=> Git-Protocol: version=2
<= HTTP/1.1 200 OK
<= Content-Type: application/x-git-upload-pack-advertisement

=> POST /octocat/Hello-World.git/git-upload-pack
=> Content-Type: application/x-git-upload-pack-request
=> Git-Protocol: version=2
<= HTTP/1.1 200 OK
<= Content-Type: application/x-git-upload-pack-result
<= Transfer-Encoding: chunked
```

3. The `GET` fetches the advertisement, the `POST` carries the negotiation and receives the packfile. Push uses the same shape with `git-receive-pack`.
4. Real negotiation for a fetch, over protocol v2, trimmed:

```
fetch> command=fetch
fetch> thin-pack
fetch> no-progress
fetch> ofs-delta
fetch> want fec0e27c2650e36592a05049f34cae60fc2966c0
fetch> have 9f1a39e109aca3fc1cce3033a70a43553ebd2bb1
fetch> have f27fb125fe41194710c653bb5b43870a725be980
fetch> 0000
fetch< acknowledgments
fetch< ready
fetch< 0001
fetch< packfile
```

5. Exact framing limits, from the `gitprotocol-common` specification: maximum pkt-line data is 65,516 bytes; maximum total packet length is 65,520 bytes.
6. Sideband limits, from `gitprotocol-capabilities`: `side-band` allows up to 1,000 bytes per packet, 999 of payload plus one band byte. `side-band-64k` allows up to 65,520 bytes, 65,519 of payload plus one band byte.

Capability	Meaning
`report-status`	send a per-ref result after push
`report-status-v2`	adds proc-receive rewrite reporting
`side-band-64k`	multiplex data, progress and errors
`ofs-delta`	deltas may reference by pack offset
`atomic`	all refs update or none do
`delete-refs`	zero OID means delete this ref
`quiet`	suppress server-side progress

7. Protocol v2 was introduced in git 2.18 on 21 June 2018, made default in git 2.26 on 22 March 2020, demoted again in 2.27 after bugs, and made default once more in git 2.29 on 19 October 2020.
8. Push still uses the v0-style advertisement in the trace above, taken with git 2.43.0. Since git 2.32 (6 June 2021), `push.negotiate=true` lets push use v2 negotiation to avoid sending objects the server already has.
9. Smart HTTP itself dates from git 1.6.6, released 23 December 2009, written largely by Shawn O. Pearce. Before that, HTTP fetch was a "dumb" walker that downloaded loose objects one at a time.
10. Observation tools: `GIT_TRACE_PACKET=1`, `GIT_TRACE=1`, `GIT_TRACE_CURL=1` with `GIT_TRACE_CURL_NO_DATA=1`, `GIT_SSH_COMMAND` with `ssh -v`, and `git ls-remote` for a pure read of the server's refs.
11. Be clear about what kind of rule each of these is. The pkt-line framing, the three sideband channels, the capability names and the smart-HTTP URL paths are a **standard**: they are written down in git's protocol documentation, and other implementations follow them.
12. The `remote:` prefix on band 2 output is an **implementation detail** of the git client, defined by `DISPLAY_PREFIX` in `sideband.c`. Nothing on the wire contains those eight characters.
13. Naming your main remote `origin` and your default branch `main` is a **convention**. Git does not care; the tooling around it often does.

39.6.6 WORDS - remember these

1. pkt-line — a length-prefixed frame — 4 hex digits of total length followed by payload; `0000` is flush.
2. Sideband — three streams down one pipe — band 1 data, band 2 progress printed as `remote:`, band 3 fatal error.
3. Packfile — one compressed bundle of objects — the `.pack` format with deltas and an accompanying `.idx`.
4. receive-pack — the server side of push — the process that reads the pack and updates refs.
5. upload-pack — the server side of fetch — the process that negotiates and builds a pack for the client.
6. Capability advertisement — what each end can do — the space-separated list sent after a NUL byte on the first ref line.

39.7 The reader's own failure, analysed properly

39.7.1 PLAIN - in simple words

1. Twice in one session, the reader's push stopped with this message.

```
send-pack: unexpected disconnect while reading sideband packet
fatal: the remote end hung up unexpectedly
```

2. Here is what that sentence means, word by word.
3. `send-pack` is the program on your own machine that does the pushing. So the complaint is coming from your side, not the server's.
4. `reading sideband packet` means your git had finished sending and had switched to listening for the answer.
5. `unexpected disconnect` means the connection went silent while it was listening. Not an error message. Silence.
6. So the stage is exactly this: the pack was already sent, and your git was waiting for the report that says whether it worked.
7. And now the crucial point. Your git never got the report. That is the only fact you have.
8. It does not tell you the push failed. It does not tell you the push succeeded. It tells you that the answer did not arrive.
9. The server may have taken the pack, checked it, moved the branch, and then died before it could say so.
10. Or the server may have died before doing any of that.
11. From your side those two look identical. The only way to find out is to ask the server again.

39.7.2 PLAIN - a picture in your head

1. You post a signed contract by registered mail and wait for the receipt.
2. The receipt never comes.
3. Did the office receive and file your contract, and then lose the receipt in the post? Or did the envelope never arrive at all?
4. You genuinely cannot tell from your end. Both stories produce exactly the same experience: you sent something and heard nothing.
5. The only sane move is to phone the office and ask what they have on file.
6. Note what you must not do. You must not assume it failed and post a second contract, because if the first one landed you now have two.
7. And you must not assume it succeeded and move on, because if it did not you have lost the work.

Where this comparison breaks:

8. With a contract, a duplicate is a real problem. With a git push, re-pushing the identical commits is harmless, because the ref update either has already happened or will happen. That is a lucky property of git, not a general one.

39.7.3 PLAIN - a worked example

1. This can be reproduced exactly. Here is a server that behaves normally, accepts the pack, updates the branch, and then drops the connection instead of delivering its report.

```
$ git ls-remote origin
(nothing: the server has no refs yet)

$ git push origin main
send-pack: unexpected disconnect while reading sideband packet
fatal: the remote end hung up unexpectedly
$ echo $?
128
```

2. That looks like a failure. Now ask the server.

```
$ git ls-remote origin
36e4614a1df6e4ff7839130240981cabb5b970ca    HEAD
36e4614a1df6e4ff7839130240981cabb5b970ca    refs/heads/main

$ git status -sb
## main...origin/main [gone]
```

3. The push worked. The branch is on the server. The error was about the reply, not about the write.
4. Also note `origin/main [gone]`. Your local copy of the server's state was never updated, because the report never arrived. Your machine still thinks nothing is there.
5. Now the second server, which dies before applying anything. Same client, same commit, same command.

```
$ git push origin main
send-pack: unexpected disconnect while reading sideband packet
fatal: the remote end hung up unexpectedly
$ echo $?
128

$ git ls-remote origin
(nothing)
```

6. Identical message. Identical exit code. Opposite outcome.
7. That is the whole lesson in one pair of transcripts. The error text carries no information about whether the write landed.
8. In the reader's own session, both failures had NOT applied. That was discovered by running `git ls-remote` and looking, not by reasoning from the error message. If it had been deduced from the message, the conclusion would have been luck, not knowledge.

39.7.4 PLAIN - what is really happening inside

1. Sequence of events on a push, with the failure point marked.

```
1. client starts receive-pack on the server
2. server sends ref advertisement + capabilities  -> flush
3. client sends "old new refname"  -> flush
4. client streams the packfile
5. server reads pack, runs index-pack, verifies objects
6. server runs pre-receive hook
```

```
7. server updates refs/heads/main
8. server runs post-receive hook
9. server sends "unpack ok" and "ok refs/heads/main"
      <-- the connection dies anywhere from step 5 to step 9
10. client prints the report, updates origin/main
```

2. The client's error is raised in step 9. Everything from step 5 to step 8 may or may not have completed.
3. Concretely: in git's source, `send-pack.c` calls `recv_sideband("send-pack", ...)`. Inside `sideband.c`, when the packet reader returns end-of-file, it produces exactly the message the reader saw.
4. The word `send-pack` in the message is simply the name the caller passed in. It is a label, not a diagnosis.
5. Why does the connection die? Any of: an idle timeout on a middlebox, a carrier-grade NAT table entry expiring, a proxy closing an idle connection, the server process being killed, a load balancer draining, or a genuinely dropped path like the one in the reader's traceroute.
6. Note that step 4, streaming the pack, can take a long time with nothing coming back. That is exactly when idle-connection timers fire.
7. The general shape of the problem: a **non-idempotent** operation, meaning one where doing it twice is not the same as doing it once, sent over a transport that can fail after the request but before the reply.
8. Chapter 36 made the same point about HTTP APIs: `POST` is not idempotent, so a client that times out cannot safely retry, and that is precisely why idempotency keys exist. This is the same problem in a different suit.
9. Git is kinder than most, because pushing the same commits twice is harmless. But you still cannot know the state without asking.

39.7.5 TECHNICAL - the engineer's version

1. The rule, stated properly: **a failure while reading a response tells you nothing about whether the request was applied.** Only the outcome of a fresh query does.
2. This is a two-generals situation. No number of extra acknowledgements makes the final one guaranteed to arrive.
3. Correct procedure after any interrupted push:

```
# what does the server actually hold right now?
git ls-remote origin refs/heads/<branch>

# what did I try to send?
git rev-parse HEAD

# refresh the remote-tracking refs, then read them
git fetch origin
git status -sb
```

4. Do not use `git status` alone. Before a fetch, `origin/main` is a cached value written by the last successful exchange, not a live fact.
5. Techniques that make this class of failure less likely or less painful:

Technique	What it buys you
`git push --atomic`	all-or-nothing across several refs
`--force-with-lease`	fails if the server moved unnoticed
`http.postBuffer` raise	fewer chunked-upload edge cases
SSH `ServerAliveInterval`	keeps NAT and proxy timers alive
`GIT_TRACE_PACKET=1`	shows the exact stage of failure
smaller pushes	shorter windows for a timeout

6. `git push --atomic` uses the `atomic` capability so several ref updates land as one transaction. It does not solve the reporting problem; it only removes the partial-ref-update case.
7. `--force-with-lease` is the strongest safety net here, because it encodes an expectation about the server's current value and fails if reality differs.
8. Generalisation beyond git, worth memorising:

Operation type	Safe to blind-retry?	What to do instead
read (GET, fetch)	yes	just retry
idempotent write	yes	just retry
create (POST)	no	re-query, or use key
git push	in practice yes	re-query first

9. The honest version: git push is described as "in practice safe to retry" because the second push either fast-forwards to the same value or reports "Everything up-to-date". That is a property of the ref-update rule, not a guarantee written in a specification.
10. In the reader's session, the diagnosis of the wider outage came from the same discipline: `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and then timed out after 15 seconds with no reply of any kind, while the same request succeeded over mobile data. What was proven was "no response on this path". What was not proven was who dropped it.

39.7.6 WORDS - remember these

1. Non-idempotent — doing it twice is not the same as once — an operation whose repetition changes the result.
2. Two generals problem — you can never be sure the last message arrived — the classic impossibility result for agreement over a lossy channel.
3. Re-query — go and look rather than guess — issue a fresh read to establish the true state after an ambiguous failure.
4. Stale remote-tracking ref — your cached idea of the server — `origin/main` as written by the last successful fetch or push.
5. Atomic push — all refs or none — the `--atomic` flag, backed by the server's `atomic` capability.
6. Lease — an assertion about the server's current value — the expected old OID carried by `--force-with-lease`.

39.8 Fetch versus pull

39.8.1 PLAIN - in simple words

1. `git fetch` downloads. That is all it does.
2. It brings any objects you are missing into `.git/objects`, and it updates your record of where the server's branches are.
3. It does not touch your files. It does not touch your branch. It does not touch your index. After a fetch, your working directory is byte for byte what it was before.
4. `git pull` is two commands in one: a fetch, then an integration step that combines the fetched work into your current branch.
5. That second step is where the surprises live, because it changes your files and can create commits or conflicts.
6. So `git fetch` is always safe to run. `git pull` is not always safe to run.
7. The usual advice is: fetch, look at what arrived, then decide how to integrate it.

39.8.2 PLAIN – a picture in your head

1. Think of a shared noticeboard in an office, and your own copy of it in a folder.
2. `git fetch` is walking over, photographing the board, and filing the photo. Your desk is untouched. You now know what the board says.
3. `git pull` is walking over, photographing the board, and then immediately rewriting your own working notes to match it.
4. If your notes and the board disagree, that second act is where the mess happens, right there on your desk, without warning.

Where this comparison breaks:

5. A photograph is passive. A fetched object is not: it is a real object in your database that other commands can now reach and use.
6. Copying the board means overwriting. Git never overwrites your commits; it merges or replays. But it does overwrite your working files.

39.8.3 PLAIN – a worked example

1. Here is a fetch, with the state of everything printed before and after.

```
before: HEAD=7f77069  origin/main=1303ba7
before: file f contains "A" and "from x1"

$ git fetch origin
From file:///tmp/gw/ff
   1303ba7..2c979a9  main       -> origin/main

after : HEAD=7f77069  origin/main=2c979a9
after : file f contains "A" and "from x1"

$ git status -sb
## main...origin/main [ahead 1, behind 1]
```

2. Only one thing changed: `origin/main`. HEAD did not move. The file did not change. And now `git status` can tell you the truth: one commit each way.
3. Now a pull in the same situation, on a modern git with nothing configured.

```
$ git pull
hint: You have divergent branches and need to specify how to
hint: reconcile them.
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
fatal: Need to specify how to reconcile divergent branches.
```

4. That refusal is a good thing, and it is recent. Older git versions silently merged, which is how so many repositories acquired hundreds of commits reading “Merge branch ‘main’ of …”.

39.8.4 PLAIN – what is really happening inside

1. `git fetch origin` does five things.
2. Connect and get the ref advertisement.
3. Work out which objects are missing, using `want` and `have` lines.
4. Receive one packfile and store it under `.git/objects/pack/`.
5. Update remote-tracking refs, that is `refs/remotes/origin/*`, to the server’s values.
6. Write `FETCH_HEAD` and append to the reflog. Nothing else moves.

7. `git pull` runs that, then runs either `git merge FETCH_HEAD` or `git rebase FETCH_HEAD`, depending on configuration.
8. The surprise has three separate causes.
9. First, the integration step is invisible in the command name. Nothing in the word "pull" says "and then merge".
10. Second, which integration happens depends on configuration that may differ between your machine and a colleague's.
11. Third, it acts on your working directory, so a bad outcome is immediately in your face, mid-task, possibly with conflicts.

39.8.5 TECHNICAL - the engineer's version

1. `git pull` is documented as `git fetch` followed by either `git merge` or `git rebase`. The choice is made by `pull.rebase`, then `branch.<name>.rebase`, then the command-line flags.
2. Recommended settings, with what each buys you:

Setting	Effect
`pull.ff only`	refuse to auto-merge
`pull.rebase true`	replay your work on top
`fetch.prune true`	delete gone remote-tracking
`push.default simple`	push only the current branch
`push.autoSetupRemote true`	first push sets upstream
`rebase.autoStash true`	stash and restore around it

3. `push.default = simple` became the default in git 2.0, released 28 May 2014. `push.autoSetupRemote` arrived in git 2.37, released 27 June 2022.
4. `git fetch --prune` deletes remote-tracking refs whose branches no longer exist upstream. Without it, `origin/old-feature` lingers forever.
5. `git fetch --all` fetches from every remote. `git remote update` is the older spelling of the same thing.
6. `FETCH_HEAD` is a file listing everything the last fetch brought, with the ones marked "not-for-merge" excluded from the integration step.
7. The refspec controls what is fetched. The default for a clone is `+refs/heads/*:refs/remotes/origin/*`. The leading `+` means "allow non-fast-forward updates to my remote-tracking refs", which is why a force-push upstream does not break your fetch.
8. Shallow and partial fetches change the object set: `--depth=N` truncates history, `--filter=blob:none` omits file contents until needed. Partial clone requires the server capability `filter`, which GitHub's advertisement in the earlier trace does list.
9. The honest version: experts disagree about `pull.rebase`. Rebase-by-default gives a clean line but rewrites your local commits every pull, which is confusing if you have already shared the branch. Merge-by-default is truthful but noisy. `pull.ff only` sidesteps the argument by making you choose each time, and is the safest default to teach.

39.8.6 WORDS - remember these

1. Fetch — download and record — transfer missing objects and update `refs/remotes/*` only.
2. Pull — fetch plus integrate — fetch followed by merge or rebase into the current branch.
3. Remote-tracking ref — your note of where the server was — a ref under `refs/remotes/<remote>/`, updated only by fetch, push or remote commands.
4. Refspec — a mapping of their refs to mine — `+<src>:<dst>`, where `+` allows non-fast-forward updates.
5. FETCH_HEAD — what the last fetch brought — a file listing fetched tips, some marked not-for-merge.
6. Prune — forget branches that no longer exist — `git fetch --prune` or `fetch.prune true`.

39.9 Merge

39.9.1 PLAIN - in simple words

1. Merging combines two lines of work into one.
2. Git starts by finding where the two lines split apart. That shared ancestor is called the **merge base**.
3. It then has three versions of every file: the base version, your version, and their version.
4. For each file it asks: did I change it, did they change it, or both?
5. If only one side changed a file, take that side. If neither changed it, keep the base. If both changed it, look closer.
6. If both changed different parts of the file, take both changes.
7. If both changed the same part, git stops and asks you. That is a conflict.
8. The result is stored as a new commit with two parents, one for each line that was joined.
9. There is one special case. If your branch has not moved at all since the split, there is nothing to combine. Git just moves your branch forward. That is called a **fast-forward**, and it creates no commit.

39.9.2 PLAIN - a picture in your head

1. Two people are given identical copies of a report and go away to edit it.
2. When they come back, an editor puts three copies on the desk: the original, copy A and copy B.
3. For each paragraph the editor compares all three. If only A touched it, take A's. If only B touched it, take B's.
4. If both touched the same paragraph in different ways, the editor cannot decide and leaves both versions marked up for a human.
5. If B never actually changed anything, the editor simply hands over A's copy and calls it done. That is fast-forward.

Where this comparison breaks:

6. A human editor reads for meaning. Git compares lines of text and nothing else. Two changes that read fine together can still be wrong together, and git will never notice. Section 39.10 gives a real example.

39.9.3 PLAIN - a worked example

1. Fast-forward first. Branch `feature` is ahead; `main` has not moved.

```
$ git log --oneline --graph --all
* 80b9042 C: feature file
* 3fc5061 B: on main
* 0a83b6e A: base

$ git merge-base main feature
3fc5061b9e3a37928186aab85f0c7ed2fc6b73c4

$ git merge-base --is-ancestor main feature && echo "ancestor"
ancestor

$ git merge feature
Updating 3fc5061..80b9042
Fast-forward
 feature.txt | 1 +
 1 file changed, 1 insertion(+)

$ git rev-list --merges --count HEAD
0
```

2. Zero merge commits were created, because `main` was already an ancestor of `feature`. Moving the label forward is enough.

3. Now a real three-way merge. Both branches changed the same file in different places.

```
BEFORE
* 1f96d1d B: rename header       (main)
| * 8681015 C: add contact line  (feature)
|/
* 9db1830 A: base page

BASE (A)        OURS (main, B)   THEIRS (feature, C)
header          site header      header
body            body             body
footer          footer           footer
                                 contact
```

4. Line 1 changed on our side only. Line 4 was added on their side only. No overlap, so git can take both.

```
$ git merge --no-edit feature
Auto-merging page.txt
Merge made by the 'ort' strategy.
 page.txt | 1 +

AFTER
*   012c5e8 Merge branch 'feature'
|\
| * 8681015 C: add contact line
* | 1f96d1d B: rename header
|/
* 9db1830 A: base page

$ cat page.txt
site header
body
footer
contact
```

5. And the merge commit itself, which is the only commit in the repository with two `parent` lines.

```
$ git cat-file -p HEAD
tree ba2693f4d0086cd9a8eab7371433d6aa3acdd178
parent 1f96d1d9b440cb9f8b8d53ab39a8becb65a44eda
parent 86810156b0b336611ae572e45ad596dc623886a4
author Reader <reader@example.com> 1772772600 +0530
committer Reader <reader@example.com> 1772772600 +0530

Merge branch 'feature'
```

39.9.4 PLAIN - what is really happening inside

1. Finding the merge base is a graph problem, not a guess. Git walks backwards from both tips, marking which commits are reachable from which side.
2. The merge base is a commit reachable from both tips with no descendant that is also reachable from both. `git merge-base A B` prints it.
3. Sometimes there is more than one such commit. Git's default strategy then merges the bases together first, recursively, and uses the result as the base. That is why the old strategy was called `recursive`.
4. Fast-forward is detected by a single question: is the current tip an ancestor of the target? `git merge-base --is-ancestor A B` answers it with an exit code.
5. If yes, there is nothing to combine, so git writes the target's tree into the index and worktree and moves the branch. No new object is created.

6. `--no-ff` forces a merge commit even when a fast-forward is possible. The reason to use it is that the merge commit is the only record that these commits were a group.

```
with --no-ff:
*   a78fc04 Merge branch 'feature'
|\
| * 52ee392 C (feature)
| * 9aa4587 B (feature)
|/
* 7c451a3 A

$ git log --oneline --first-parent
a78fc04 Merge branch 'feature'
7c451a3 A
```

7. That last command is the payoff of `--no-ff`. `--first-parent` follows only the mainline, so a two-year history of merges reads as one line per feature.
8. The three-way merge itself is applied per file, comparing base, ours and theirs, then per region within the file.

39.9.5 TECHNICAL - the engineer's version

1. Strategies: `ort` is the default, `recursive` is the old default, `resolve` handles a single base only, `octopus` merges more than two heads, `ours` discards the other side's content entirely, `subtree` shifts paths.
2. `ort`, short for "ostensibly recursive's twin", became the default in git 2.34, released 15 November 2021. GitHub reported it outperforming `recursive` by roughly 500 times on rename-heavy merges and by more than 9,000 times on some rebases, largely because it does not use the index as its working data structure.
3. Do not confuse the `ours` strategy with the `-X ours` option. The strategy throws away the other branch's changes. The option only decides conflicting hunks in your favour.

Thing	Effect
`-s ours`	keep our tree entirely
`-X ours`	our side wins conflicting hunks only
`-X theirs`	their side wins conflicting hunks
`-X ignore-all-space`	ignore whitespace when merging
`--no-ff`	always create a merge commit
`--ff-only`	refuse anything but a fast-forward
`--squash`	stage the result, make no merge link

4. Rename detection is on by default, controlled by `merge.renameLimit` and `diff.renames`. It is why moving a file on one side and editing it on the other usually merges correctly.
5. `git merge --abort` restores the pre-merge state using `MERGE_HEAD` and `ORIG_HEAD`. It is safe while a merge is in progress.
6. During a conflicted merge, `.git/MERGE_HEAD`, `.git/MERGE_MSG` and `.git/MERGE_MODE` exist. Their presence is how git knows a merge is unfinished and why `git status` prints "You have unmerged paths".
7. An octopus merge with more than two parents is legal and appears in the Linux kernel history. It refuses to run if any pair conflicts.

39.9.6 WORDS - remember these

1. Merge base — where the two lines split — the best common ancestor of two commits in the DAG.
2. Three-way merge — compare base, ours, theirs — the algorithm that decides per region which side's change to keep.

3. Fast-forward — just move the label — an update where the old tip is an ancestor of the new one, so no merge commit is needed.
4. Merge commit — the join — a commit with two or more parents.
5. ort — the modern merge engine — the default merge strategy since git 2.34, November 2021.
6. First-parent history — the mainline only — `git log --first-parent`, which treats each merge as one entry.

39.10 Conflicts at the file level

39.10.1 PLAIN - in simple words

1. A conflict is not git failing. It is git refusing to guess.
2. Git compares text line by line. It works on regions of lines, not on meaning.
3. If your change and their change are in different regions of the file, git takes both without asking.
4. If they overlap, git cannot know which one you want, so it writes both into the file surrounded by markers and stops.
5. The markers are ugly on purpose. They are meant to be impossible to miss and impossible to leave in by accident.
6. Your job is to edit the file until it is what you actually want, delete the markers, and tell git you are done with `git add`.
7. There is a second, nastier kind of conflict that git will never report. Two changes that merge perfectly as text, but break the program.
8. Git cannot see that, because git does not know what your code means.

39.10.2 PLAIN - a picture in your head

1. Two proofreaders mark up the same page.
2. One changes a word in paragraph two. The other adds a sentence to paragraph nine. The typesetter applies both without a thought.
3. Both change the same sentence in paragraph five, differently. The typesetter stops, prints both versions one above the other with a big line between them, and puts the page in the "needs a decision" tray.
4. Now the nasty case. One proofreader renames a character from Anna to Anya throughout chapter one. The other adds a new scene in chapter nine mentioning Anna.
5. No paragraph overlaps. The typesetter merges happily. The book is now inconsistent, and nothing in the process caught it.

Where this comparison breaks:

6. A typesetter might notice the name change by reading. Git never reads. It will merge a file into nonsense with complete confidence, every time.

39.10.3 PLAIN - a worked example

1. Both branches change the same line of a config file.

```
BASE            OURS (main)      THEIRS (feature)
PORT = 8080     PORT = 3000      PORT = 9090

$ git merge feature
Auto-merging config.py
CONFLICT (content): Merge conflict in config.py
Automatic merge failed; fix conflicts and then commit.

$ cat config.py
HOST = "localhost"
<<<<<<< HEAD
```

```
PORT = 3000
=======
PORT = 9090
>>>>>>> feature
TIMEOUT = 30
```

2. Line by line. `<<<<<<< HEAD` opens the conflicted region and names your side. Everything until `=======` is your version.
3. `=======` is the divider. Everything after it until `>>>>>>>` is their version. `>>>>>>> feature` closes the region and names their side.
4. The unconflicted lines, `HOST` and `TIMEOUT`, are outside the markers and have already been merged.
5. The default marker style hides the base. Turning on `diff3` shows it, which is much more useful.

```
$ git checkout --conflict=diff3 config.py
$ cat config.py
HOST = "localhost"
<<<<<<< ours
PORT = 3000
||||||| base
PORT = 8080
=======
PORT = 9090
>>>>>>> theirs
TIMEOUT = 30
```

6. Now you can see that the original was 8080, so both sides changed it, and neither side is a superset of the other.
7. Meanwhile, the index holds all three versions at once.

```
$ git ls-files -u
100644 5c03df4f...9181 1    config.py
100644 bd085f95...ca9b 2    config.py
100644 ee2a92c1...4f85 3    config.py
```

8. Stage 1 is the base, stage 2 is ours, stage 3 is theirs. Resolving means replacing those three entries with one stage-0 entry, which is exactly what `git add` does.

39.10.4 PLAIN - what is really happening inside

1. Now the semantic conflict. This one is real, run end to end.
2. Start with two files. `lib.py` defines `get_user`. `app.py` calls it.
3. On a branch called `refactor`, rename the function to `fetch_user` and fix the one call site that exists.
4. On `main`, add a brand new file `report.py` that calls `get_user`.
5. Neither side touched a line the other touched. Watch git.

```
$ git merge refactor
Merge made by the 'ort' strategy.
 app.py | 4 ++--
 lib.py | 2 +-
 2 files changed, 3 insertions(+), 3 deletions(-)

$ python3 -c "import report; report.report()"
  File "/tmp/gw/sem/report.py", line 1, in <module>
    from lib import get_user
ImportError: cannot import name 'get_user' from 'lib'
```

6. Zero conflicts. Zero warnings. Broken program.

7. This is not a bug. Git did exactly what it promises: combine text changes that do not overlap. It never promised the result would compile.
8. The only defence against semantic conflicts is a test suite that runs on the merge result, not on either branch alone. That is precisely what continuous integration on the merged head is for.
9. Binary files are the other special case. Git cannot combine regions of a JPEG or a spreadsheet, so any change on both sides is a whole-file conflict.

```
$ git merge designer
warning: Cannot merge binary files: logo.png (HEAD vs. designer)
CONFLICT (content): Merge conflict in logo.png

$ git status --short
UU logo.png
```

10. There are no markers in the file. The working copy is left as one of the two versions. You choose a side with `git checkout --ours` or `--theirs`, then `git add`.

39.10.5 TECHNICAL - the engineer's version

1. Conflict styles, set by `merge.conflictStyle`:

Style	Shows	Since
`merge`	ours and theirs only	always (default)
`diff3`	ours, base, theirs	long-standing
`zdiff3`	diff3 with common lines out	git 2.35, 2022

2. `zdiff3` arrived in git 2.35, released 24 January 2022. It hoists lines common to both sides out of the conflict region, making the real disagreement smaller.
3. The resolution workflow, in full:

```
git status                  # which paths are unmerged
git diff                    # combined diff of conflicts only
git mergetool               # optional three-pane editor
<edit files, remove markers>
git add <path>              # collapses stages 1,2,3 into stage 0
git status                  # confirm nothing is unmerged
git commit                  # message is prefilled from MERGE_MSG
```

4. `git mergetool` drives an external tool named by `merge.tool`. It creates `*.BASE`, `*.LOCAL` and `*.REMOTE` temporary files from index stages 1, 2 and 3, and removes them afterwards unless `mergetool.keepBackup` is set.
5. `git checkout --ours <path>` and `--theirs <path>` pull stage 2 or stage 3 out of the index. During a rebase these are reversed relative to intuition, because the commit being replayed is "theirs".
6. **rerere** stands for "reuse recorded resolution". Enable it with `git config rerere.enabled true`. Git then records the conflict text and your resolution, and replays it automatically next time the same conflict appears.

```
$ git config rerere.enabled true
$ git merge feature
Recorded preimage for 'cfg.py'
<resolve by hand, git add, git commit>
Recorded resolution for 'cfg.py'.

$ git reset --hard HEAD~1
$ git merge feature
Resolved 'cfg.py' using previous resolution.
```

```
$ cat cfg.py
PORT = 3000
```

7. The recordings live in `.git/rr-cache/<hash>/preimage` and `postimage`. The hash is computed from the normalized conflict text, so an identical conflict anywhere gets the same resolution.
8. rerere was written by Junio C Hamano and merged on 6 February 2006, with the commit message "git-rerere: reuse recorded resolve". It is most valuable during long rebases, where the same conflict recurs on every replayed commit.
9. For binary formats, `.gitattributes` with `*.png binary` or a custom `merge=` driver is the practical answer. Git LFS stores a pointer file, so conflicts appear on the pointer rather than the payload, which is smaller but no easier to merge.

39.10.6 WORDS - remember these

1. Conflict — git refusing to guess — overlapping changes recorded as index stages 1, 2 and 3 with markers in the worktree.
2. Conflict marker — the ugly fence — `<<<<<<<`, `=======`, `>>>>>>>`, plus `|||||||` for the base in diff3 style.
3. Semantic conflict — merges fine, breaks anyway — a logically incompatible pair of changes that do not overlap textually.
4. rerere — remember how I fixed this — recorded conflict resolutions replayed automatically.
5. Unmerged path — a file still in dispute — a path with index entries at stages other than 0.
6. Merge driver — custom combining logic for a file type — a program named in gitattributes via `merge=<name>`.

39.11 Rebase

39.11.1 PLAIN - in simple words

1. Rebase means: take my commits, and pretend I had started from somewhere else.
2. Git works out which commits are mine, sets them aside, moves my branch to the new starting point, and applies each of my changes again, one at a time.
3. Each time it applies one, it makes a **new commit**. Not the old one moved. A new object, with a new name.
4. It must be a new object, because a commit records the name of its parent, and the parent is now different. Change any byte and the name changes.
5. So after a rebase your work looks the same but every commit has a different identifier.
6. The result is a straight line instead of a fork, which many teams prefer to read.
7. Rebase can also edit history as it replays: reorder commits, reword messages, squash several into one, or drop one entirely. That is interactive rebase.

39.11.2 PLAIN - a picture in your head

1. You wrote three pages of notes on top of page 10 of a shared notebook.
2. Meanwhile the shared notebook grew to page 14.
3. Merging means gluing your three pages in as a side branch, with a note saying where they joined.
4. Rebasing means copying your three pages out by hand onto fresh sheets, placing them after page 14, and burning the originals.
5. The words on your new sheets are identical. The sheets are not the same sheets.

Where this comparison breaks:

6. Copying by hand can introduce mistakes. Rebase reapplies changes exactly, but it can hit conflicts, because the surrounding text is now different.

7. The originals are not really burned. They stay in the reflog for 90 days.

39.11.3 PLAIN - a worked example

1. A branch with two commits, and `main` has moved on.

```
BEFORE
* a91e93d M1: main moved on          (main)
| * 9ca2c92 F2: extend feat.txt      (feature)
| * 26e79b3 F1: add feat.txt
|/
* c32f284 A: base

$ git switch feature
$ git rebase main
Successfully rebased and updated refs/heads/feature.

AFTER
* 6a7f276 F2: extend feat.txt        (feature)
* 003972b F1: add feat.txt
* a91e93d M1: main moved on          (main)
* c32f284 A: base
```

2. The hashes side by side.

Commit	Before rebase	After rebase
F1	26e79b3	003972b
F2	9ca2c92	6a7f276
tip	9ca2c92	6a7f276

3. Two commits went in, two came out, and not one identifier survived.
4. Interactive rebase on a messy branch. This is the todo list git presents.

```
$ git rebase -i HEAD~4
pick 7644785 add parser
pick bd687cc wip
pick 894e838 fix typo in parser
pick 0ec5c9b add tests
```

5. Edit it to this, and save.

```
pick   7644785 add parser
fixup  bd687cc wip
fixup  894e838 fix typo in parser
reword 0ec5c9b add tests
```

6. The result: five commits become three, the file content is byte for byte identical, and the last message was rewritten.

```
$ git log --oneline
c868f3c add unit tests for the parser
4dcdb2e add parser
292e19e base
```

39.11.4 PLAIN - what is really happening inside

1. `git rebase <upstream>` runs these steps.
2. Work out the commit list: everything reachable from HEAD but not from upstream, oldest first. That is `git rev-list upstream..HEAD --reverse`.

3. Check out the upstream tip, with HEAD detached.
4. For each commit in the list, compute its change against its own parent and apply that change to the current state.
5. Create a new commit with the same message, the same author and author date, a **new committer date**, and the new parent.
6. When the list is finished, move the branch ref to the last new commit and re-attach HEAD to it.
7. Step 5 is why the hash changes: the parent line differs, the tree usually differs, and the committer timestamp differs. Chapter 38 showed that the hash is computed over exactly those bytes.
8. If applying a change fails, the rebase stops and asks you to fix it, then `git rebase --continue`. `git rebase --abort` restores everything.
9. Interactive rebase is the same loop, but the list is written to a todo file you may edit first.

Verb	What it does
`pick`	replay this commit unchanged
`reword`	replay it, open the editor for the message
`edit`	replay it, then stop so you can amend
`squash`	fold into the previous, combine both messages
`fixup`	fold into the previous, discard this message
`drop`	do not replay it at all
`exec`	run a shell command at this point
`break`	stop here, continue later

10. `git commit --fixup <sha>` writes a message starting `fixup!`. Running `git rebase -i --autosquash` then reorders that commit next to its target and marks it `fixup` for you, with no editing by hand.

39.11.5 TECHNICAL - the engineer's version

1. Two backends exist. The `apply` backend uses `format-patch` and `git am`. The `merge` backend replays with the merge machinery and is the default since git 2.26 (March 2020). `--empty`, `--keep-empty` and `--reapply-cherry-picks` control what happens to commits that become empty.
2. `git rebase --onto <newbase> <upstream> <branch>` moves only the commits that are on `branch` but not on `upstream`. It is the tool for detaching a topic built on another topic.

```
$ git rebase --onto main topic1 topic2
```

3. `git rebase -i --autosquash` was added by Nanako Shiraishi on 6 January 2010 and shipped in git 1.7.0 on 12 February 2010. Set `rebase.autoSquash true` to make it automatic.
4. `git rebase --exec 'make test'` runs a command after each replayed commit, so you can prove every commit in the branch builds, not just the tip.
5. `rebase.autoStash true` stashes uncommitted work before the rebase and restores it afterwards.
6. History: `git-rebase-script` was added by Junio C Hamano on 25 June 2005, with the message "rebase local commits to new upstream head". Interactive mode was added by Johannes Schindelin on 24 June 2007 and shipped in git 1.5.3 on 2 September 2007.
7. `git rebase` sets `ORIG_HEAD` to the pre-rebase tip, and writes reflog entries labelled `rebase (start)`, `rebase (pick)`, `rebase (fixup)` and `rebase (finish)`, so every intermediate state is recoverable.

39.11.6 WORDS - remember these

1. Rebase — replay my work on a new base — create new commits with the same changes and different parents.
2. Upstream — the thing I am rebasing onto — the commit that becomes the new parent of the first replayed commit.
3. Interactive rebase — edit the plan before replaying — `git rebase -i` with a todo list of verbs.

4. Autosquash — fold marked fixes automatically — reorder and mark `fixup!` and `squash!` commits during `rebase -i`.
5. Committer date — when this object was made — rewritten on every replay, unlike the author date.
6. Todo list — the rebase plan — the editable file of `pick`, `squash`, `fixup` and other verbs.

39.12 The reader's rebase, explained

39.12.1 PLAIN - in simple words

1. In the reader's session, a local branch was rebased onto a `main` that had moved, and the branch's SHA changed.
2. Nothing went wrong. That is the only possible outcome.
3. A commit's name is computed from its exact bytes. Those bytes include the name of its parent.
4. Rebasing gives every replayed commit a different parent. Different parent means different bytes, which means a different name.
5. So the changes are the same and the commits are not the same objects.
6. The old commits still exist. They are simply no longer pointed at by the branch.

39.12.2 PLAIN - a picture in your head

1. Two identical recipes, each written on a card that also says which card comes before it in the box.
2. Move one recipe to a different position in the box and the "comes after" line must change.
3. The ingredients are identical. The card is not the same card, because the card includes that line.

Where this comparison breaks:

4. You could rub out the line on a real card. A git object's name is derived from its bytes, so editing the bytes produces a different object rather than an edited one.

39.12.3 PLAIN - a worked example

1. The same commit, before and after rebase, printed in full.

```
OLD F1 (26e79b3)
tree   d18b4ea7a3728589bfef3b1836e869ac1db45faf
parent c32f2845c02885738408ebd404b13b9f2e3c7c4e
author    Reader <reader@example.com> 1772941200 +0530
committer Reader <reader@example.com> 1772941200 +0530

NEW F1 (003972b)
tree   ed1a2fa48bd51d2a79edc8c6e3628c892e5d4c9b
parent a91e93d9b813a53be8d8b7f51733815d0de72dea
author    Reader <reader@example.com> 1772941200 +0530
committer Reader <reader@example.com> 1772942400 +0530
```

2. Three of the four lines differ. The parent is a different commit. The tree is different because it now also contains main's new version of `core.txt`. The committer timestamp is when the replay happened.
3. Only the author line survived, which is why `git log` still shows your original date.
4. Now recompute the new name by hand, exactly as Chapter 38 did.

```
$ git cat-file commit 003972b > c.raw
$ wc -c < c.raw
219
$ { printf 'commit %d\0' 219; cat c.raw; } | sha1sum
003972bffe4b385f957000613aa02ea8e5208924
$ git rev-parse 003972b
003972bffe4b385f957000613aa02ea8e5208924
```

5. The hash is not stored anywhere. It is arithmetic over those 219 bytes, and anyone can repeat it.

39.12.4 PLAIN - what is really happening inside

1. The change each commit introduces really is identical. Git can prove that with a patch identifier, a hash of the diff with line numbers and whitespace normalized away.

```
$ git show 26e79b3 | git patch-id --stable
3644a7626a72280f69af9830a10cf2f013748966 26e79b3d...
$ git show 003972b | git patch-id --stable
3644a7626a72280f69af9830a10cf2f013748966 003972bf...
```

2. Same patch identifier on the left. Different commit identifier on the right. That is the whole story in two lines.
3. The old commits are still in the object database, reachable through the reflog.

```
$ git reflog feature
6a7f276 feature@{0}: rebase (finish): onto a91e93d
9ca2c92 feature@{1}: commit: F2: extend feat.txt
26e79b3 feature@{2}: commit: F1: add feat.txt
c32f284 feature@{3}: branch: Created from HEAD
```

4. So the pre-rebase branch is one command away: `git reset --hard feature@{1}`.
5. Practical consequence for the reader: after that rebase, the branch on the server and the branch on the machine had no commits in common past the base. A plain push would be refused as non-fast-forward, and a forced push would be required. Section 39.13 covers when that is acceptable.

39.12.5 TECHNICAL - the engineer's version

1. The invariant: SHA-1 over `commit <len>\0` plus the commit body. Any change to tree, parent, author, committer or message yields a different object ID.
2. `git patch-id --stable` produces a diff-content identity that survives rebase, cherry-pick and reordering. `git cherry` and `git rebase` use it to skip changes already present upstream.
3. `git range-diff <base>..<old> <base>..<new>` compares two versions of the same series and is the correct tool for reviewing a rebased branch. It was added in git 2.19, September 2018.
4. What survives and what does not:

Property	Survives a rebase?
author name and date	yes
commit message	yes, unless edited
the diff (patch-id)	yes
tree object ID	usually not
parent object ID	no
committer date	no
commit object ID	no
GPG signature	no, must re-sign

5. Signed commits are invalidated by rebase, because the signature covers the commit bytes. `git rebase -S` re-signs each replayed commit.
6. Because remote-tracking refs are updated only by network operations, a rebase makes `git status` report a large "diverged" count immediately. That is arithmetic on the graph, not a warning about anything being wrong.

39.12.6 WORDS - remember these

1. Patch identifier — the fingerprint of a change — a hash of the normalized diff, unchanged by rebase.
2. Rewritten history — the same work, new objects — a series replaced by a replayed series with different IDs.
3. Diverged — no longer a straight line — two refs whose merge base is behind both of them.
4. range-diff — compare two versions of a series — `git range-diff`, added in git 2.19, September 2018.
5. Re-signing — the signature does not follow — GPG signatures must be recreated after any rewrite, with `git rebase -S`.
6. Reflog recovery — the pre-rebase tip is still there — `branch@{1}` names the value the branch held before the last update.

39.13 When rebasing is safe and when it destroys work

39.13.1 PLAIN - in simple words

1. There is one rule, and it is short. Do not rebase commits that other people already have.
2. Rebasing your own branch, before anyone else has fetched it, is safe and good practice. It makes your work easy to read and easy to review.
3. Once you have pushed a branch and a colleague has fetched it, their machine holds those exact commit objects.
4. If you rebase and force-push, the server now has different objects with the same branch name.
5. Your colleague's git still has the originals, plus anything built on top of them. Their next pull tries to combine two versions of the same work.
6. The result is duplicated commits, avoidable conflicts, and confusion about which version is real.
7. Main, master, develop and release branches are shared by definition. Never rebase those.

39.13.2 PLAIN - a picture in your head

1. You hand out photocopies of a chapter to five reviewers.
2. Then you rewrite the chapter and renumber every paragraph.
3. Each reviewer's comments now point at paragraph numbers that mean something different, or nothing at all.
4. Nobody's work is destroyed, but reconciling it is now manual and tedious for five people instead of one.

Where this comparison breaks:

5. Reviewers can see that the chapter changed. Git cannot. It sees two unrelated sets of commits with the same branch name and tries to merge them, which is how duplicate commits appear in the log.

39.13.3 PLAIN - a worked example

1. Alice and Bob both have commit B. Bob builds C on top of it. Alice amends B, which is a one-commit rewrite, and force-pushes.

```
Alice: 2bd9139 B: with a better message   (rewritten)
       fac214d A

Bob:   4370404 C: bob's work on top of B
       426111e B                          (the original)
       fac214d A
```

2. Bob fetches, and his repository now contains both versions.

```
* 2bd9139 B: with a better message   (origin/main)
| * 4370404 C: bob's work on top of B  (main)
```

```
| * 426111e B
|/
* fac214d A
```

3. Bob pulls with the default merge behaviour, and gets a conflict in a file neither of them meant to fight over.

```
$ git pull
Auto-merging f
CONFLICT (content): Merge conflict in f
```

4. The fix Bob needs is `git pull --rebase`, which replays only his own C onto the new B.

```
$ git pull --rebase
Successfully rebased and updated refs/heads/main.
$ git log --oneline
0b6514b C: bob's work on top of B
2bd9139 B: with a better message
fac214d A
```

5. Note that Bob had to know what happened. Git gave him no message saying "the upstream branch was rewritten".

39.13.4 PLAIN - what is really happening inside

1. A normal push is only allowed if the old value on the server is an ancestor of the new value. That is the fast-forward rule.

```
$ git push origin main
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs
hint: Updates were rejected because the remote contains work
hint: that you do not have locally.
```

2. `--force` switches that check off entirely. Whatever the server had is replaced, and any commit only reachable from the old value becomes garbage.
3. `--force-with-lease` replaces the check with a different one: the server's current value must equal what your remote-tracking ref says it is.
4. So `--force-with-lease` says "overwrite, but only if nobody has pushed since I last looked". Here is it refusing.

```
my origin/main says : 2bd9139
the server really has: 0b6514b

$ git push --force-with-lease origin main
 ! [rejected]        main -> main (stale info)
error: failed to push some refs

$ git push --force origin main
 + 0b6514b...a848939 main -> main (forced update)
```

5. The plain `--force` succeeded and Bob's commit C is now unreachable on the server. Only Bob's own clone still has it.
6. One honest warning: `--force-with-lease` compares against your remote-tracking ref, and a background `git fetch` updates that ref without you noticing. Then the lease is satisfied and the protection is gone. Use `--force-with-lease=<ref>:<expected-oid>` when it matters.

39.13.5 TECHNICAL - the engineer's version

1. The decision table:

Situation	Rebase?	Then push with
local branch, never pushed	yes	plain push
pushed, nobody else uses it	yes	–force-with-lease
open PR, reviewers on it	prefer not	–force-with-lease
shared feature branch, 2+ people	no	merge instead
main, master, develop, release	never	never force
tags already published	never	never move

2. `--force-if-includes`, added in git 2.30 (December 2020), strengthens the lease by also requiring that your local history includes everything your remote-tracking ref has seen. Turn it on with `push.useForceIfIncludes true`.
3. Server-side defences: `receive.denyNonFastForwards`, and on hosted services, protected branches that reject force pushes outright.
4. Recovery when a force push destroys a colleague's commit: the commit is still in that colleague's clone, and on GitHub-style services it is often still reachable through the reflog on the server or through the events API.
5. `git rebase` on a branch that has an open pull request is common practice, and opinions differ. One camp rebases and force-pushes to keep the series clean; the other adds fixup commits during review and squashes at merge time so that review comments stay anchored. Both are defensible.
6. `git range-diff` makes the first camp's approach reviewable, because it shows what changed between the old series and the new one.

39.13.6 WORDS - remember these

1. Published history — commits other people have — anything that has been pushed and fetched by someone else.
2. Fast-forward rule — you may only extend — the default push check that the old tip is an ancestor of the new one.
3. Force push — replace whatever is there — `--force`, which switches the ancestry check off.
4. Lease — only if it is still what I saw — `--force-with-lease`, which compares against your remote-tracking ref.
5. Protected branch — the server says no — a hosting-service rule rejecting force pushes and deletions.
6. Duplicate commits — the same work twice — the visible symptom of merging a rewritten branch with its original.

39.14 Merge versus rebase as a team policy

39.14.1 PLAIN - in simple words

1. Both merge and rebase produce a repository containing the same final files. Neither is more correct than the other.
2. The argument is about what the history looks like afterwards, and how easy it is to work with.
3. Merging keeps a true record: this work happened on a side branch, and it was joined on this date.
4. Rebasing produces a straight line that reads like a story, but the story is tidied. It never happened quite that way.
5. Squash merging goes further: an entire branch becomes one commit on the mainline, and the individual steps are discarded.
6. Teams pick one, write it down, and enforce it with a setting on the hosting service. The worst outcome is a repository where everyone does something different.

39.14.2 PLAIN - a picture in your head

1. Three ways to write up a two-week experiment.
2. Merge: the full lab notebook, every dead end included, with dates.
3. Rebase: the same experiment rewritten as a clean sequence of steps that leads to the result.
4. Squash: a one-paragraph abstract.
5. All three are honest about the result. They differ in what you can go back and inspect a year later.

Where this comparison breaks:

6. A lab notebook cannot be rewritten without fraud. Git history is a tool for communication, and tidying it is normal and legitimate.

39.14.3 PLAIN - a worked example

1. The same feature, three policies, as `git log --oneline --graph`.

```
MERGE (--no-ff)
*   a78fc04 Merge branch 'feature'
|\
| * 52ee392 C (feature)
| * 9aa4587 B (feature)
|/
* 7c451a3 A

REBASE then fast-forward
* 6a7f276 C (feature)
* 003972b B (feature)
* 7c451a3 A

SQUASH
* 400e7f2 Add the whole feature (squashed)
* 040d29f A
```

2. One extra fact about squash merging that surprises people.

```
$ git merge-base --is-ancestor feature main && echo yes || echo no
no
```

3. After a squash merge, git does not know the branch was merged. Merging it again would try to replay B, C and D. Delete squashed branches immediately.

39.14.4 PLAIN - what is really happening inside

1. The arguments for merge.
2. It records what actually happened, including when parallel work existed.
3. It never rewrites objects, so nothing anybody has fetched ever becomes invalid.
4. `git log --first-parent` collapses each feature to one line, giving a clean mainline view on demand.
5. Reverting a whole feature is one command, `git revert -m 1 <merge>`.
6. The arguments for rebase.
7. A linear history is easier to read, and `git log` needs no graph rendering.
8. Every commit on the mainline was tested against the code it actually sits on top of, which makes bisect more reliable.
9. There are no merge commits whose diff is confusing to review.
10. The arguments for squash.
11. One commit per feature is the simplest possible mainline.
12. Reviewers' intermediate "fix typo" commits never reach the main branch.
13. The cost: bisect lands on a large commit, and the intermediate steps are gone unless the branch is kept.

14. The honest summary: this is a readability and bisect-ability argument, not a correctness argument. Every option produces identical file contents. Experts disagree, and both large, successful projects and small teams run all three policies happily.

39.14.5 TECHNICAL - the engineer's version

1. Comparison of the three:

Property	Merge	Rebase	Squash
rewrites objects	no	yes	yes
mainline is linear	no	yes	yes
keeps step-by-step	yes	yes	no
bisect granularity	fine	fine	coarse
revert whole feature	easy	manual	easy
force push needed	no	often	no

2. The Linux kernel merges: Linus Torvalds pulls from maintainers, producing a deeply merged history, and explicitly asks maintainers not to rebase published trees. Many web and application teams squash-merge every pull request.
3. `git config merge.ff false` on a specific branch forces merge commits. `git config pull.ff only` prevents accidental merges on pull.
4. On hosting services the equivalent settings are the allowed merge methods for pull requests: merge commit, squash and merge, rebase and merge.
5. Bisect interacts with policy. With merge-based history, `git bisect` can land on a commit inside a feature branch that was never tested alone. `git bisect --first-parent`, added in git 2.29 (October 2020), restricts the search to the mainline and avoids that.
6. Commit hygiene matters more than the policy. A branch of ten commits each of which builds and passes tests is valuable under any of the three; a branch of ten commits named "wip" is not.

39.14.6 WORDS - remember these

1. Linear history — one line, no forks — a history where every commit has exactly one parent.
2. Squash merge — one commit per feature — combine a branch into a single commit with no merge link.
3. First-parent view — the mainline only — following only the first parent of each merge.
4. Bisect-ability — how precisely a search can land — a property of how large and how independently testable each commit is.
5. Merge method — the button the service offers — merge commit, squash and merge, or rebase and merge.
6. Commit hygiene — each commit is a whole, working step — the practice that makes any policy work.

39.15 Cherry-pick, revert, bisect and the rest

39.15.1 PLAIN - in simple words

1. `git cherry-pick` copies one commit's change onto your current branch. Use it to move a single fix without moving everything around it.
2. `git revert` creates a new commit that undoes an old one. Use it when the bad commit is already public.
3. `git bisect` finds which commit introduced a bug, by testing a shrinking range automatically.
4. `git blame` shows which commit last changed each line of a file.
5. `git log -S"text"` finds commits where a piece of text was added or removed.
6. `git worktree` gives you a second checked-out directory sharing one object database, so you can look at another branch without stashing.

39.15.2 PLAIN - a picture in your head

1. Bisect is the game of guessing a number between 1 and 1000 with yes-or-no questions.
2. Ask "is it above 500?" and half the possibilities vanish, whatever the answer.
3. Ten questions cover a thousand numbers; twenty cover a million.
4. Git plays the same game over commits. Each test halves the range.

Where this comparison breaks:

5. The number game assumes one answer. Bisect assumes the bug appeared once and stayed. If it comes and goes, bisect will mislead you. Mark such commits `git bisect skip`.

39.15.3 PLAIN - a worked example

1. Thirty-one commits. Somewhere in there, `average([1,2,4])` started returning 2.0 instead of 2.333.

```
$ ./test.sh
AssertionError: average([1,2,4]) gave 2.0
```

2. Give git a known-bad commit and a known-good commit, and a script that exits 0 for good and non-zero for bad.

```
$ git bisect start HEAD 8c84483
$ git bisect run ./check.sh
Bisecting: 14 revisions left to test (roughly 4 steps)
[e07d528] c015: routine change
running './check.sh'
Bisecting: 7 revisions left to test (roughly 3 steps)
[56821a0] c022: routine change
running './check.sh'
Bisecting: 3 revisions left to test (roughly 2 steps)
[bb1e2b9] c018: routine change
running './check.sh'
Bisecting: 0 revisions left to test (roughly 1 step)
[a83bd2e] c017: routine change
running './check.sh'
bb1e2b9ca8de5e3324381ae15dec599b8c409a9f is the first bad commit
```

3. Four tests over thirty commits, because log base 2 of 30 is about 4.9.
4. And the guilty change, which was hidden among unrelated edits.

```
-    return sum(nums) / len(nums)
+    return sum(nums) // len(nums) + 0.0
```

5. Finish with `git bisect reset`, which puts you back where you started.

39.15.4 PLAIN - what is really happening inside

1. Cherry-pick computes the diff of one commit against its parent, applies it here, and makes a new commit. The change is the same; the object is new.

```
original S   = 9e25370ab0a266f7b25787b2a08e719f8dd9d964
copy on main = d673ae20eb28f1b43254cad402c16571ec99e9b3

patch-id of both = c92b260713d372c3c1c2babe11f8384d5319ef45
```

2. `git cherry -v main hotfix` uses that patch identifier to say which commits on `hotfix` are already on `main`. A leading `-` means already there, `+` means not yet.
3. Revert computes the inverse diff and commits it. Reverting a merge needs `-m 1` to say which parent's line of history to keep.

```
$ git revert HEAD
error: commit 88e4ee8... is a merge but no -m option was given.

$ git revert -m 1 --no-edit HEAD
```

4. Bisect works on the graph. `git rev-list --bisect` picks the commit that splits the remaining candidates most evenly, so each answer removes about half regardless of which answer it is.
5. `git bisect run <cmd>` automates it. Exit code 0 means good, 1 to 124 and 126 to 127 mean bad, and exit code 125 means "cannot test, skip this one".

39.15.5 TECHNICAL - the engineer's version

1. The wider toolbox:

Command	Use it when
`git cherry-pick -x`	you want a "cherry picked from"
`git cherry-pick -n`	stage without committing
`git revert -n`	revert several as one commit
`git bisect skip`	this revision cannot be tested
`git bisect log/replay`	save and rerun a bisect session
`git blame -C -M`	follow copies and moves
`git log -S<string>`	find when a string appeared
`git log -G<regex>`	find diffs matching a pattern
`git worktree add`	second checkout, one object store
`git reflog expire`	prune the local diary

2. History: `git rev-list --bisect` was added by Linus Torvalds on 17 June 2005, and `git-bisect-script` followed on 30 July 2005 with the commit message "Making it easier to find which change introduced a bug".
3. `git bisect run` takes any command, so the test can be a unit test, a build, a `grep` on the output, or a shell one-liner. It is the highest-value five minutes in this chapter.
4. `git bisect --first-parent` (git 2.29, October 2020) confines the search to the mainline, which matters in merge-heavy repositories.
5. `git worktree` shares `.git/objects` and refs across checkouts, so a second worktree costs disk space for files only, not for history.
6. `git log -S` is a "pickaxe" search on the number of occurrences of a string; `git log -G` matches the diff text with a regular expression. They answer different questions, and `-S` is usually the one you want.

39.15.6 WORDS - remember these

1. Cherry-pick — copy one change here — apply one commit's diff and create a new commit for it.
2. Revert — undo by adding — commit the inverse of an earlier commit.
3. Bisect — binary search over history — halve the candidate range with each test until one commit remains.
4. Pickaxe — find when text appeared — `git log -S<string>`.
5. Worktree — a second checkout of one repository — an extra working directory sharing the same object database.
6. Skip — this one cannot be judged — `git bisect skip`, or exit code 125 from a `bisect run` script.

39.98 Common wrong ideas

1. Wrong: `git add` only marks a file for later. Right: it hashes the content and writes a real object into `.git/objects` immediately.
2. Wrong: the index stores differences. Right: it stores a complete snapshot, one entry per tracked file, with stat data and object IDs.
3. Wrong: `git commit` copies your files. Right: the blobs already exist; commit writes trees, one commit object, and moves a ref.
4. Wrong: a push failure means nothing was written. Right: a disconnect while reading the response says nothing about the write. Re-query with `git ls-remote`.
5. Wrong: `git status` shows the server's current state. Right: it compares against a cached remote-tracking ref that only a fetch or push updates.
6. Wrong: `git fetch` can mess up my work. Right: fetch changes only `refs/remotes/*` and the object database. `git pull` is the one to be careful with.
7. Wrong: rebase moves my commits. Right: it creates new commit objects with new identifiers, and leaves the old ones in the reflog.
8. Wrong: a clean merge means the code works. Right: git merges text, not meaning. Two non-overlapping changes can break the program completely.
9. Wrong: `git reset --hard` loses commits forever. Right: commits survive in the reflog for 90 days. Uncommitted edits, however, are gone.
10. Wrong: `--force-with-lease` is always safe. Right: it compares against your remote-tracking ref, which a background fetch can silently refresh.

39.99 Chapter summary in 20 lines

1. A tracked file exists in three places: working directory, index, object database. HEAD says which commit you are on.
2. `git status` is nothing more than two comparisons: index against HEAD, and working directory against index.
3. The index is a real binary file, `.git/index`, starting with the four bytes `DIRC`, present in git's first commit on 7 April 2005.
4. It holds one entry per tracked file: path, object ID, mode, and cached stat data from the filesystem.
5. That stat cache is why `git status` is fast: most files are never opened, only asked about.
6. `git add` hashes the content and writes the object at once, then records its name in the index. `git add -p` can stage a version that exists nowhere.
7. `git commit` writes trees from the index, writes one commit object, and moves one ref. Three steps, in that order.
8. The index exists so you can choose what goes in a commit. Other systems manage without one; it is a design choice, not a necessity.
9. `git restore`, `git reset --soft/--mixed/--hard`, `git revert`, `git clean` and `git stash` differ only in which of the three places they touch.
10. Only `git reset --hard` and `git clean` can destroy work that git never stored. Everything else is recoverable through the reflog.
11. A push sends one packfile of only the missing objects, framed in pkt-lines, with three sideband channels: data, progress, fatal error.
12. Twenty commits is one round trip, not twenty. In the real measurement, sixty objects in 3.95 KiB.
13. `send-pack: unexpected disconnect while reading sideband packet` means the pack was sent and the reply never arrived.
14. That error is identical whether the server applied the write or not. Both cases were reproduced. Only `git ls-remote` can tell you which happened.
15. In the reader's session both pushes had not applied, and that was established by re-querying, not by

reading the error.

16. `git fetch` changes only remote-tracking refs and the object store. `git pull` adds an integration step that touches your files.
17. Merge finds the common ancestor and does a three-way merge; fast-forward needs no commit because the old tip is already an ancestor.
18. Conflicts are git refusing to guess about overlapping text. Semantic conflicts merge cleanly and still break the code.
19. Rebase replays each commit onto a new parent, so every commit gets a new identifier even though the patch identifier is unchanged.
20. Rebase your own unpushed work freely; never rewrite history other people already hold, and prefer `--force-with-lease` when you must force.

40

CHAPTER

KB

Your Own Git Server, and What GitHub Adds

40.0 What this chapter gives you

1. You will be able to say, in one sentence, what a git server is, and prove it by building one in an empty folder in under a second.
2. You will be able to explain what `--bare` means, what a bare repository contains, what it deliberately lacks, and why pushing into a normal repository's checked-out branch is refused by default.
3. You will be able to run the whole loop yourself: create a server repository, push to it, clone it somewhere else, commit there, push back, and pull the change into the first copy, with real output at every step.
4. You will be able to do the same thing over SSH and see that a remote URL is nothing but a path plus a way of getting to it.
5. You will be able to read and write a refspec, and say exactly what `+refs/heads/*:refs/remotes/origin/*` does, character by character.
6. You will be able to write a server-side hook that deploys a website on every push, and say why a client-side hook is a helper and never a rule.
7. You will be able to state precisely what GitHub is: hosting for bare repositories, plus a web interface, plus identity and permissions, plus automation, founded in 2008 and bought by Microsoft in 2018 for 7.5 billion United States dollars.
8. You will be able to sort every feature you use into "this is git" or "this is GitHub", and defend each answer.
9. You will be able to explain the reader's own outage exactly: why git kept working while the GitHub API did not, using a three-column table of operations.
10. You will be able to choose between GitHub, a self-hosted Forgejo or GitLab, and a plain SSH remote with no web interface at all, and say what you gain and what you take on.
11. You will be able to mirror, migrate and back up a repository properly, and say what a backup of git does not cover.

40.1 A git server is just another repository

40.1.1 PLAIN – in simple words

1. Here is the whole idea of this chapter in one line.
2. A git "server" is just another git repository that other machines can reach.
3. That is all. There is no server program that must be bought. There is no special file format. There is no central authority.
4. Every copy of a git repository holds the full history. Every copy can send history to another copy and receive history from another copy.
5. So a "server" is only a copy that everybody agrees to treat as the meeting point.
6. We call it the server because of where it sits and who can reach it, not because of what it is made of.

7. The one thing a meeting-point copy usually does differently is that it has no working files. Only the history. We will see why in the next section.
8. Everything else in this chapter is proof of that single claim.

40.1.2 PLAIN - a picture in your head

1. Think of a study group of four students who each keep a notebook of the same lecture course.
2. Each notebook is complete. Any student can copy pages from any other student.
3. The group decides to leave one spare notebook in the department office, because the office is open to all four and nobody has to chase anybody.
4. The office notebook is not magic. It is the same kind of notebook. It has the same pages. It is special only because everyone can walk to it.
5. If the office burns down, the course is not lost. Four complete notebooks still exist in four bags.
6. That is exactly the relationship between your laptop's repository and the one you call the server.

Where this comparison breaks:

7. Copying pages by hand is slow and error-prone. Git copies exactly, and it copies only the pages the other side is missing.
8. Two students writing on the same page at the same time makes a mess. Git refuses that case rather than making a mess: it rejects the push and makes you combine the work first.
9. And a notebook has one order of pages. Git history is a graph, so two students can genuinely have two different valid orders at once.

40.1.3 PLAIN - a worked example

1. We will use three folders on one machine for the whole chapter. One machine is enough to prove the point, because git does not know or care whether the other repository is across the room or across the world.
2. The layout is this.

```
/tmp/lab/proj          your working copy, folder number one
/tmp/lab/srv/proj.git  the "server", a repository with no files
/tmp/lab/copy          a second working copy, folder number three
```

3. Here is a real repository being created and given two commits. Every block in this chapter is real output from a real run, not an illustration.

```
$ git init proj
Initialized empty Git repository in /tmp/lab/proj/.git/

$ git add README.md && git commit -m "First commit: add README"
[main (root-commit) 041c4ad] First commit: add README
 1 file changed, 1 insertion(+)
 create mode 100644 README.md

$ git add app.py && git commit -q -m 'Add app.py' && git log --oneline
08391eb Add app.py
041c4ad First commit: add README
```

4. Two commits exist. Nothing has touched a network. Nothing has touched a server, because there is not one yet.
5. In the next section we will make the server, and it will take one command and about ten milliseconds.

40.1.4 PLAIN - what is really happening inside

1. When two git repositories talk, one of them runs a small program that reads history out, and the other runs a small program that writes history in.
2. The reading program is called `git-upload-pack`. It says "here are all the names I have and what they point to", and then sends whatever objects the other side asks for.
3. The writing program is called `git-receive-pack`. It accepts a bundle of objects and a list of name changes, and applies them.
4. When you type `git fetch`, your git starts `git-upload-pack` on the other side and reads from it.
5. When you type `git push`, your git starts `git-receive-pack` on the other side and writes to it.
6. That is the entire client and server story. Two small programs and a way to connect their inputs and outputs.
7. If the other repository is a folder on the same disk, git simply runs the program directly. No network at all.
8. If it is on another machine, git needs some way to run that program over there and pipe bytes back and forth. That is what SSH does. That is what HTTPS does, with a small wrapper. That is what the git protocol does.
9. So the phrase "git server" really means "a machine willing to run `git-upload-pack` and `git-receive-pack` on a repository for you".

40.1.5 TECHNICAL - the engineer's version

1. Git's wire protocol is a framed byte stream called **pkt-line**. Each frame begins with four hexadecimal digits giving the frame's total length, including those four digits. The value `0000` is a flush packet and marks the end of a section.
2. We can see the raw advertisement by running the server program by hand over SSH. This is the real output, trimmed on the right to fit the page.

```
$ ssh gitbox 'git upload-pack --advertise-refs \
    /tmp/lab/srv/proj.git' | cat -A | head -3
010b64952410dab49907fec630138e55e7504834b082 HEAD^@multi_ack
 thin-pack side-band side-band-64k ofs-delta shallow ...
 symref=HEAD:refs/heads/main object-format=sha1
 agent=git/2.43.0$
003d64952410dab49907fec630138e55e7504834b082 refs/heads/main$
0000
```

3. Read the first line. `010b` is hexadecimal for 267, the length of that frame. Then a 40-character SHA-1, a space, the ref name `HEAD`, a NUL byte (shown by `cat -A` as `^@`), and then the capability list.
4. The second frame is `003d`, hexadecimal for 61: four length digits, 40 hash characters, one space, `refs/heads/main` at 15 characters, and one newline. That is $4 + 40 + 1 + 15 + 1 = 61$. The framing is exact.
5. `0000` ends the advertisement.
6. Protocol versions: version 0 is the original, version 1 adds a version string, and **protocol v2** was introduced in Git 2.18 (June 2018) and made the default for fetch in Git 2.26 (March 2020). Version 2 lets the client ask for a filtered ref list instead of receiving every ref up front, which matters on repositories with tens of thousands of refs.
7. The four transports and how the server program is started:

Transport	How the program starts	Default port
local path	run directly by your git	none
ssh	remote shell command	22
https	git-http-backend via CGI	443

Transport	How the program starts	Default port
git	git-daemon	9418

8. Port 9418 is registered with IANA for the git protocol. The git protocol has no authentication and no encryption, which is why GitHub permanently disabled unencrypted `git://` access on 15 March 2022.
9. Commands that observe this layer: `git ls-remote`, `GIT_TRACE=1 git fetch`, `GIT_TRACE_PACKET=1 git fetch`, and `git daemon --verbose`.

40.1.6 WORDS - remember these

1. Repository — a project's full history plus its settings — a directory containing an object database, a ref namespace and a configuration file.
2. Server — the copy everybody can reach — any repository exposed through a transport that can start upload-pack or receive-pack.
3. upload-pack — the program that sends history out — the server side of fetch and clone, which advertises refs and streams a packfile.
4. receive-pack — the program that takes history in — the server side of push, which validates ref updates and runs the receive hooks.
5. pkt-line — git's way of cutting a stream into labelled chunks — a framing format with a four-hex-digit length prefix and `0000` as flush.
6. Transport — the road the bytes travel on — the mechanism (file, ssh, http, git) that connects the two programs' standard input and output.

40.2 Bare repositories

40.2.1 PLAIN - in simple words

1. A normal repository has two parts: the files you edit, and a hidden folder called `.git` that holds the history.
2. A **bare repository** is the second part on its own. The history, with no files to edit.
3. You make one with `git init --bare`. By convention its folder name ends in `.git`, for example `proj.git`.
4. A bare repository is what you use for a server, because a server has nobody sitting at it editing files.
5. It is missing two things on purpose: a working tree (the actual editable files) and an index (git's staging list of what will go into the next commit).
6. Those two things only make sense when a human is working. A server has no human working in it.
7. You cannot run `git status` usefully in a bare repository, because there is no working tree to compare against.
8. You can run `git log`, `git show`, `git cat-file` and everything else that only reads history, because all of the history is right there.

40.2.2 PLAIN - a picture in your head

1. Think of a normal repository as a desk. On the desk are the pages you are writing. In a drawer under the desk is the complete archive of every version ever saved.
2. A bare repository is the drawer, on its own, in a storeroom. No desk.
3. You can pull any past version out of the drawer. You just cannot scribble on anything, because there is no desk surface.
4. When somebody pushes, they are handing new folders into the drawer and updating the labels on the front.

Where this comparison breaks:

5. A drawer takes more space as you add papers. Git's drawer stores each distinct version once and stores similar versions as small differences, so it grows far more slowly than the total of all versions.
6. And a real drawer does not check what you put in it. Git verifies every object against its hash on the way in and on the way out.

40.2.3 PLAIN – a worked example

1. Here are the two kinds side by side, from a real run.

```
$ git init --bare srv/proj.git
Initialized empty Git repository in /tmp/lab/srv/proj.git/

$ ls -1 /tmp/lab/proj/.git      $ ls -1 /tmp/lab/srv/proj.git
COMMIT_EDITMSG                  HEAD
HEAD                            branches
branches                        config
config                          description
description                     hooks
hooks                           info
index                           objects
info                            refs
logs
objects
refs
```

2. Look at what matches. `HEAD`, `branches`, `config`, `description`, `hooks`, `info`, `objects`, `refs` are in both. That is the whole history machinery.
3. Look at what only the normal repository has: `index`, `logs` and `COMMIT_EDITMSG`. Those are all about a person doing work here.
4. And the normal repository has one more thing that the bare one does not: the actual files, one level up.

```
$ ls -1 /tmp/lab/proj
README.md
app.py
```

5. Git records the difference in one configuration setting.

```
$ git -C /tmp/lab/proj config --get core.bare
false

$ git -C /tmp/lab/srv/proj.git config --get core.bare
true

$ cat /tmp/lab/srv/proj.git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = true
```

6. That is the entire difference at the configuration level. One boolean.

40.2.4 PLAIN – what is really happening inside

1. Now the important part: why you cannot push into a normal repository's checked-out branch.
2. A checked-out branch means three things are supposed to agree: the branch pointer, the index, and the files on disk.
3. If somebody pushes new commits into that branch from outside, git would move the branch pointer but would not touch the index or the files.

4. The person sitting at that repository would then be told they have deleted everything that arrived, because their files no longer match their branch.
5. So git refuses. Here is the real refusal, with the long lines wrapped to fit this page and nothing else changed.

```
$ git push wt main
remote: error: refusing to update checked out branch:
  refs/heads/main
remote: error: By default, updating the current branch in a
  non-bare repository is denied, because it will make the
  index and work tree inconsistent with what you pushed, and
  will require 'git reset --hard' to match the work tree to
  HEAD.
remote:
remote: You can set the 'receive.denyCurrentBranch'
  configuration variable to 'ignore' or 'warn' in the remote
  repository to allow pushing into its current branch;
  however, this is not recommended unless you arranged to
  update its work tree to match what you pushed in some
  other way.
To /tmp/lab/wt
 ! [remote rejected] main -> main (branch is currently
   checked out)
error: failed to push some refs to '/tmp/lab/wt'
```

6. Read the message closely. It does not say "you may not do this". It says "this would make the index and work tree inconsistent".
7. A bare repository has no index and no work tree, so there is nothing to make inconsistent, so the refusal never applies.
8. That is the real reason servers use bare repositories. Not policy. Arithmetic.

40.2.5 TECHNICAL - the engineer's version

1. `git init --bare` sets `core.bare = true` and lays the repository contents directly in the given directory rather than in a `.git` subdirectory.
2. Contents of a bare repository, and what each part is:

Entry	What it holds
HEAD	symbolic ref, usually refs/heads/main
objects/	the object database
refs/	loose refs, heads and tags
packed-refs	refs collapsed into one file
config	repository configuration
hooks/	executable hook scripts
info/	exclude file, alternates, http info

3. Absent from a bare repository: `index` (the staging area, a binary file in the format documented as the git index format, version 2, 3 or 4), the working tree, `logs/` unless `core.logAllRefUpdates` is turned on, and `COMMIT_EDITMSG`.
4. The refusal above comes from `receive.denyCurrentBranch`, whose default is `refuse`. Related settings are `receive.denyDeleteCurrent`, default `refuse`, and `receive.denyNonFastForwards`, default false.
5. Three ways to make a bare repository, all equivalent in result:

```
git init --bare proj.git
git clone --bare /path/to/proj proj.git
git clone --mirror /path/to/proj proj.git
```

6. They differ in what refs come along. `--bare` from `init` starts empty. `clone --bare` copies branches and tags but sets no fetch refspec. `clone --mirror` copies every ref under `refs/*` and configures the repository to keep tracking them. We measure that difference in 40.10.
7. There is a fourth arrangement worth knowing: a non-bare repository with `receive.denyCurrentBranch = updateInstead`. Then a push into the checked out branch also updates the working tree, provided the working tree is clean. This is useful for a small deploy target and is handled by the `push-to-checkout` hook.
8. Observation commands: `git rev-parse --is-bare-repository`, `git rev-parse --git-dir`, `git count-objects -vH`.

40.2.6 WORDS - remember these

1. Bare repository — the history with no editable files — a repository with `core.bare = true`, containing the object database and refs at top level, with no working tree and no index.
2. Working tree — the files you actually edit — the checked-out state of one commit, materialized on the filesystem.
3. Index — git's list of what goes in the next commit — a binary file mapping paths to blob hashes and stat data, also used during merges.
4. Checked-out branch — the branch you are currently on — the branch named by the symbolic ref `HEAD`.
5. denyCurrentBranch — the rule that stops pushes into a busy branch — the `receive.denyCurrentBranch` configuration key, default `refuse`.

40.3 Doing it end to end

40.3.1 PLAIN - in simple words

1. We will now do the whole loop, with nothing hidden.
2. Make a server repository. Tell your project about it. Send your history.
3. Copy the server repository into a fresh folder as if you were a second person on a second machine.
4. Make a commit in that fresh folder. Send it to the server.
5. Go back to the first folder. Bring the change down. Now both folders and the server agree.
6. Then we do the same thing again over SSH, to show that the only thing that changed is the address.
7. Here is the shape of the whole thing.

```
folder one              server              folder three
/tmp/lab/proj      /tmp/lab/srv/proj.git   /tmp/lab/copy
     |                     |                     |
     |---- push main ---->|                     |
     |                     |<---- clone ---------|
     |                     |                     |
     |                     |<--- push main ------|
     |<--- fetch/pull ----|                     |
```

40.3.2 PLAIN - a picture in your head

1. Think of a shared drop box in a corridor between two rooms.
2. You put a copy of your work in the box. The person in the other room takes a copy out.
3. They add a page and put their copy back in. You take the new page.
4. Nobody ever has to be in the corridor at the same time. The box holds the agreed state.
5. Git's box is smarter than a real one, because it never loses the older versions when a new one arrives. It stacks them all, in order, with each one pointing at the one before.

Where this comparison breaks:

6. A drop box has no opinion. Git's box refuses anything that would erase history it already holds, unless you force it explicitly.
7. And a drop box holds one thing at a time. Git's box holds many named lines of work at once, one name per branch.

40.3.3 PLAIN – a worked example

1. Step one. Point the project at the server and push.

```
$ git remote add origin /tmp/lab/srv/proj.git

$ git remote -v
origin  /tmp/lab/srv/proj.git (fetch)
origin  /tmp/lab/srv/proj.git (push)

$ git push -u origin main
To /tmp/lab/srv/proj.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

2. The server now really has the history. Look inside it directly.

```
$ ls -1 /tmp/lab/srv/proj.git/refs/heads
main

$ cat /tmp/lab/srv/proj.git/refs/heads/main
fa776d1882d36bdc0af68926f71ff710cdea281a

$ git -C /tmp/lab/srv/proj.git log --oneline
fa776d1 Change app.py
08391eb Add app.py
041c4ad First commit: add README
```

3. A branch on the server is one file containing one 40-character hash. That is all a branch has ever been.
4. Step two. Clone it into a third folder, as a second person would.

```
$ git clone /tmp/lab/srv/proj.git /tmp/lab/copy
Cloning into '/tmp/lab/copy'...
done.

$ git -C /tmp/lab/copy log --oneline
fa776d1 Change app.py
08391eb Add app.py
041c4ad First commit: add README

$ git -C /tmp/lab/copy branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
```

5. Step three. Commit in the copy and push it back. This is the terminal view, where the percentage counters overwrite themselves in place.

```
$ git add NOTES.md && git commit -q -m 'Add NOTES.md'

$ git push --progress origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
```

```
Delta compression using up to 2 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 328 bytes | 328.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To /tmp/lab/srv/proj.git
   fa776d1..249b3ed  main -> main
```

6. Three objects and 328 bytes went across: one commit, one tree, one blob. Nothing else was sent, because the server already had everything else.
7. Step four. Back in folder one, fetch first so you can see the two halves of a pull separately.

```
$ git status -sb
## main...origin/main

$ git fetch origin
From /tmp/lab/srv/proj
   fa776d1..249b3ed  main       -> origin/main

$ git status -sb
## main...origin/main [behind 1]

$ ls
README.md
app.py
```

8. Read that carefully. After `fetch`, git knows about the new commit and says you are behind by one. But `ls` shows no new file. Fetch changed nothing in your working tree. It only downloaded history.
9. Now pull, which is fetch plus merge.

```
$ git pull
Updating fa776d1..249b3ed
Fast-forward
 NOTES.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 NOTES.md

$ ls
NOTES.md
README.md
app.py
```

10. The loop is closed. Three repositories, one history, no service, no account, no web page.

40.3.4 PLAIN – what is really happening inside

1. Now the same thing over SSH, to prove the point of this chapter.
2. A remote URL has two parts: how to get there, and where it is once you arrive. Change the first part and nothing else changes.

```
$ git remote add ssh-origin \
    ssh://root@127.0.0.1:2222/tmp/lab/srv/proj.git

$ git ls-remote ssh-origin
249b3edc818ac4a38fc39b982e19a880675326de    HEAD
249b3edc818ac4a38fc39b982e19a880675326de    refs/heads/main
```

3. Exactly the same repository, reached a different way. Now push to it.

```
$ git commit -q -am 'Document how to run app'
```

```
$ git push ssh-origin main
To ssh://127.0.0.1:2222/tmp/lab/srv/proj.git
   249b3ed..6495241  main -> main

$ git -C /tmp/lab/srv/proj.git log --oneline -1
6495241 Document how to run app
```

4. And clone over SSH using the short form, which needs an entry in your SSH configuration file because the short form cannot carry a port number.

```
$ cat /root/.ssh/config
Host gitbox
  HostName 127.0.0.1
  Port 2222
  User root
  IdentityFile /root/.ssh/id_ed25519

$ git clone gitbox:/tmp/lab/srv/proj.git /tmp/lab/viassh
Cloning into '/tmp/lab/viassh'...

$ git -C /tmp/lab/viassh log --oneline -2
6495241 Document how to run app
249b3ed Add NOTES.md
```

5. What SSH actually did was log in and run one command. Here is git's own verbose trace of that decision, reproduced by running the command by hand.

```
$ ssh gitbox 'git upload-pack --advertise-refs \
    /tmp/lab/srv/proj.git' | head -c 60
010b64952410dab49907fec630138e55e7504834b082 HEAD
```

6. So SSH is not doing anything clever. It is a remote login that runs `git-upload-pack` or `git-receive-pack` and connects the pipes.
7. That is why an SSH remote needs no git-specific server software at all. If the account can log in and git is installed, you have a git server.

40.3.5 TECHNICAL – the engineer's version

1. The git protocol also works, which shows the third transport. Here is a real `git daemon` serving the same bare repository.

```
$ git daemon --reuseaddr --base-path=/tmp/lab/srv \
    --export-all --verbose

$ git ls-remote git://127.0.0.1/proj.git
a2648f3588da557802930bc984745c8a574b7a38    HEAD
8a65572b3a58cded170998be063e26b8cbea76e6    refs/heads/demo
0327736b4327edde443c3dfa9456555015d0b3f4    refs/heads/feature/login
a2648f3588da557802930bc984745c8a574b7a38    refs/heads/main
c541c6934d9bd45d02e86d902253422366134e87    refs/notes/commits
9ee3b0bda37deeffc6317b5dcda3c483895f490c    refs/tags/v1.0
```

2. The daemon log confirms protocol v2 and which program was requested.

```
[17254] Ready to rumble
[17260] Connection from 127.0.0.1:42896
[17260] Extended attribute "host": 127.0.0.1
[17260] Extended attribute "protocol": version=2
[17260] Request upload-pack for '/proj.git'
[17254] [17260] Disconnected
```

3. And the git protocol is read-only unless you go out of your way, because it has no authentication:

```
$ git push origin main
fatal: remote error: access denied or repository not exported:
  /proj.git
```

4. Timings and sizes from this run, on a small repository:

Operation	Objects	Bytes on wire
push of one commit	3	328
bare repo, loose	41	164.00 KiB
bundle of all refs	all	3.1 KiB

5. Server-side authorization for SSH remotes is normally done by the account, not by git. The standard pattern on a self-hosted box is a single `git` user whose `~/.ssh/authorized_keys` has one line per developer key, each line prefixed with `command="git-shell -c \"$SSH_ORIGINAL_COMMAND\""` plus `no-port-forwarding,no-agent-forwarding,no-pty`.
6. `git-shell` is a restricted login shell shipped with git that only permits `git-upload-pack`, `git-upload-archive` and `git-receive-pack`. It has existed since the early git releases and is the simplest real access control you can deploy.
7. Useful environment variables for debugging transports: `GIT_TRACE=1`, `GIT_TRACE_PACKET=1`, `GIT_SSH_COMMAND`, `GIT_CURL_VERBOSE=1`.

40.3.6 WORDS - remember these

1. Remote — a saved nickname for another repository — a named set of configuration keys holding one or more URLs and one or more refspecs.
2. origin — the usual nickname for where you cloned from — a convention, set automatically by `git clone`, changeable with `git clone -o name`.
3. Fetch — download history without touching your files — updating remote-tracking refs and the object database only.
4. Pull — fetch and then combine — `git fetch` followed by `git merge` or `git rebase`, depending on `pull.rebase`.
5. Fast-forward — moving a branch pointer along a straight line — an update where the old commit is an ancestor of the new one, so no merge is needed.
6. git-shell — a login shell that only allows git — a restricted shell that rejects every command except the three git transport programs.

40.4 Remotes in depth

40.4.1 PLAIN - in simple words

1. A remote is a saved nickname for another repository.
2. Instead of typing a long address every time, you say `origin` and git looks up the address.
3. You can have as many remotes as you like. They are just entries in a text file inside your repository.
4. The address can take four shapes: a plain folder path, an SSH address, an HTTPS address, or a `git://` address.
5. You can add a remote, rename it, change its address and delete it, and none of that touches your history at all. It is only a phone book entry.
6. A remote also carries a rule that says which of the other side's names get copied into which of your names. That rule is called a **refspec**.
7. The refspec is the part that most people never read, and it is the part that explains almost every confusing thing about push and fetch.

40.4.2 PLAIN - a picture in your head

1. Think of the contacts list in a phone.
2. `origin` is a contact name. Behind it sits a number.
3. Changing the number does not change any of your past conversations. It only changes where the next call goes.
4. Deleting the contact does not delete the conversations either.
5. Now imagine each contact also has a filing rule attached: "anything this person sends me goes in the folder named after them". That is the refspec.
6. So messages from Anita land in a folder called Anita, and messages from Bala land in a folder called Bala, and they never overwrite each other even if both send a file with the same name.

Where this comparison breaks:

7. A phone contact has one number. A git remote can have one address for reading and several different addresses for writing.
8. And a filing rule in a phone is your private business. A refspec is negotiated with the other side, because the other side must agree that the names you ask for exist.

40.4.3 PLAIN - a worked example

1. Everything about a remote lives in the repository's configuration file. Here is the real file after adding two remotes.

```
$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = /tmp/lab/srv/proj.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
    remote = origin
    merge = refs/heads/main
[remote "ssh-origin"]
    url = ssh://root@127.0.0.1:2222/tmp/lab/srv/proj.git
    fetch = +refs/heads/*:refs/remotes/ssh-origin/*
```

2. Now the refspec, character by character. The value is:

```
+refs/heads/*:refs/remotes/origin/*
```

3. It splits at the colon into a source and a destination.

Part	Meaning
`+`	allow a non-fast-forward update
`refs/heads/*`	source: every branch over there
`:`	separator
`refs/remotes/origin/*`	destination: my copy of them

4. In plain words: "take every branch on the other side, and store it under a name of mine that begins with `refs/remotes/origin/`, and let it move backwards or sideways if it has to."
5. So the other side's `main` becomes your `refs/remotes/origin/main`, which you normally type as `origin/main`.

6. This is why `origin/main` is not a branch you can commit on. It is a read-only copy of a name that lives somewhere else.
7. Watch the destination folder appear on disk after fetching from two different remotes.

```
$ ls -R .git/refs
.git/refs:
heads
remotes
tags

.git/refs/heads:
main

.git/refs/remotes:
origin
ssh-origin

.git/refs/remotes/origin:
main

.git/refs/remotes/ssh-origin:
main
```

8. Now push with an explicit refspec, which lets you send a local branch to a differently named remote branch.

```
$ git push origin HEAD:refs/heads/wip-login
To /tmp/lab/srv/proj.git
 * [new branch]      HEAD -> wip-login

$ git ls-remote origin
64952410dab49907fec630138e55e7504834b082    HEAD
64952410dab49907fec630138e55e7504834b082    refs/heads/main
0327736b4327edde443c3dfa9456555015d0b3f4    refs/heads/wip-login
```

9. And an empty source means "delete the destination". That is the whole trick behind deleting a remote branch.

```
$ git push origin :refs/heads/wip-login
To /tmp/lab/srv/proj.git
 - [deleted]         wip-login

$ git ls-remote origin
64952410dab49907fec630138e55e7504834b082    HEAD
64952410dab49907fec630138e55e7504834b082    refs/heads/main
```

10. `git push origin --delete wip-login` is a friendlier spelling of exactly the same refspec.

40.4.4 PLAIN – what is really happening inside

1. The leading + deserves its own demonstration, because it is the difference between "I refuse to lose history" and "overwrite it".
2. We put a commit on a branch, copy it with a refspec that has no +, then rewrite the commit on the server, then fetch again.

```
$ git fetch -q origin \
    '+refs/heads/demo:refs/remotes/nf/demo'
$ git rev-parse --short refs/remotes/nf/demo
4f4b9de
```

3. Now the branch is rewritten on the server, with `git commit --amend` followed by a force push. The old commit `4f4b9de` becomes `8a65572`, and the two are not ancestors of each other.

```
$ git push --force origin demo
To /tmp/lab/srv/proj.git
 + 4f4b9de...8a65572 demo -> demo (forced update)
```

4. Fetch with a refspec that has no `+`, and git refuses. Fetch with `+`, and git obeys. The same command, run twice, differing by one character.

```
$ git fetch origin refs/heads/demo:refs/remotes/nf/demo
From /tmp/lab/srv/proj
 ! [rejected]        demo -> nf/demo  (non-fast-forward)
 + 4f4b9de...8a65572 demo -> origin/demo  (forced update)

$ git fetch origin '+refs/heads/demo:refs/remotes/nf/demo'
From /tmp/lab/srv/proj
 + 4f4b9de...8a65572 demo -> nf/demo  (forced update)

$ git rev-parse --short refs/remotes/nf/demo
8a65572
```

5. Notice something in the middle of the first result. `nf/demo` was rejected, and in the very same command `origin/demo` was force-updated.
6. That is because the default refspec for `origin` has the `+`, and the one we typed by hand did not. Two rules, one fetch, two different outcomes.
7. That single output block explains the whole meaning of the plus sign better than any paragraph could.

40.4.5 TECHNICAL – the engineer's version

1. URL forms, all four, with real examples from this chapter's run:

Form	Example
local path	/tmp/lab/srv/proj.git
ssh, full	ssh://root@host:2222/tmp/proj.git
ssh, short	gitbox:/tmp/lab/srv/proj.git
https	https://github.com/git/git
git	git://127.0.0.1/proj.git

2. The short SSH form, `user@host:path`, is called the scp-like syntax. It cannot express a port number. If you need a non-default port you must use the `ssh://` form or a `Host` block in `~/.ssh/config`. That is a real limitation, not a style preference.
3. In the `ssh://` form the path is absolute. In the scp-like form the path is relative to the login user's home directory unless it starts with a slash. This trips people up constantly.
4. Remote management commands:

```
git remote add <name> <url>
git remote rename <old> <new>
git remote remove <name>
git remote set-url <name> <newurl>
git remote set-url --push <name> <pushurl>
git remote set-url --add --push <name> <another>
git remote show <name>
git remote prune <name>
```

5. Different fetch and push URLs are genuinely useful. Here we set one remote that reads from one place and writes to two, so a single push updates both a primary and a backup.

```
$ git remote -v
both    /tmp/lab/srv/proj.git (fetch)
both    /tmp/lab/srv/proj.git (push)
both    /tmp/lab/srv/mirror.git (push)

$ git push both main
Everything up-to-date
To /tmp/lab/srv/mirror.git
 * [new branch]      main -> main
```

6. Common reasons to have several remotes:

Situation	Typical remotes
you forked a project	origin, upstream
migrating platforms	origin, newhost
offsite backup	origin, backup
internal plus public	origin, public

7. Refspec grammar, stated properly. A refspec is `[+]<src>[:<dst>]`. For fetch, `src` is a ref on the remote and `dst` is a ref in your repository. For push, `src` is a local ref or any expression that resolves to a commit, and `dst` is a ref on the remote. The `+` sets the force flag for that one mapping. An empty `src` on push means delete `dst`. An omitted `dst` on fetch means "download it but only record it in `FETCH_HEAD`".
8. A remote may carry several fetch refspecs. Adding one is how you fetch GitHub pull-request refs:

```
git config --add remote.origin.fetch \
  '+refs/pull/*/head:refs/remotes/origin/pr/*'
```

9. URL rewriting is a useful production trick. `insteadOf` rewrites a prefix for all commands:

```
$ git config --global \
    url.'ssh://root@127.0.0.1:2222/tmp/lab/srv/'.insteadOf 'kb:'

$ git ls-remote kb:proj.git | head -1
a2648f3588da557802930bc984745c8a574b7a38    HEAD
```

10. `pushInsteadOf` does the same but only for pushes, which is the standard way to clone over HTTPS in continuous integration while pushing over SSH from a developer machine using the same recorded URL.

40.4.6 WORDS - remember these

1. Refspec — the rule that maps their names onto your names — a string of the form `[+]<src>:<dst>` used by fetch and push.
2. Remote-tracking ref — your read-only copy of their branch — a ref under `refs/remotes/<remote>/`, updated only by fetch, push and remote prune.
3. Force flag — permission to lose history on one ref — the `+` prefix in a refspec, or `--force` on the command line.
4. Scp-like syntax — the short SSH address form — `user@host:path`, which cannot carry a port number.
5. insteadOf — a find-and-replace for remote addresses — configuration keys `url.<base>.insteadOf` and `url.<base>.pushInsteadOf`.
6. FETCH_HEAD — a scratch note of what the last fetch brought — a file listing the refs and hashes downloaded by the most recent fetch.

40.5 Hooks: server-side and client-side

40.5.1 PLAIN - in simple words

1. A **hook** is a small program that git runs automatically at a certain moment.
2. Hooks live in the `hooks` directory of a repository. They are ordinary executable files. Shell, Python, anything the machine can run.
3. Git ships them as `.sample` files, which are inert. Rename one to remove the `.sample` and it becomes live.
4. There are hooks that run on your own machine, and hooks that run on the server when a push arrives.
5. The three important server hooks are `pre-receive`, `update` and `post-receive`.
6. `pre-receive` runs once for the whole push, before anything is saved. If it exits with a non-zero status, the entire push is rejected.
7. `update` runs once per branch or tag being changed. It can reject one branch and let the others through.
8. `post-receive` runs once after everything is saved. It cannot reject anything. It is for doing things afterwards: deploying, notifying, triggering a build.
9. Client hooks such as `pre-commit` and `commit-msg` run on your own machine. They are helpful, and they are not enforcement, because anybody can skip them with one flag.

40.5.2 PLAIN - a picture in your head

1. Think of a building with a security desk at the entrance and a noticeboard inside.
2. `pre-receive` is the guard who checks the whole delivery at the door. If any item on the trolley is wrong, the whole trolley goes back.
3. `update` is a second guard who checks each box on the trolley separately. Some boxes may pass while others are turned away.
4. `post-receive` is the person who, once the delivery is inside, updates the noticeboard and phones the department that ordered it. That person cannot send the delivery back. It is already in.
5. Client hooks are the checklist taped to your own desk before you leave the house. Useful. Easy to ignore.

Where this comparison breaks:

6. A guard can be persuaded. A hook is a program and does exactly what it says, every time, with no judgement.
7. And a real guard sees the whole delivery. A hook sees only three pieces of text per ref: the old hash, the new hash and the ref name. Everything else it must look up for itself.

40.5.3 PLAIN - a worked example

1. Here is a real, working `post-receive` hook that deploys a website. It is in the server repository at `hooks/post-receive` and is executable.

```
#!/bin/sh
# Deploy the site when main is updated.
DEPLOY_DIR=/tmp/lab/www
BRANCH=refs/heads/main

while read old new ref
do
  if [ "$ref" = "$BRANCH" ]; then
    echo "post-receive: deploying $ref"
    echo "post-receive: old=$(git rev-parse --short $old)"
    echo "post-receive: new=$(git rev-parse --short $new)"
    git --work-tree="$DEPLOY_DIR" --git-dir=. checkout -f main
    echo "post-receive: deployed to $DEPLOY_DIR"
  else
```

```
    echo "post-receive: ignoring $ref"
  fi
done
```

2. Line by line, because every line earns its place.
3. `#!/bin/sh` is the shebang. It tells the kernel which interpreter to run. Without it, git will try to execute the file directly and fail.
4. `DEPLOY_DIR` is where the website files should end up. In production this would be the directory your web server serves.
5. `BRANCH` is the only branch we deploy. Pushing a feature branch must not change the live site.
6. `while read old new ref` is the key line. Git feeds `post-receive` one line per updated ref on standard input, and each line has exactly three fields: the hash before, the hash after, and the full ref name.
7. The `if` compares the ref name to `refs/heads/main`. Note it is the full ref name, not the short name. A hook never sees `main` on its own.
8. `git rev-parse --short` turns a 40-character hash into a 7-character one for the log line. For a brand new branch the old hash is forty zeros.
9. The deploy line is the interesting one. `--work-tree` tells git to put the files somewhere other than the repository, and `--git-dir=.` says the history is right here. `checkout -f main` then writes every file of `main` into the deploy directory, overwriting whatever is there.
10. `echo` output from a server hook is sent back to the person pushing, prefixed with `remote:`. That is how you talk to your users.
11. Now the real push, with the hook live. The deploy directory was empty before this command.

```
$ git add index.html && git commit -q -m 'Add home page'

$ git push origin main
remote: post-receive: deploying refs/heads/main
remote: post-receive: old=6495241
remote: post-receive: new=adb4a16
remote: Already on 'main'
remote: post-receive: deployed to /tmp/lab/www
To /tmp/lab/srv/proj.git
   6495241..adb4a16  main -> main

$ ls -la /tmp/lab/www
-rw-r--r-- 1 root root   21 Aug 13 03:12 NOTES.md
-rw-r--r-- 1 root root   47 Aug 13 03:12 README.md
-rw-r--r-- 1 root root   18 Aug 13 03:12 app.py
-rw-r--r-- 1 root root   17 Aug 13 03:12 index.html

$ cat /tmp/lab/www/index.html
<h1>KedByte</h1>
```

12. That is a complete deployment system in fourteen lines of shell. No service, no agent, no vendor.

40.5.4 PLAIN - what is really happening inside

1. Now the two hooks that can say no. First `pre-receive`, enforcing a branch naming policy.

```
#!/bin/sh
fail=0
while read old new ref
do
  case "$ref" in
    refs/heads/main)      ok=yes ;;
    refs/heads/feature/*) ok=yes ;;
    refs/tags/*)          ok=yes ;;
```

```
    refs/notes/*)         ok=yes ;;
    *)                    ok=no  ;;
  esac
  if [ "$ok" = "no" ]; then
    echo "policy: bad branch name: $ref"
    fail=1
  fi
done
if [ "$fail" = "1" ]; then
  echo "policy: whole push rejected"
  exit 1
fi
exit 0
```

2. Watch it reject a whole push because one of two branches broke the rule. Both branches had real new commits. Both were refused.

```
$ git push origin main tmp/scratch
remote: policy: bad branch name: refs/heads/tmp/scratch
remote: policy: whole push rejected
To /tmp/lab/srv/proj.git
 ! [remote rejected] main -> main (pre-receive hook declined)
 ! [remote rejected] tmp/scratch -> tmp/scratch (pre-receive
   hook declined)
error: failed to push some refs to '/tmp/lab/srv/proj.git'

$ git ls-remote origin refs/heads/main
adb4a16ce58824aa2079ee092be20959ecf7cb92    refs/heads/main
```

3. The server's `main` did not move. That is the all-or-nothing behaviour of `pre-receive`, and it is why it is the right place for policy that must never be half-applied.
4. Now `update`, which is per-ref. This one makes `main` fast-forward only.

```
#!/bin/sh
ref="$1"; old="$2"; new="$3"
zero=0000000000000000000000000000000000000000

if [ "$ref" = "refs/heads/main" ]; then
  if [ "$new" = "$zero" ]; then
    echo "policy: main may not be deleted"
    exit 1
  fi
  if [ "$old" != "$zero" ]; then
    base=$(git merge-base "$old" "$new")
    if [ "$base" != "$old" ]; then
      echo "policy: main is fast-forward only"
      exit 1
    fi
  fi
fi
exit 0
```

5. `update` receives its three values as command-line arguments, not on standard input. That is a genuine difference from `pre-receive` and it is easy to get wrong.
6. The forty zeros mean "nothing". A new hash of all zeros means the ref is being deleted. An old hash of all zeros means the ref is being created.
7. `git merge-base "$old" "$new"` finds the youngest common ancestor. If that ancestor is the old commit itself, then the new commit is a descendant and the update is a fast-forward. If not, history is being

rewritten.

8. Here is a real force push being stopped by that rule.

```
$ git commit -q --amend -m 'Add about page (message fixed)'

$ git push --force origin main
remote: policy: main is fast-forward only
remote: error: hook declined to update refs/heads/main
To /tmp/lab/srv/proj.git
 ! [remote rejected] main -> main (hook declined)
error: failed to push some refs to
  '/tmp/lab/srv/proj.git'

$ git ls-remote origin refs/heads/main
a07dcbed1fbb04efa6dbb8f6272a2aed8c361556    refs/heads/main
```

9. That is branch protection, built by hand, in twenty lines. GitHub's branch protection is the same idea with a web form in front of it.
10. One honest note from the same run. When we tried `git push origin :main` to delete the branch, git's own built-in `receive.denyDeleteCurrent` check fired before our hook did, because `main` is the branch that the bare repository's HEAD points at. Our hook line is still worth having, for every branch that is not the current one.

40.5.5 TECHNICAL - the engineer's version

1. Server-side hooks, in the exact order git runs them on a push:

Hook	Runs	Can reject	Input
pre-receive	once per push	yes, all refs	stdin
update	once per ref	yes, that ref	argv
post-receive	once per push	no	stdin
post-update	once per push	no	argv

2. `post-update` is the oldest of these and its main modern use is running `git update-server-info`, which is only needed for the "dumb" HTTP transport, where a plain web server serves the repository files with no git-aware program behind it.
3. Client-side hooks that matter day to day:

Hook	When	Typical use
pre-commit	before message editor	linting, secret scanning
prepare-commit-msg	before editor opens	insert a template
commit-msg	after message written	enforce message format
pre-push	before push starts	run tests
pre-rebase	before a rebase	protect published branches

4. Here are two real client hooks and the real refusals they produce. First `pre-commit`, refusing a marker string:

```
#!/bin/sh
if git diff --cached -U0 | grep -q 'DO-NOT-COMMIT'; then
  echo "pre-commit: found DO-NOT-COMMIT marker"
  exit 1
fi
exit 0
```

```
$ git add secrets.py && git commit -m 'KB-42: add config'
pre-commit: found DO-NOT-COMMIT marker
```

5. Then `commit-msg`, enforcing a ticket prefix. Git passes the path of the file holding the message as the first argument.

```
#!/bin/sh
first=$(head -n 1 "$1")
if ! echo "$first" | grep -Eq '^KB-[0-9]+: .+'; then
  echo "commit-msg: need a KB-<number>: prefix"
  echo "commit-msg: you wrote: $first"
  exit 1
fi
exit 0
```

```
$ git commit -m 'add contact page'
commit-msg: need a KB-<number>: prefix
commit-msg: you wrote: add contact page

$ git commit -m 'KB-43: add contact page'
[main 40048a8] KB-43: add contact page
 1 file changed, 1 insertion(+)
```

6. And now the honest part, demonstrated rather than asserted. One flag skips both hooks, and the resulting commit pushes perfectly happily.

```
$ git commit --no-verify -m 'whatever, no ticket, no checks'
[main a2648f3] whatever, no ticket, no checks
 1 file changed, 1 insertion(+)
 create mode 100644 leak.py

$ git push -q origin main && git ls-remote origin main
a2648f3588da557802930bc984745c8a574b7a38    refs/heads/main
```

7. The secret marker went in. The ticket prefix was skipped. Both hooks were bypassed by a single documented flag.
8. **Client hooks are not enforcement.** They are a convenience that catches honest mistakes. Three independent reasons: `--no-verify` exists, hooks are not committed to the repository so a fresh clone has none, and a hook file that is not executable is silently ignored.
9. Because hooks are not versioned, teams use a manager that installs them from a committed configuration. The common ones are pre-commit (a Python tool, first released in 2014), Husky for Node.js projects, and setting `core.hooksPath` to a directory that is committed. `core.hooksPath` arrived in Git 2.9 (June 2016) and is the plainest option.
10. Real enforcement lives on the server: `pre-receive` and `update` hooks on a self-hosted server, or branch protection plus required status checks on a hosted platform. Everything else is advice.

40.5.6 WORDS - remember these

1. Hook — a program git runs at a set moment — an executable in the hooks directory or in `core.hooksPath`, invoked with a defined argument or stdin contract and judged by its exit status.
2. pre-receive — the all-or-nothing gate on a push — a server hook reading `old new ref` lines on stdin, whose non-zero exit rejects every ref.
3. update — the per-branch gate — a server hook called once per ref with three arguments, whose non-zero exit rejects that ref only.
4. post-receive — the after-the-fact action — a server hook that runs once the refs are updated and cannot reject anything.

5. no-verify — the flag that skips client hooks — `--no-verify` on commit and push, which is why client hooks are advice and not policy.
6. core.hooksPath — a way to share hooks with a team — a configuration key pointing at a directory of hooks, which can be committed to the repository.

40.6 What GitHub actually is

40.6.1 PLAIN - in simple words

1. GitHub is four things stacked on top of each other.
2. One: hosting for bare repositories. Machines that hold your repositories and run `git-upload-pack` and `git-receive-pack` for you.
3. Two: a web interface. Pages that let you read code, history and differences in a browser, and comment on them.
4. Three: an identity and permission system. Accounts, organizations, teams, tokens, and rules about who may do what to which repository.
5. Four: automation. Servers that run your tests and builds when something happens, and store the results.
6. GitHub was founded in 2008 by Tom Preston-Werner, Chris Wanstrath and P. J. Hyett. The site opened to the public in April 2008.
7. Microsoft announced it would buy GitHub on 4 June 2018 for 7.5 billion United States dollars in Microsoft stock. The purchase completed on 26 October 2018.
8. Here is the sentence that matters most in this chapter: the git part of GitHub is not the valuable part.
9. Anybody can host bare repositories. It took us one command and ten milliseconds. What is hard to copy is the other three layers, and above all the fact that millions of developers already have an account there.

40.6.2 PLAIN - a picture in your head

1. Think of a public library that also happens to have meeting rooms, a membership desk and a printing service.
2. Holding books is the easy part. Any building with shelves can hold books.
3. What makes the library useful is that you know where it is, everyone else uses it too, there is a catalogue, there is a librarian who checks membership cards, and there is a room you can book.
4. GitHub is that. The shelves are cheap. The catalogue, the membership system and the fact that everyone is already a member are the product.
5. That is also why leaving is harder than it looks. Your books come with you. Your reading room bookings, your notes in the margins and your library card do not.

Where this comparison breaks:

6. A library lends you one copy and keeps the original. Git gives you the whole collection every time, so the "original" has no special status.
7. And a library card is issued by the library. Your git history is not issued by GitHub. It is yours, complete, on your own disk.

40.6.3 PLAIN - a worked example

1. Here is proof that GitHub's git layer is ordinary git, taken live during the writing of this chapter.

```
$ git ls-remote https://github.com/git/git | head -5
745601a9a94110d74769ab605ccd4f61339758d2    HEAD
165e5ad3169d0fd26637da3383a4514f1a9d1e72    refs/heads/bisect
426ec014a6630fb78c5a7334461bc33568f07743    refs/heads/jch
e9019fcafe0040228b8631c30f97ae1adb61bcdc    refs/heads/maint
```

```
745601a9a94110d74769ab605ccd4f61339758d2    refs/heads/master

$ git ls-remote https://github.com/git/git | wc -l
5297
```

2. That is exactly the same command, producing exactly the same shape of output, as the one we ran against a folder in `/tmp`.
3. The git source repository on GitHub advertises 5,297 refs. Our little bare repository advertised seven. Same protocol, same program, different scale.
4. Now here is proof of the second layer, the part that is not git. GitHub stores pull requests as extra refs that plain git can see but that plain git never creates.

```
$ git ls-remote https://github.com/cli/cli 'refs/pull/*' \
    | head -4
e9a3253762e768badaa1d4a5b3d267416d1e42f4    refs/pull/1/head
9966ca8bdb06ff86febbb49ff08480ef3fb4f790    refs/pull/10/head
24b83202679eeddb72bf9ebeb35752bf3f5ea737    refs/pull/100/head
b4b8e8a13bb99bae479929048ea5f64808d0bc32    refs/pull/10002/head

$ git ls-remote https://github.com/cli/cli 'refs/pull/*' \
    | wc -l
5147
```

5. Five thousand one hundred and forty-seven pull-request refs. Git can list them, because they are refs. Git cannot create one, cannot comment on one, cannot approve one and does not know what "pull request" means.
6. That single distinction is the subject of the next two sections.

40.6.4 PLAIN – what is really happening inside

1. When you `git push` to GitHub, this is the sequence.
2. Your git resolves `github.com` to an address. In the reader's own session it resolved to `20.207.73.82`, which is in a Microsoft-owned range, because GitHub is a Microsoft company and fronts traffic through Microsoft's network edge.
3. Your git opens a connection, either TCP port 22 for SSH or TCP port 443 for HTTPS.
4. It authenticates. Over SSH that is your key pair. Over HTTPS that is a personal access token or a credential helper.
5. GitHub's front end checks who you are and whether you may write to that repository. This check is GitHub's, not git's.
6. If allowed, GitHub runs `git-receive-pack` against the bare repository it holds for you. From this point the bytes are ordinary git.
7. GitHub's own `pre-receive` hooks then run. Branch protection, required signatures, push rules and secret scanning all live here.
8. If nothing objects, the refs update and GitHub's equivalent of `post-receive` fires. That is what creates the timeline entry, sends the notification, updates the pull request, and starts an Actions workflow.
9. Steps 4, 5, 7 and 8 are GitHub. Steps 3 and 6 are git. That is the whole boundary.

40.6.5 TECHNICAL – the engineer's version

1. Dates and figures worth committing to memory:

Event	Date
Git first released by Linus Torvalds	April 2005
GitHub launched publicly	April 2008

Event	Date
GitHub introduces Git LFS	April 2015
Microsoft announces acquisition	4 June 2018
Acquisition completes	26 Oct 2018
Password auth for git removed	13 Aug 2021
Unencrypted git:// disabled	15 Mar 2022

2. GitHub removed password authentication for git operations over HTTPS on 13 August 2021. Since then HTTPS pushes require a personal access token, an OAuth token, a GitHub App installation token, or SSH keys instead.
3. On 24 March 2023 GitHub replaced its RSA SSH host key after the private key was briefly exposed in a public repository. Every user with the old key in `known_hosts` saw a host key verification failure. This is a good reminder that host key pinning is real security with real operational cost.
4. GitHub's git hosting uses a custom system for replication and routing. Publicly described pieces include Spokes, formerly called DGit, for three-way replication of repositories across servers, and a proxy layer in front of the git transports. The important engineering point is that nothing about this changes the wire protocol. Your git client cannot tell.
5. The scale of GitHub's non-git surface is what makes it expensive to run: the REST API, the GraphQL API, webhooks, Actions runners, package registries, Pages hosting and code search. None of those are git.
6. Estimated user counts published by GitHub have grown from roughly 100 million developers reported in early 2023 to figures above that since. Treat any single number as approximate and dated, because the company publishes it as a marketing figure rather than an audited one.

40.6.6 WORDS - remember these

1. Forge — a website that hosts repositories and adds collaboration — the general term covering GitHub, GitLab, Gitea, Forgejo and Bitbucket.
2. Hosting — somebody else runs the bare repository — a service that exposes upload-pack and receive-pack over authenticated transports.
3. Personal access token — a password made for programs — a revocable credential with a fixed set of scopes, used in place of a password.
4. Scope — the list of things a token is allowed to do — a named permission such as `repo`, `workflow` or `read:org` attached to a token.
5. refs/pull — GitHub's storage for pull request heads — read-only refs created by GitHub, visible to `git ls-remote`, not creatable by git.

40.7 What GitHub adds that plain git does not have

40.7.1 PLAIN - in simple words

1. Git has exactly one job: move history between repositories, exactly.
2. It has no idea what a pull request is. No idea what a review is. No idea what an issue, a label, a release, a package or a build is.
3. Every one of those is added by the website you happen to be using.
4. Here is the whole list, in plain words, one item per line.
5. Pull requests. A written request saying "please take my branch into yours", with a conversation attached to it.
6. Code review. Named people reading the change and marking it approved, or asking for changes, or just commenting.
7. Branch protection. A rule saying "nobody may push straight into main".
8. Required checks. A rule saying "the tests must pass before the merge button is allowed to work".

9. CODEOWNERS. A file listing which people must review which folders.
10. Issues and projects. A shared to-do list, and a board of cards.
11. Releases. A named point in history with files attached, such as a compiled program somebody can download.
12. Packages. A place to publish libraries that other projects install.
13. Actions. Computers that run your tests and builds when something happens, and keep the logs.
14. Two programming interfaces, REST and GraphQL. Ways for programs to ask GitHub questions and give it orders.
15. Permissions. Accounts, organizations, teams, roles and tokens.
16. Not one of those is part of git. Git ships none of them, and would work exactly the same if GitHub had never existed.

40.7.2 PLAIN - a picture in your head

1. Think of a shared workshop where people build furniture.
2. Git is one machine in the corner. It copies a design exactly, down to the last measurement, and hands the copy to somebody else.
3. That machine is superb at that job and does nothing else. It has no opinion about who you are or whether the design is any good.
4. The workshop around the machine is everything else. A pass reader on the door. A noticeboard of jobs to do. A sign-off sheet outside the paint room saying two people must approve the colour. An overnight test rig. A shelf of finished pieces with labels on them.
5. You can take the copying machine home and still build furniture.
6. You cannot take the noticeboard, the sign-off sheets, the pass reader or the test rig. They belong to the building.
7. GitHub is the building. Git is the machine.

Where this comparison breaks:

8. The building also stores the designs, so day to day the two feel like one thing. The split is in the protocol, not in the floor plan.
9. And the sign-off sheet is not decoration. On GitHub, branch protection really does stop the push at the door. It is a lock, not a note.

40.7.3 PLAIN - a worked example

1. A pull request has two halves. One half is refs, which git can see. The other half is a database row, which git cannot see.
2. Here is the visible half, read live with plain git, no account, no token.

```
$ git ls-remote https://github.com/cli/cli 'refs/pull/10002/*'
b4b8e8a13bb99bae479929048ea5f64808d0bc32 refs/pull/10002/head

$ git ls-remote https://github.com/cli/cli 'refs/pull/*/head' \
    | wc -l
4981

$ git ls-remote https://github.com/cli/cli 'refs/pull/*/merge' \
    | wc -l
166
```

3. Read those two numbers. 4,981 pull requests have a `head` ref, which is the commit the author is proposing.
4. Only 166 have a `merge` ref. That is GitHub's pre-computed trial merge, kept only while a pull request is open and can be merged cleanly.

5. Now the invisible half. Git cannot answer any of these, at any price: who opened it, what the title says, who approved it, which comment sits on which line, whether the tests passed, whether the merge button works.
6. Proof by absence. Ask git for every command it has that mentions requests or reviews.

```
$ git help -a | tr ' ' '\n' | grep -iE 'request|review|issue'
request-pull
```

7. One result, and it is not what you think. `git request-pull` has existed since 2005 and prints an email for you to send to a maintainer by hand.

```
$ git request-pull main /tmp/mg/base feature/login
The following changes since commit cd18e975:

  Base commit (2026-08-13 07:43:39 +0000)

are available in the Git repository at:

  /tmp/mg/base feature/login

for you to fetch changes up to 00ee7d7c:

  Fix login typo (2026-08-13 07:43:39 +0000)

Reader (2):
      Start login
      Fix login typo

 login.py | 2 ++
 1 file changed, 2 insertions(+)
```

8. That is the true ancestor of the pull request. It produces text for a mailing list. It has no state, no approval, no button and no thread.
9. GitHub's version added the state, the approval, the button and the thread. That addition is the product.

40.7.4 PLAIN - what is really happening inside

1. A pull request is three things glued together.
2. One: two ref names. A base, meaning where you want the work to land, and a head, meaning the work. Both are ordinary git refs.
3. Two: a row in a database, holding a number, a title, a body, a state (open, closed or merged) and an author.
4. Three: a conversation. Comments, review states, requested reviewers, labels, and links to issues.
5. When you press Merge, GitHub runs an ordinary git merge on its own server, into the base branch, using whichever strategy you picked.
6. Review states are values on that database row: `APPROVED`, `CHANGES_REQUESTED`, `COMMENTED`, `DISMISSED`, `PENDING`.
7. Branch protection is enforced by GitHub's own `pre-receive` hook. Earlier in this chapter we wrote a `pre-receive` hook by hand that rejected a push. GitHub runs a much larger version of the same thing.
8. A required status check is a record attached to a **commit SHA**, not to a branch. Something posts "the tests passed for this SHA" through the API.
9. The protection rule then asks one question: does the head SHA of this pull request carry a passing record with the required name?
10. That is exactly why, in the reader's own session, checking CI by run ID against the exact head SHA was stronger than trusting a green tick beside a branch name. The tick is drawn next to a name, and names move. The record is attached to a SHA, and a SHA never moves.

11. CODEOWNERS is the one hybrid. The file is a real blob inside your repository, so git stores and versions it. The rule "these owners must approve" is GitHub's, and git ignores the file completely.
12. Actions is a scheduler plus a fleet of machines. A push fires an internal event, a workflow file in the repository says what to run, and a runner picks up the job.
13. Releases and packages are file storage with a database index in front. The attached files are not git objects and are not in the history.
14. So the rule is short: refs and objects are git; anything with an opinion is GitHub.

40.7.5 TECHNICAL - the engineer's version

1. The classification. "Truth lives in" means where the authoritative copy is.

Feature	git or GitHub	Truth lives in
Commit, tree, blob, tag	git	object database
Branch and tag refs	git	refs, packed-refs
Merge, rebase, cherry-pick	git	local repository
Push, fetch, clone	git	wire protocol
Server-side hooks	git	hooks directory
Submodules	git	gitlink plus file
Pull request	GitHub	GitHub database
Review state	GitHub	GitHub database
Branch protection rule	GitHub	GitHub database
Required status check	GitHub	GitHub database
CODEOWNERS file	git	a blob in the tree
CODEOWNERS behaviour	GitHub	rule engine
Issue, label, milestone	GitHub	GitHub database
Projects board	GitHub	GitHub database
Release and its assets	GitHub	object storage
Packages registry	GitHub	object storage
Actions workflow run	GitHub	runners plus DB
REST and GraphQL API	GitHub	GitHub servers
Teams, roles, tokens	GitHub	GitHub database
Fork, star, watch	GitHub	GitHub database

2. Dates, so you can see that every GitHub feature has a birthday and git has none of them.

GitHub feature	Date
Pull Requests 2.0 post	31 Aug 2010
Releases introduced	2 Jul 2013
Protected branches, checks	3 Sep 2015
GraphQL API announced	14 Sep 2016
CODEOWNERS introduced	6 Jul 2017
Actions announced	16 Oct 2018
Actions generally available	13 Nov 2019
Merge queue GA	12 Jul 2023

3. Rate limits, from GitHub's published documentation, current as of 2026. REST, unauthenticated: 60 requests per hour per IP address. REST, with a personal access token: 5,000 per hour. GitHub Apps and OAuth apps owned by a GitHub Enterprise Cloud organization: 15,000 per hour. GraphQL is budgeted in points, 5,000 points per hour for a user token.

4. The API keeps its own books, and will tell you so:

```
$ curl -s https://api.github.com/rate_limit
{ "resources": {
    "core":    { "limit": 15000, "remaining": 15000 },
    "search":  { "limit": 30,    "remaining": 30 },
    "graphql": { "limit": 5000,  "remaining": 5000 } } }
```

5. Honest note on that output: the request above ran through a proxy that attached an application credential, which is why `core` shows 15,000 and not 60. The number you see depends on how you authenticated.
6. Git has no equivalent of any of that, because git has no accounts. There is nothing in the git protocol to rate-limit against.
7. The permission model, as of 2026. Repository roles: Read, Triage, Write, Maintain, Admin. Organization roles: Member, Moderator, Billing manager, Owner, plus custom roles on paid plans. Teams can nest, and effective permission is the union of every grant a user receives.
8. Token types. Classic personal access tokens carry coarse scopes such as `repo`, `workflow`, `read:org` and `admin:repo_hook`. Fine-grained personal access tokens, generally available since 2022, are limited to chosen repositories, carry per-resource read or write permissions, and must have an expiry date.
9. The reader's own session shows why the scope list is not decoration. One push was refused because the token lacked the `workflow` scope and the push modified a CI workflow file. Git had already packed and sent the objects. GitHub's server-side rule refused the ref update.
10. GraphQL exposes fields with no git equivalent at all. The reader saw `mergeStateStatus` values `CLEAN`, `BLOCKED` and `UNKNOWN`; the full set also includes `BEHIND`, `DIRTY`, `DRAFT`, `HAS_HOOKS` and `UNSTABLE`.
11. Git cannot compute any of those values, because each depends on a rule git does not know exists.
12. Scale figures published by GitHub are marketing figures, not audited ones. Roughly 100 million developers were reported in early 2023, about 150 million for May 2025, and the repository count was reported to pass one billion in June 2025. Treat all three as approximate and dated.

40.7.6 WORDS - remember these

1. Pull request — a request to merge one branch into another, with a discussion — a GitHub database object referencing a base ref and a head ref, exposed to git only as `refs/pull/N/head`.
2. Review state — the verdict a reviewer left — one of `APPROVED`, `CHANGES_REQUESTED`, `COMMENTED`, `DISMISSED`, `PENDING`.
3. Branch protection — a rule stopping direct pushes to a branch — a server-side policy implemented in GitHub's `pre-receive` path.
4. Required status check — the tests must pass before merging — a named check record that must exist and be successful for the head SHA.
5. Check run — one reported result for one commit — an API object attached to a SHA with a name, status, conclusion and output.
6. CODEOWNERS — a file saying who owns which folders — a versioned path pattern file whose enforcement is entirely the forge's, not git's.
7. Fork — your own server-side copy of somebody's repository — a GitHub database relationship plus a repository that may share object storage.
8. GraphQL API — one endpoint you send a shaped query to — GitHub's typed schema at a single endpoint, budgeted in points rather than requests.
9. mergeStateStatus — GitHub's summary of whether merging is allowed — a GraphQL enum with values such as `CLEAN`, `BLOCKED` and `BEHIND`.
10. Merge queue — a line that tests changes in order before merging — a GitHub service that builds candidate merges and merges only what passes.

40.8 The exact boundary, using the reader's own outage as evidence

40.8.1 PLAIN - in simple words

1. During the reader's outage one sentence describes everything: git kept working, and the GitHub API did not.
2. Committing worked. Branching worked. Merging worked. Rebasing worked. Reading old history worked. Tagging worked.
3. Pushing failed. Fetching failed. Looking at a pull request failed. Checking a CI run failed.
4. That is not luck and not a coincidence. It follows from two facts the book has already given you.
5. Fact one: a git command needs the network if and only if it must read from, or write to, another repository. Everything else is reading and writing files inside the `.git` folder, and files need no network.
6. Fact two: every GitHub-specific feature is a network service. There is no local copy of a pull request, an issue, a review or a build result.
7. Put the two together and the boundary is exact. The local half of git kept working. The remote half of git, and the whole of GitHub, did not.
8. Nothing here is special to GitHub. Point your remote at a self-hosted Forgejo and the same line falls in the same place.

40.8.2 PLAIN - a picture in your head

1. Imagine you keep a paper diary at home, and your office keeps a shared wall planner that everyone updates.
2. The phone line to the office is cut.
3. You can still write in your diary. Every page, any amount. The cut line does not touch your pen.
4. You can still read every old page, because they are in your hands.
5. You cannot see today's wall planner, and you cannot change it.
6. Here is the sharp part. You can still look at the photo of the wall planner you took last Tuesday. It will show you Tuesday's planner, confidently, and it will be out of date.
7. That photo is `origin/main`. During the outage the reader's machine reported `origin/main` as `0a95cc8`. That was a cached value from the last successful fetch, not a live fact about the server.

Where this comparison breaks:

8. A photo obviously looks like a photo. A remote-tracking ref does not. `git log origin/main` prints normally, with no warning and no note saying how old the information is.
9. And a wall planner suggests one true copy. Git has no true copy. During the outage `origin/main` was stale, but the reader's own `main` was not somehow less real.

40.8.3 PLAIN - a worked example

1. Here is the boundary as a table. Read the third column as the reason, not as an excuse.

Operation	Works offline	Why
git commit	yes	writes .git only
git branch, git switch	yes	writes a ref file
git merge, git rebase	yes	objects already local
git log, git diff, blame	yes	reads local objects
git tag, git stash	yes	local refs and objects
git bisect	yes	walks the local graph
git show origin/main	yes, but stale	cached tracking ref
git fetch, git pull	no	must read the remote
git push	no	must write remote refs

Operation	Works offline	Why
git clone	no	nothing local yet
git ls-remote	no	asks the other side
Open or merge a PR	no	GitHub database only
Read review comments	no	GitHub database only
Check a CI run result	no	GitHub service only
Create or close an issue	no	GitHub database only
Get a release asset	no	GitHub storage only
Branch protection verdict	no	server-side rule
Any gh CLI command	no	wraps the GitHub API

2. Every row above was reproduced in a sandbox for this chapter. The remote was pointed at `203.0.113.9`, an address reserved by RFC 5737 for documentation, so nothing could answer. The outage was real, not faked with a flag.

```
git log --oneline -1        rc=0  f657694 First commit
git branch -a               rc=0  * main
git switch -c feature/x     rc=0
git commit --allow-empty    rc=0
git tag v0.9                rc=0
git diff HEAD~1 --stat      rc=0
git rev-parse origin/main   rc=0  f6576945ec22d250a17a6fe3...
git status -sb              rc=0  ## feature/x
```

3. Note line seven. `git rev-parse origin/main` succeeded and printed a full SHA while the server was unreachable. That is the cached photo.
4. In the same second, on the same repository, the network commands:

```
$ git ls-remote origin
ssh: connect to host 203.0.113.9 port 22: Connection timed out
fatal: Could not read from remote repository.

$ git fetch origin
ssh: connect to host 203.0.113.9 port 22: Connection timed out
fatal: Could not read from remote repository.

$ git push origin main
ssh: connect to host 203.0.113.9 port 22: Connection timed out
fatal: Could not read from remote repository.
```

5. Three failures, all identical, all at the same layer: the connection never opened. Git never got as far as speaking git.
6. That is the same shape as the reader's own evidence. `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and then timed out after 15 seconds with no response of any kind. No RST, no ICMP unreachable, silence.
7. What that proves: nothing on that path answered. What it does not prove: who dropped it. The same request succeeded instantly over mobile data, which is consistent with a silent drop somewhere on the fixed-line path, and not consistent with GitHub being down.

40.8.4 PLAIN - what is really happening inside

1. Take the three cases apart, one at a time.
2. Case one, a local git command. `git commit` reads the index file, writes blob, tree and commit objects under `.git/objects`, and rewrites one 41-byte ref file. Zero sockets are opened. There is nothing to fail.

3. Case two, a remote git command. `git push` must change a ref on another machine. It resolves a name, opens TCP port 22 or 443, authenticates, runs the remote helper, and negotiates. Any broken step ends it.
4. Case three, a GitHub feature. Opening a pull request is an HTTPS request to `api.github.com`. There is no offline path, because there is no local copy of the data and no local code that understands it.
5. Now the important consequence for the reader. Because case one never touches a network, work does not stop. The reader committed to branches all through the outage and pushed them later, when the path came back.
6. Nothing had to be redone. The commits made during the outage were already complete, already hashed, already in the object database. Pushing later just copied objects that already existed.
7. That is the entire practical payoff of the distributed design, and the reader lived it.
8. One more piece from the same session. Twice, a push failed with `send-pack: unexpected disconnect while reading sideband packet`.
9. That message is different from the ones above. It means the connection was established, authentication passed, objects were sent, and the link died while git was reading the server's reply.
10. So the error tells you the answer never arrived. It tells you nothing about whether the server acted before the link died.
11. Both times the operation had not been applied. But that could not be known from the message. It had to be re-checked with `git ls-remote origin`, which asks the server what its refs are now.
12. The rule to carry away: a failure while reading a response is not evidence that the write did not happen. Only a fresh query is evidence.

40.8.5 TECHNICAL - the engineer's version

1. What each operation actually needs on the wire.

Operation	Endpoint	Port
git over SSH	ssh to the host	22
git over HTTPS	the smart HTTP path	443
GitHub REST API	api.github.com	443
GitHub GraphQL API	api.github.com/graphql	443
gh CLI	the same two APIs	443
Web pages, PR view	github.com	443
Release asset download	objects storage host	443
Actions log fetch	a logs host	443

2. Note what that table means. Everything except plain SSH shares one port and often one path. When that path is silently dropped, git over HTTPS and the whole API fail together, which is what the reader observed.
3. Plain SSH on port 22 is a different destination and can survive when HTTPS does not. In the reader's session one transport worked while the other failed, in the same minute, on the same machine.
4. That asymmetry is a diagnostic, not a fix. If SSH works and HTTPS does not, the fault is path-specific or port-specific, not a dead server.
5. Failure taxonomy, with the exact signal for each:

Symptom	What it means	Proven or suggested
Connection refused	a host sent RST	server or firewall said no
Timed out, no reply	silent drop	proven: nothing answered
TLS handshake error	reached, policy fail	proven: TCP worked
403 from the API	authenticated, denied	proven: service is up

Symptom	What it means	Proven or suggested
sideband disconnect	reply lost mid-stream	write status unknown

6. Only the last row is ambiguous, and it is the one that bites. The git protocol sends the report-status response inside sideband channel 1 during `receive-pack`. If the stream dies there, the client has no report.
7. The correct recovery is a re-query, never a retry decided by guesswork:

```
$ git ls-remote origin refs/heads/main
0a95cc8a5f0b...   refs/heads/main
```

8. Compare that SHA with your local `main`. If they match, the push landed. If not, it did not. That is a fact from the server, not an inference.
9. There is a second reason the re-query matters. A push is not atomic across refs by default. Pushing several refs at once can leave some applied and some not, unless you use `--atomic`, added in Git 2.4 (April 2015), which makes the whole push succeed or fail as one unit on servers that support it.
10. Chapter 37 gave the local versus remote split. This chapter adds the second axis. Together they explain every row of the table in 40.8.3:

```
                    | needs no network | needs the network
 ------------------------------------------------------
  pure git          | commit, log,     | fetch, push,
                    | merge, rebase,   | clone, ls-remote
                    | tag, bisect      |
 ------------------------------------------------------
  forge feature     | (nothing)        | PR, review, CI,
                    |                  | issues, releases
```

11. The bottom-left cell is empty, and that emptiness is the point. There is no such thing as an offline pull request.
12. Tools that appear to break this rule cache data locally, which is the same stale-photo problem as `origin/main` and should be treated the same way.

40.8.6 WORDS - remember these

1. Local operation — anything git can do with your own files — a command whose entire working set is the object database, index and refs on this disk.
2. Remote-tracking ref — your last photo of the server — a ref under `refs/remotes/` updated only by fetch, never live.
3. Silent drop — a packet vanishes with no reply — no RST and no ICMP time-exceeded, observed as a connection timeout.
4. Sideband — the channel carrying the server's messages — multiplexed stream inside the pack protocol, where the report-status reply travels.
5. Re-query — ask again instead of guessing — using `git ls-remote` to learn the server's current refs after an ambiguous failure.
6. Atomic push — all refs or none — `git push --atomic`, added in Git 2.4 (April 2015), requiring server support.

40.9 The self-hosted alternatives

40.9.1 PLAIN - in simple words

1. If you do not want to use GitHub, you have four honest choices. Here they are from heaviest to lightest.
2. GitLab Community Edition. Close to GitHub in features, and heavy to run.

3. Gitea, or its community fork Forgejo. Most of what a small team needs, in a program small enough for a Raspberry Pi.
4. Bitbucket Data Center from Atlassian. Bought mainly by companies that already run Jira and want one supplier.
5. Plain SSH remotes with no web interface at all. A machine, an account, and git. Nothing else.
6. All four store bare repositories the same way and speak the same protocol. Every difference is in the layers above git.
7. What you gain by self-hosting: control over the rules, privacy for the code, and no lock-in to one company's features or prices.
8. What you lose is simpler than people admit. You become the operator. Backups are yours. Uptime is yours. Upgrades and security patches are yours. At three in the morning, it is you.

40.9.2 PLAIN – a picture in your head

1. Compare eating in a canteen with cooking at home.
2. In the canteen, somebody else buys the food, cooks it, washes up and keeps the doors open. You accept their menu and pay their price.
3. Cooking at home, you choose everything. You know exactly what is in the food. Nobody can change the price or close the kitchen.
4. But you also shop, cook, wash up and mend the cooker. Every day. Including days you are ill and days you are away.
5. Most people cook at home for four and hire a caterer for four hundred.
6. Self-hosting scales the same way. It is genuinely easy for a small trusted group and genuinely hard for a large public project full of strangers.

Where this comparison breaks:

7. A broken cooker ruins one dinner. A broken git server does not lose your history at all, because every clone is a complete copy. The stakes are lower than the picture suggests.
8. What an outage really costs you is the meeting point, plus the metadata: issues, reviews and build results. Those are the parts with no second copy on anybody's laptop.

40.9.3 PLAIN – a worked example

1. Here is the smallest useful server anyone can build, and it needs no software beyond git and an SSH daemon.
2. Make one account on a machine, called `git`. Set its login shell to `git-shell`, a program that ships with git and refuses everything except git's own transport commands.
3. Put each developer's public key in that account's `authorized_keys` file.
4. Proof that `git-shell` really is a lock and not a suggestion:

```
$ git-shell -c "ls -la"
fatal: unrecognized command 'ls -la'
```

5. And proof that the git commands still work through it. This is the real first bytes of a ref advertisement, wrapped to fit the page:

```
$ git-shell -c "git-upload-pack '/srv/git/proj.git'"
01032304fd03...16d6 HEAD multi_ack thin-pack side-band
side-band-64k ofs-delta shallow deepen-since deepen-not
no-progress include-tag multi_ack_detailed
symref=HEAD:refs/heads/main object-format=sha1
agent=git/2.43.0
0046a3bfdce1...3a refs/heads/feature/login
```

6. That is the whole server. Somebody who logs in with that key can clone, fetch and push, and cannot get a shell, list files or read other accounts.
7. Cost of running it: git itself. Measured in the sandbox for this chapter, peak memory of the git process during real work:

```
git clone (11,772 commits)   0.58 s   peak RSS 42.1 MB
git gc    (blobless clone)   0.93 s   peak RSS 39.2 MB
git status (1,356 files)     0.29 s   peak RSS 16.5 MB
git log --oneline (all)      0.17 s   peak RSS 14.8 MB
```

8. Tens of megabytes, for a few seconds, per operation. That is why a plain SSH git server runs happily on the smallest machine you can rent.
9. Now compare that with what each option needs before it will start at all.

Option	Min RAM	Web UI	Effort
SSH plus git-shell	tens of MB	none	very low
Gitolite added	tens of MB	none	low
cgit or gitweb	~128 MB	read only	low
Gitea or Forgejo	1 GB	full	low
GitLab CE	8 to 16 GB	full	high
Bitbucket Data Center	8 GB and up	full	high, paid

10. And what you actually get for that memory.

Feature	SSH only	Gitea, Forgejo	GitLab CE
Push and pull	yes	yes	yes
Server-side hooks	yes	yes	yes
Web code browsing	no	yes	yes
Pull requests	no	yes	yes, called MRs
Issues, labels	no	yes	yes
Built-in CI	no	yes	yes
Package registry	no	yes	yes
Fine-grained roles	limited	yes	yes

40.9.4 PLAIN - what is really happening inside

1. The four options differ in exactly one way: how many processes must be running before a developer can do anything.
2. Plain SSH: one process, and only while somebody is connected. `sshd` accepts the connection, `git-shell` checks the command, and `git-upload-pack` or `git-receive-pack` does the work and exits.
3. Nothing runs between pushes. There is no database, no queue, no web server and no cache. That is why it costs almost nothing.
4. Gitolite adds a single Perl program in front. It reads one configuration repository that lists users, repositories and per-branch rules, and it installs an `update` hook to enforce them.
5. Gitea and Forgejo are one compiled program written in Go. It contains the web server, the issue tracker, the review system and its own SSH server. Storage is SQLite for small installations, or PostgreSQL or MySQL.
6. GitLab is not one program. A standard installation runs a Ruby on Rails application, background job workers, PostgreSQL, Redis, a reverse proxy, a metrics stack, and Gitaly, which is GitLab's own service that every other part must call in order to touch git at all.

7. That list is the entire reason for the memory difference. It is not that GitLab is badly built. It is that it is many services, and each one has a floor.
8. In every case the bytes on the wire are identical. A developer cloning from plain SSH and a developer cloning from GitLab run the same client, speak the same protocol and get the same objects.

40.9.5 TECHNICAL - the engineer's version

1. Provenance and licensing, which matter more than feature lists when you are choosing something to depend on for ten years.

Project	Origin	Licence
Gogs	2014, by Jiahua Chen	MIT
Gitea	Nov 2016 fork of Gogs	MIT
Forgejo	Dec 2022 fork of Gitea	GPL since Aug 2024
GitLab CE	2011, D. Zaporozhets	MIT
GitLab EE	same codebase, add-ons	proprietary
Bitbucket	2008, by Jesper Nohr	proprietary

2. The Gitea and Forgejo split is a governance story, not a technical one. Gitea's trademarks and operations were transferred to a company in 2022, contributors asked for community control, the request was refused, and Forgejo was forked in December 2022. Codeberg e.V., a German non-profit association, leads it.
3. Forgejo stopped tracking Gitea after Gitea 1.21, in February 2024, and became a hard fork. It relicensed from MIT to the GNU General Public Licence in August 2024. Code written before that remains MIT.
4. Bitbucket was founded in 2008 by Jesper Nohr, acquired by Atlassian on 29 September 2010, and dropped Mercurial support in July 2020. Support for the self-hosted **Bitbucket Server** product ended on 15 February 2024. Self-hosting now means Bitbucket Data Center, which is licensed and priced for larger organizations.
5. That last date is a lesson in its own right. A self-hosted product can be discontinued by its vendor. Self-hosting removes the hosting dependency; it does not by itself remove the vendor dependency. Only an open licence with a living community does that.
6. Hardware, from the projects' own documentation as of 2026. Gitea states that "2 CPU cores and 1GB RAM is typically sufficient for small teams/projects" and that a Raspberry Pi 3 is enough for small workloads. GitLab's single-node baseline is 8 vCPU and 16 GB of RAM, with 8 GB possible only with extra configuration. Older GitLab documentation quoted 4 GB, so check the version-specific page rather than trusting a number you remember.
7. That is a factor of roughly sixteen in memory between Gitea and GitLab for the same core job. It is the single most useful figure in this section.
8. Access control on a plain SSH server, in increasing order of precision: UNIX file permissions on the repository directory; one shared `git` account with `git-shell`; Gitolite for per-repository and per-branch rules; and server-side `pre-receive` or `update` hooks for anything else.
9. Read-only web browsing without a full forge: cgit, a C program, or gitweb, which ships with git itself. Both render a repository as HTML and neither has accounts, issues or reviews.
10. Honest trade-off summary. Self-hosting is a good decision when the group is small and trusted, when the code must not leave your control for legal reasons, when your costs at a hosted provider have become large, or when you need rules a hosted provider will not give you.
11. It is a poor decision when your project depends on drive-by contributions from strangers, when nobody on the team wants to be on call, or when the real cost of an engineer's time doing operations exceeds the subscription you were avoiding.
12. Experts disagree about the middle ground. One camp argues that a Forgejo instance is so cheap to run that any team should own its own. The other argues that discovery, and the fact that contributors

already have GitHub accounts, outweighs everything else for public projects. Both are right about different projects, and the deciding question is whether strangers need to contribute.

40.9.6 WORDS - remember these

1. Self-hosting — running the server yourself — operating the repository host, its storage, its backups and its upgrades on infrastructure you control.
2. git-shell — a login that can only do git — a restricted shell shipped with git that accepts only `git-upload-pack`, `git-receive-pack` and `git-upload-archive`.
3. Gitolite — a rules file in front of a git server — a Perl layer using one configuration repository and an `update` hook for per-branch access control.
4. Forge — a website around repositories — the general term for GitHub, GitLab, Gitea, Forgejo and Bitbucket.
5. Gitaly — GitLab's git service — the component every other GitLab process must call to read or write a repository.
6. Vendor lock-in — hard to leave — dependence on features whose data has no portable export, as opposed to the commit history, which always exports.
7. Operator — the person who is responsible when it breaks — the role you take on the moment you self-host, covering backups, uptime and patching.

40.10 Migrating and mirroring

40.10.1 PLAIN - in simple words

1. Two jobs get confused with each other, so separate them first.
2. Mirroring means keeping a second copy that stays in step with the first. Both live. You did not leave anywhere.
3. Migrating means moving to a new home and shutting the old one, or leaving it as an archive.
4. For the git part, both jobs are easy and take one or two commands.
5. For everything else, both jobs are hard, and some of it simply cannot be moved at all.
6. Here is the sentence to remember, and it is the honest warning of this whole section: **your commit history is portable, and almost nothing else is.**
7. Commits, branches, tags and file contents move perfectly, byte for byte, with identical hashes.
8. Issues, pull requests, review approvals, build history, release downloads, stars, webhooks and protection rules do not move by themselves, because none of them is in git.

40.10.2 PLAIN - a picture in your head

1. Think about moving house.
2. Your books, clothes and furniture go in boxes. They arrive exactly as they left. Nothing about them changes.
3. The fitted kitchen does not come. The neighbours do not come. Your children's school place does not come. The postal redirection has to be arranged, lasts a while, and then stops.
4. Git history is the boxes. Everything the website added is the fitted kitchen and the school place.
5. And notice which one people actually worry about on moving day. Not the boxes.

Where this comparison breaks:

6. Moving house removes things from the old place. Cloning does not. After you migrate, the old repository still exists, complete, and can still serve.
7. That is why a git migration is unusually safe. You can run both for weeks and switch back with one `git remote set-url`.

40.10.3 PLAIN – a worked example

1. First, what `--mirror` actually gives you. This is a real run.

```
$ git clone --mirror proj mirror.git

$ git -C mirror.git config --get remote.origin.mirror
true

$ git -C mirror.git config --get remote.origin.fetch
+refs/*:refs/*

$ git -C mirror.git show-ref
a3bfdce1... refs/heads/feature/login
2304fd03... refs/heads/main
fb32611e... refs/tags/v1.0
```

2. Compare that with an ordinary clone of the same repository:

```
$ git clone proj plain
$ git -C plain show-ref
2304fd03... refs/heads/main
2304fd03... refs/remotes/origin/HEAD
a3bfdce1... refs/remotes/origin/feature/login
2304fd03... refs/remotes/origin/main
fb32611e... refs/tags/v1.0
```

3. Read the difference carefully. The ordinary clone has one local branch and puts the rest under `refs/remotes/origin/`. The mirror keeps every ref at its original name.
4. That is the whole point of a mirror: it is a copy of the ref namespace, not a copy for working in.
5. Second, keeping a mirror up to date. One command, and `--prune` matters:

```
$ git -C mirror.git remote update --prune
From /tmp/mir/up
 - [deleted]   (none) -> hotfix/urgent
```

6. Third, pushing one commit to two servers with one command. Add a second push URL to the same remote:

```
$ git remote set-url --add --push origin /tmp/lab3/srv1.git
$ git remote set-url --add --push origin /tmp/lab3/srv2.git
$ git remote -v
origin  /tmp/lab3/srv1.git (fetch)
origin  /tmp/lab3/srv1.git (push)
origin  /tmp/lab3/srv2.git (push)

$ git push origin main
To /tmp/lab3/srv1.git
   99b5afc..a98de55  main -> main
To /tmp/lab3/srv2.git
 * [new branch]      main -> main
```

7. One `git push`, two servers, and both now hold the identical SHA `a98de5517c0e80180ed2a1ccd6795d47f155beb5`.
8. Fourth, the trap. A mirror push is a replica, not a backup. Watch what happens when a branch is deleted upstream:

```
$ git -C up branch -D hotfix/urgent
Deleted branch hotfix/urgent (was d9a902a).
```

```
$ git -C mir.git remote update --prune
 - [deleted]  (none) -> hotfix/urgent

$ git -C mir.git push --mirror /tmp/mir/dest.git
To /tmp/mir/dest.git
 - [deleted]  hotfix/urgent
```

9. The deletion was faithfully copied to the second server. A mirror reproduces mistakes exactly as well as it reproduces work. Keep that in mind for the backup section.

40.10.4 PLAIN - what is really happening inside

1. `git clone --mirror` does three things and nothing magical.
2. It makes the new repository bare, so there is no working tree.
3. It sets `remote.origin.mirror` to `true`.
4. It sets the fetch refspec to `+refs/*:refs/*`, meaning "take every ref, of every kind, and store it under the same name, forcing where needed".
5. Compare that with the ordinary clone refspec, `+refs/heads/*:refs/remotes/origin/*`, which takes only branches and files them under a separate prefix.
6. That single line of configuration is the entire difference between a copy you work in and a copy that stands in for the original.
7. On the push side, `git push --mirror` sends every local ref and deletes every destination ref that you do not have. That is why the deletion propagated above.
8. One consequence surprises everybody who mirrors a GitHub repository. `+refs/*:refs/*` includes GitHub's pull request refs. Here is a real mirror clone of a public repository:

```
$ git clone --mirror https://github.com/octocat/Hello-World hw.git
$ git -C hw.git show-ref | wc -l
3439
$ git -C hw.git show-ref | grep -c refs/pull/
3436
```

9. Three branches, and three thousand four hundred and thirty-six pull request refs. If you then push that mirror to a new host, you are trying to create three thousand refs the new host neither wants nor allows.
10. The fix is to fetch or push only what you mean, using an explicit refspec such as `+refs/heads/*:refs/heads/*` and `+refs/tags/*:refs/tags/*`.

40.10.5 TECHNICAL - the engineer's version

1. What actually travels, and what does not.

Thing	Inside git?	Copied by clone
Commits, trees, blobs	yes	yes
Branches	yes	yes, renamed
Lightweight tags	yes	yes
Annotated tag objects	yes	yes
Notes in refs/notes	yes	only with refspec
Signatures on commits	yes	yes
Issues and comments	no	no
Pull requests, reviews	no	no
CI runs and logs	no	no
Releases and assets	no	no
Wiki content	separate repo	separate clone

Thing	Inside git?	Copied by clone
Webhooks, secrets	no	no
Protection rules	no	no
Stars, forks, watchers	no	no

2. The wiki row is worth knowing precisely: GitHub stores a repository wiki as a separate git repository, reachable at the repository address with `.wiki.git` on the end. So the wiki does migrate, but only if you deliberately clone it as a second repository.
3. A correct migration of the git part, in full:

```
git clone --mirror https://github.com/you/proj.git
cd proj.git
git remote set-url --push origin git@newhost:you/proj.git
git push --prune origin \
    '+refs/heads/*:refs/heads/*' \
    '+refs/tags/*:refs/tags/*'
```

4. That is deliberate rather than clever. It moves branches and tags, prunes what is gone, and does not attempt to move `refs/pull/*`.
5. Verify the move by comparing ref lists rather than by looking at a web page:

```
OLD=https://old/proj.git
NEW=git@newhost:you/proj.git
git ls-remote --heads --tags $OLD | sort > old.txt
git ls-remote --heads --tags $NEW | sort > new.txt
diff old.txt new.txt && echo "identical"
```

6. Metadata is a separate project with separate tools, and no standard format exists. As of 2026 the realistic options are:

Source and target	Tool	What it carries
GitHub to Gitea, Forgejo	built-in migration	issues, PRs, labels
GitHub to GitLab	GitLab importer	issues, MRs, comments
GitLab to GitLab	project export file	most things
Anything to anything	the REST API	whatever you code

7. For a do-it-yourself export the practical route is the API. The GitHub CLI can page through issues and write JSON, which you then store in the repository itself so it is versioned with the code:

```
gh api --paginate repos/OWNER/REPO/issues?state=all \
    > issues.json
gh api --paginate repos/OWNER/REPO/pulls?state=all \
    > pulls.json
```

8. GitHub also offers migration archives through its migrations API for organizations, producing a downloadable archive of repositories and associated metadata. Availability and contents depend on your plan, so check before you rely on it.
9. Now the honest warnings, which matter more than the commands.
10. Issue numbers will not survive. Every forge assigns its own numbering, so “fixes #42” in a commit message will point at the wrong thing after migration. The commit message is immutable, so this cannot be corrected without rewriting history.
11. Cross-references, mentions of users, and links between issues and pull requests are stored as identifiers that mean nothing on the new host.

12. Timestamps and authorship on imported issues usually become "imported by the migration account on the day of import" unless the importer has special support, because most APIs will not let you post as someone else.
13. Attachments in comments are files on the old host, served from the old host's storage, and vanish when the repository does.
14. Build history and logs are effectively unmovable. Treat them as data with a retention period, not as a permanent record.
15. So the practical rule for anything you truly cannot lose: put it in the repository. A decision written in a Markdown file in `docs/` migrates perfectly, forever, because it is a blob. The same decision written in an issue comment is the platform's, not yours.

40.10.6 WORDS - remember these

1. Mirror clone — a copy of every ref under its own name — `git clone --mirror`, which sets `remote.origin.mirror=true` and the refspec `+refs/*:refs/*` on a bare repository.
2. Mirror push — make the far side match me exactly — `git push --mirror`, which creates, updates and deletes refs to match the local set.
3. Prune — remove refs the other side no longer has — `--prune` on fetch, remote update or push.
4. Second push URL — one remote, two destinations — extra push URLs added with `git remote set-url --add --push`.
5. Migration — moving to another host — copying refs and objects with git, then copying metadata with each platform's own tools, imperfectly.
6. Platform metadata — everything not in git — issues, reviews, runs, releases and rules, which have no interchange standard.

40.11 Working with others, practically

40.11.1 PLAIN - in simple words

1. There are only two ways to arrange a team around a repository.
2. The shared repository model. Everyone can push to the same repository, and they push branches, not straight to main.
3. The fork model. Nobody outside the core team can push at all. Outsiders make their own server-side copy, push there, and ask for their work to be taken.
4. The shared model is for people you trust: a company team, a class group, a startup.
5. The fork model is for strangers: any public project where anyone in the world may send a change.
6. Both end at the same place, a pull request, so they look identical in the web interface. The difference is only in who may write where.
7. On top of that choice sits a second one: how branches are organized over time. There are three well-known answers, and they have real trade-offs.
8. And on top of that sits one rule everybody agrees on: protect main.

40.11.2 PLAIN - a picture in your head

1. Think of a shared document at a college.
2. The shared repository model is a group project. Everyone has edit rights, but the agreement is that you work on your own page and ask before merging your page into the final report.
3. The fork model is a published book. You cannot write in the book. You make a photocopy, mark it up, and post your marked copy to the editor, who decides whether to take your change.
4. A publisher does not give edit rights to every reader. That would be absurd, and it is exactly why open-source projects use forks.
5. A group of four friends does not photocopy the report for each other. That would be pointless ceremony, and it is why company teams use the shared model.

Where this comparison breaks:

6. A photocopy goes out of date the moment the book changes. A fork does not have to. You can add the original as a second remote and pull from it, so your copy stays current.
7. And the editor in this picture merges by hand. A pull request merge is performed by the server, exactly, using git.

40.11.3 PLAIN - a worked example

1. Here is the fork-and-pull-request path drawn out. Only two steps in it are git; the rest are the website.

```
upstream/proj        you cannot write here
     |
     |  "Fork"  (GitHub action, not a git command)
     v
you/proj             your copy on the server, you own it
     |
     |  git clone            (git)
     v
laptop: main -> feature/login
     |
     |  git push origin feature/login   (git)
     v
you/proj  feature/login
     |
     |  "Open pull request"  (GitHub action)
     v
upstream: refs/pull/N/head -> review -> merge
```

2. In the shared repository model, the first two boxes disappear. You clone `upstream/proj` directly, push `feature/login` to it, and open the pull request from there.
3. Now branch names. Git's rules are stricter than people expect, and this is real output from `git check-ref-format`:

```
OK   feature/login
OK   feature/ABC-123-add-login
OK   release/1.0
OK   Feature/Login
BAD  feature/add login
BAD  feature//login
BAD  feature/login.lock
BAD  feature/.hidden
BAD  feature/login~1
BAD  feature/login^
```

4. So a space is illegal, a double slash is illegal, a component starting with a dot is illegal, an ending of `.lock` is illegal, and the characters `~ ^ : ? * [ \` are illegal.
5. `Feature/Login` is legal but a bad idea, because on macOS and Windows the filesystem usually ignores case and two branches differing only in case will collide.
6. And here is the trap that catches every team once:

```
$ git branch feature/login
$ git branch feature
fatal: cannot lock ref 'refs/heads/feature':
 'refs/heads/feature/login' exists;
 cannot create 'refs/heads/feature'
```

7. A loose ref is a file on disk, so `feature` cannot be both a file and a directory. Pick a prefix scheme and never use a bare prefix as a branch.

8. Finally, the three ways a pull request can land. All three of these are real graphs from the same starting point:

```
1. merge commit (--no-ff)
*   0685fe9 Merge pull request: login
|\
| * 00ee7d7 Fix login typo
| * 046ed60 Start login
* | ec53eed Unrelated work on main
|/
* cd18e97 Base commit

2. squash merge
* 23a1f88 Add login (#42)
* ec53eed Unrelated work on main
* cd18e97 Base commit

3. rebase then fast-forward
* 12a1676 Fix login typo
* 0ae1582 Start login
* ec53eed Unrelated work on main
* cd18e97 Base commit
```

9. Look at what each one costs. The merge commit keeps every step and the shape. The squash keeps one tidy commit and throws away the steps. The rebase keeps the steps and throws away the shape, and it made new SHAs: `046ed60` became `0ae1582`.
10. That last point is the reader's own experience. Rebasing a local branch onto a moved `main` changed the branch's SHA, because rebase does not move commits. It creates new commit objects with new parents, and therefore new hashes.

40.11.4 PLAIN - what is really happening inside

1. A fork, mechanically, is a repository on the server plus a database row saying "this came from that". Git has no idea it happened.
2. When you fork on GitHub, the new repository may share object storage with the original, which is why forking a huge repository is instant. That is an implementation detail of one product, not part of git.
3. A pull request from a fork works because both repositories are on the same server, so the server can read your head commit directly.
4. Now the three branching models, described by what they actually do to the graph.
5. Git flow, published by Vincent Driessen on 5 January 2010, keeps two permanent branches, `main` and `develop`, plus `feature/*`, `release/*` and `hotfix/*`. Work flows feature into develop, develop into a release branch, release into main and back into develop.
6. GitHub flow, published by Scott Chacon on 31 August 2011, keeps one permanent branch. Branch off main, open a pull request, get review, merge, deploy. There is no develop branch and no release branch.
7. Trunk-based development keeps one permanent branch and adds a rule about time: branches live hours, not days, and unfinished work is hidden behind feature flags rather than hidden on a branch.
8. Notice what the three actually disagree about. Not tooling. They disagree about how long a change may live away from the mainline.
9. Long-lived branches are the cost. Every day a branch lives, the mainline moves under it, and the merge gets harder. This is not an opinion; it is what a graph does when two lines diverge.
10. Protecting main is the mechanism that makes any of the three safe. A protected branch cannot be pushed to directly, cannot be force-pushed and cannot be deleted, and merges into it require whatever checks you set.
11. Without protection, all three models are just polite suggestions, exactly like a client-side hook.

40.11.5 TECHNICAL - the engineer's version

1. The two collaboration models compared on the things that actually differ.

Question	Shared repo	Fork model
Who may push branches	the team	only your fork
Write access needed	yes	no
Number of remotes	one	two
Suits	trusted teams	public projects
CI secrets exposure	trusted	needs care

2. The last row is a real security point. A workflow triggered by a pull request from a fork must not be given repository secrets, or any stranger could exfiltrate them. GitHub's `pull_request` event withholds secrets from fork pull requests for exactly this reason, while `pull_request_target` runs with them and is dangerous if misused.
3. The three branching models, with honest trade-offs.

Model	Long-lived branches	Best for
Git flow	main and develop	versioned releases
GitHub flow	main only	continuous deploy
Trunk-based	main only	high-rate teams

4. Driessen added a note to his own 2010 article in March 2020 saying that if your team does continuous delivery you should adopt a much simpler workflow such as GitHub flow, and that git flow still fits software that is explicitly versioned or must support several versions in the wild.
5. That is the fairest summary available, and it comes from the author of the model people argue about.
6. The research position: the 2018 book Accelerate, by Nicole Forsgren, Jez Humble and Gene Kim, reports that high-performing teams have fewer than three active branches, branch lifetimes under a day, and rarely or never have code freezes.
7. Read that as a correlation found in survey data, not as a law. The critics' point is fair too: trunk-based development requires strong automated tests and feature flags, and a team without them will simply break main faster.
8. Recommended day-one setup for a small team, concretely. This is a working configuration, not a philosophy.
9. Set the default branch name once per machine, so nobody argues:

```
git config --global init.defaultBranch main
```

10. That option arrived in Git 2.28, released 27 July 2020. GitHub changed the default branch for newly created repositories to `main` in October 2020. Older repositories were not renamed, which is why both names are still everywhere.
11. On the repository, turn on branch protection for `main` with these settings: require a pull request before merging, require at least one approving review, require status checks to pass, require branches to be up to date before merging, and block force pushes and deletions.
12. Choose one merge method and disable the others, so history has one shape. Squash merge is the usual choice for small teams because it makes `main` one commit per change and keeps `git log --oneline` readable.
13. Agree a branch prefix scheme and write it in the README. A workable one is `feature/`, `fix/`, `chore/`, `docs/`, plus the ticket identifier, giving names like `feature/ABC-123-add-login`.
14. Add a `CODEOWNERS` file even with three people. It costs two lines and it means nobody has to remember who reviews what.

15. Add one CI workflow that runs the tests on every pull request, and make it a required check. An unenforced test suite decays within weeks.
16. Turn on tag protection or use signed tags for anything you release, so a version number cannot be quietly repointed.
17. Verify CI by run identifier against the exact head SHA rather than by trusting a green tick beside a branch name. In the reader's own session that distinction was the difference between believing a result and knowing one.
18. Do not add: a develop branch, a release branch per version, or a rule that two people must approve, until you have a concrete reason. Each of those is a real cost paid every day.

40.11.6 WORDS - remember these

1. Fork — your own server-side copy of a project — a forge-level repository copy plus a recorded parent relationship, invisible to git.
2. Upstream — the original project you forked from — conventionally a second remote named `upstream` alongside `origin`.
3. Shared repository model — everyone pushes branches to one repository — collaboration requiring write access, with policy enforced by protection rules.
4. Git flow — the two-permanent-branch model from 2010 — `main` plus `develop` with `feature/*`, `release/*` and `hotfix/*` branches.
5. GitHub flow — branch, pull request, merge, deploy — a single permanent branch model published in 2011.
6. Trunk-based development — branches measured in hours — mainline development with short-lived branches and feature flags instead of long branches.
7. Feature flag — a switch that hides unfinished work — a runtime condition allowing incomplete code to be merged but not executed.
8. Protected branch — main cannot be pushed to directly — a server-side rule blocking direct pushes, force pushes and deletion.
9. Squash merge — many commits become one — a merge that applies the combined change as a single new commit with one parent.
10. check-ref-format — the rules for legal ref names — `git check-ref-format`, which rejects spaces, `..`, `~`, `^`, `:`, `?`, `*`, `[`, trailing `.lock` and leading dots.

40.12 Monorepos and large repositories

40.12.1 PLAIN - in simple words

1. A monorepo is one repository holding many projects that could have been separate repositories.
2. Big companies use them for one reason above all others: one commit can change a library and every user of that library at the same time.
3. That removes the hardest problem in large systems, which is not writing code but keeping many versions of many pieces compatible.
4. You also get one place to search, one build configuration, and no arguing about which repository a change belongs in.
5. The price is size, and size hurts git in specific, measurable ways.
6. Clones get slow, because a clone copies the whole history by default.
7. Everyday commands get slow, because git checks every file it tracks.
8. Tools get slow or give up, because editors and search programs were not written with a million files in mind.
9. Git has four answers, and you can combine them: shallow clone, partial clone, sparse checkout, and Git LFS for large files.

10. And one weakness has no good answer: binary files. Git stores them, but it stores them badly, and this section will show exactly how badly.

40.12.2 PLAIN – a picture in your head

1. Think of a library that keeps every edition of every book it has ever held.
2. If you want to borrow one novel, the librarian hands you the entire archive on a trolley, including every draft of every book, and says "it is all yours now, take it home".
3. That is `git clone`. It is wonderful once you have it and painful the first time.
4. A shallow clone is asking for only the newest edition of everything.
5. A partial clone is asking for the catalogue now and the actual pages later, only when you open a book.
6. A sparse checkout is taking the whole archive home but only unpacking the one shelf you work on.
7. Git LFS is the library keeping the heavy art books in a warehouse and giving you a slip of paper saying where each one is.

Where this comparison breaks:

8. Fetching a page later feels free in a library. In a partial clone it is a network round trip, and if you ask for thousands of pages one at a time it is dramatically slower than having taken the trolley. There are real numbers for this in the worked example, and they are worse than most people guess.

40.12.3 PLAIN – a worked example

1. Every number here was measured for this chapter against one real public repository, the GitHub CLI source, which has 11,772 commits and 1,356 tracked files. It is a medium repository, not a giant one.
2. First, the three ways to clone it.

Clone type	.git size	Time	Commits
Full clone	81 MB	5.9 s	11,772
Blobless, blob:none	27 MB	3.4 s	11,772
Shallow, depth 1	15 MB	2.3 s	1

3. Read the middle row carefully. The blobless clone still has every commit and every tree. It has 11,772 commits, same as the full clone. What it does not have is old file contents.
4. That is usually the right trade, because history browsing needs commits, and commits are small. File contents are what is big.
5. Object counts make the same point:

```
full clone     in-pack 90,539   size-pack 79.75 MiB
blobless       in-pack 64,891   size-pack 25.66 MiB
shallow  1     in-pack  1,776   size-pack 14.58 MiB
```

6. Second, sparse checkout, which is about the working tree rather than the history. This checks out one directory only:

```
$ git sparse-checkout set --cone pkg/cmd/pr
$ git ls-files | wc -l
1356
$ find . -path ./.git -prune -o -type f -print | wc -l
115
```

7. Git still tracks all 1,356 files. Only 115 exist on disk. The working tree shrank from 27 MB to 1.4 MB.
8. The index shows how it is done. 1,241 entries are marked skip-worktree and 115 are present.
9. Third, the honest cost of a partial clone. The same command in both:

Command	Full clone	Blobless clone
git log -p -300	0.197 s	106 s

10. That is not a typo. Reading the patch text of the last 300 commits took a fifth of a second with all objects present, and one minute forty-six seconds when each missing blob had to be fetched over the network.
11. The `.git` directory grew from 27 MB to 29 MB during that command, and 51 extra pack files appeared, each marked `.promisor`, meaning "fetched lazily from the promising remote".
12. So partial clone is not free speed. It moves cost from clone time to the first command that needs old content.
13. Fourth, Git LFS. Here is what git actually stores when a 1 MiB binary file is tracked by LFS:

```
$ git cat-file -p HEAD:blob.bin
version https://git-lfs.github.com/spec/v1
oid sha256:5e20a633da26ff9bceb11f08d4671c234
38c49362cefce5042e8c77bb58bbdf9
size 1048576

$ git cat-file -s HEAD:blob.bin
132
```

14. The object in git is 132 bytes of text. The real megabyte lives outside the object database, under `.git/lfs/objects`, and is fetched by a separate program over a separate protocol.
15. Note the `oid` line. It is a SHA-256 of the file contents, and it matches what `sha256sum` reports for the real file. The pointer is content-addressed, just like git itself.
16. Fifth, why binaries are git's weak point. Two repositories, ten commits each, both starting from nothing:

Repository, 10 revisions	Packed size
1 MiB random binary	10.01 MiB
2.1 MB text, tiny edits	148.27 KiB

17. Ten revisions of a one-megabyte binary cost ten megabytes. Ten revisions of a two-megabyte text file cost 148 kilobytes.
18. The text repository is fourteen times smaller while holding twice as much content per revision. That gap is the whole story of binaries in git.

40.12.4 PLAIN - what is really happening inside

1. Git compresses history in two stages: zlib on each object, then delta compression, where one object is stored as a small set of instructions for turning another object into it.
2. Delta compression is why a text repository stays tiny. Change one line in a 60,000-line file and the delta is a few dozen bytes.
3. A compressed or random binary file has no useful similarity to the previous version. Change one pixel in a PNG and the compressed bytes change everywhere. The delta is as big as the file, so git stores a whole new copy.
4. That is not a bug and no setting fixes it. It is what content-addressed storage plus general-purpose compression can do.
5. Worse, git never forgets. Delete the 500 MB file today and it is still in every clone forever, because it is still reachable from an old commit.
6. Git LFS works around this by keeping a tiny text pointer in the repository and the real bytes elsewhere. A clean filter converts the file to a pointer on `git add`, and a smudge filter converts it back on checkout.
7. So the history contains only 132-byte pointers, which delta beautifully, and the big files are transferred by a separate service that fetches only the versions you actually check out.

8. The four scaling techniques attack four different costs, and it is worth being exact about which is which.
9. Shallow clone cuts the number of commits you download. It is the right tool for a build machine that only needs the newest source.
10. Partial clone cuts the number of objects you download, keeping the full commit graph. It is the right tool for a developer who needs history but not every old file.
11. Sparse checkout cuts the number of files written to disk. It does not reduce download size at all, but it makes `git status`, editors and search tools fast again.
12. Git LFS cuts what enters the history in the first place. It is the only one of the four that changes what is committed.
13. And the index is the quiet cost nobody expects. It is one file listing every tracked path with its metadata. For 1,356 files it was 156,135 bytes. Scale that to a million files and the index alone is over one hundred megabytes, read and written by ordinary commands.

40.12.5 TECHNICAL - the engineer's version

1. Real monorepos, with their published numbers.

Repository	Figure	Source year
Google, all code	86 TB, 2bn lines	2016 paper
Google, source files	~9 million unique	2016 paper
Google, commits	~35 million	2016 paper
Windows in git	~300 GB, 3.5m files	2017 post
Windows pushes	8,421 per day	2017 post

2. The Google figures come from the paper "Why Google Stores Billions of Lines of Code in a Single Repository" by Rachel Potvin and Josh Levenberg, published in Communications of the ACM in July 2016 and describing a January 2015 snapshot. It also reports roughly 25,000 developers, about 16,000 human commits and about 24,000 automated commits on a typical workday.
3. Important honesty point that most retellings omit: Google's monorepo does not run on git. It runs on Piper, an in-house system, with a virtual filesystem client. So it is evidence that monorepos work at scale, not evidence that git works at that scale.
4. The Windows figures come from Brian Harry's post of 24 May 2017, "The largest Git repo on the planet", describing about 4,000 engineers, 1,760 official builds per day across 440 branches, and the Git Virtual File System that Microsoft had to build to make it possible.
5. GVFS was announced in February 2017, later renamed VFS for Git, and the ideas were pushed upstream over the following years as partial clone, sparse checkout in cone mode, the commit-graph file and the built-in filesystem monitor.
6. Version history of the relevant features, so you know what your git can do.

Capability	Git version	Date
Shallow clone, –depth	1.5 era	2007 onward
commit-graph file	2.18	Jun 2018
Partial clone, –filter	2.19 era	Sep 2018
git sparse-checkout	2.25	13 Jan 2020
Built-in fsmonitor	2.37	27 Jun 2022

7. Partial clone filters worth knowing: `--filter=blob:none` omits all blobs; `--filter=blob:limit=1m` omits blobs above a size; `--filter=tree:0` omits trees as well and is meant for tooling, not for humans.

8. A partial clone records the remote as a promisor. Objects arrive lazily, and lazily fetched packs are marked with a `.promisor` file. In the measurement above, 51 such packs appeared during one `git log -p`.
9. The server must permit it. `uploadpack.allowFilter` must be true on the serving side, or `--filter` is silently ignored and you get a full clone.
10. Sparse checkout in cone mode restricts patterns to whole directories, which lets git match paths with prefix comparisons instead of running every pattern against every path. Non-cone mode accepts full gitignore-style patterns and is much slower on large trees.
11. Practical settings for a large repository, all real configuration keys:

```
git config core.fsmonitor true
git config core.untrackedCache true
git config feature.manyFiles true
git maintenance start
```

12. `feature.manyFiles` is an umbrella that sets index version 4 and `core.untrackedCache`. `git maintenance start` schedules background repacking and commit-graph writing rather than waiting for `git gc` to interrupt you.
13. Git LFS, released by GitHub in April 2015, is not part of git. It is a separate program plus a separate HTTP API. `git lfs track "*.psd"` writes a line to `.gitattributes` naming the clean, smudge and merge filters.
14. Consequences people discover late: anyone cloning your repository needs git-lfs installed or they get pointer files instead of content; LFS storage and bandwidth are billed separately by most hosts; and converting an existing repository to LFS rewrites history, which changes every SHA.
15. Alternatives worth naming: `git-annex`, which predates LFS and is more flexible; and simply keeping large assets out of the repository entirely, in object storage referenced by a versioned manifest file.
16. Removing a large file that is already in history requires rewriting it out. The current recommended tool is `git filter-repo`; the older `git filter-branch` is deprecated for being slow and easy to misuse. Both change every commit hash after the touched point, so every collaborator must re-clone.

40.12.6 WORDS - remember these

1. Monorepo — one repository for many projects — a single version-controlled tree containing multiple deliverables with one commit history.
2. Shallow clone — only the newest commits — `--depth N`, producing a truncated history with grafted boundary commits.
3. Partial clone — download objects on demand — `--filter=blob:none` or similar, with the origin recorded as a promisor remote.
4. Promisor remote — the server that owes you objects — the remote a partial clone will lazily fetch missing objects from.
5. Sparse checkout — only some files on disk — index entries marked skip-worktree so git neither writes nor scans them.
6. Cone mode — directory-shaped sparse patterns — the fast form of sparse checkout, matching whole directories by prefix.
7. Delta compression — store the difference, not the file — packfile encoding of one object as instructions against a similar base object.
8. Git LFS — big files kept outside history — a filter that commits a small pointer file and stores real content on a separate service.
9. Pointer file — the 130-odd bytes git actually stores — a three-line text blob giving spec version, a SHA-256 oid and a byte size.
10. filter-repo — the supported way to rewrite history — the recommended replacement for `git filter-branch`, which rewrites every affected commit and therefore every hash.

40.13 Backing up a git repository properly

40.13.1 PLAIN – in simple words

1. People say "every clone is a backup". That is half true, and the missing half is the half that hurts.
2. A clone gives you every commit reachable from the branches you cloned, and the tags. That really is a lot, and it really does survive the server burning down.
3. A clone does not give you other refs, the reflog, stashes, the repository's own configuration, its hooks, or anybody's uncommitted work.
4. A mirror gives you every ref, of every kind, under its original name. That is more than a clone.
5. A mirror still does not give you reflogs, hooks or configuration, and it has a specific danger: it copies deletions too.
6. Neither gives you a single thing that lives on the website: issues, pull requests, reviews, build history, releases, secrets or settings.
7. And neither, on its own, protects you from the most common real disaster, which is not hardware failure. It is somebody force-pushing over a branch, or deleting one, and the automatic mirror faithfully copying that within the hour.
8. A real backup therefore has three properties a mirror does not have: it is dated, it is read-only once written, and it is kept for a while.

40.13.2 PLAIN – a picture in your head

1. Imagine three ways of protecting a handwritten manuscript.
2. A friend borrowed a copy last month. That is a clone. It is genuinely useful, it is complete up to when they took it, and it is out of date.
3. A machine in the next room copies your desk every hour, exactly. That is a mirror. It is always current, and if you spill ink on page 40, the machine copies the ink stain within the hour.
4. A sealed, dated envelope goes into a fire safe every night and stays there for ninety days. That is a backup. It cannot be edited, it can be opened at any date you choose, and the ink stain does not reach it.
5. Most teams have the first two and call it done.

Where this comparison breaks:

6. A copy of a manuscript can be a slightly bad copy. A git clone cannot. Every object is verified against its hash, so a corrupted copy is detectable and usually refuses to load rather than lying to you.
7. And the friend with the month-old copy is genuinely enough to save the project, which is not true of a month-old photocopy of an edited book. Git history is append-only in normal use, so old copies stay valid.

40.13.3 PLAIN – a worked example

1. First, what a clone leaves behind. A real source repository was set up with a stash, extra reflog entries, untracked files, a hook and a local setting, then cloned normally:

```
source repository        clone
  stashes: 1               stashes: 0
  reflog entries: 3        reflog entries: 1
  untracked files: 2       untracked files: 0
  custom hooks: 1          custom hooks: 0
  local user.signingkey    (absent)
  commits: 2               commits: 2
```

2. The commits came across perfectly. Everything else did not.
3. Second, what a mirror adds. The same source, cloned with `--mirror`:

```
$ git -C mtest.git show-ref
dbcd961d... refs/heads/main
f4caa812... refs/stash
141f4579... refs/tags/backup-2026-08-13

$ ls mtest.git/hooks | grep -v sample | wc -l
0
$ git -C mtest.git reflog | wc -l
0
```

4. Look at the second line of `show-ref`. The mirror picked up `refs/stash`, because its refspec is `+refs/*:refs/*` and a stash is a ref. A plain clone never does that.
5. The hooks did not come. The reflog did not come. The source's local configuration did not come.
6. Third, a real backup run, start to finish. This is the whole procedure and every line below is genuine output.

```
$ git -C repo.git remote update --prune

$ git -C repo.git fsck --full
dangling commit de306ffa47620fa7e0d5ef7146...

$ git -C repo.git bundle create vault/proj-2026-08-13.bundle --all

$ ls -l vault/
1045 proj-2026-08-13.bundle

$ git -C repo.git bundle verify vault/proj-2026-08-13.bundle
vault/proj-2026-08-13.bundle is okay
The bundle contains these 4 refs:
a3bfdce1... refs/heads/feature/login
2304fd03... refs/heads/main
fb32611e... refs/tags/v1.0
2304fd03... HEAD
The bundle records a complete history.
The bundle uses this hash algorithm: sha1
```

7. That bundle is one ordinary file. You can copy it anywhere, put it on any storage, and it needs no git server to be useful.
8. `dangling commit` in the `fsck` output is normal, not an error. It means an object exists that no ref points at any more, which happens after a rebase or an amended commit.
9. Fourth, and this is the step almost everyone skips, the restore drill:

```
$ git clone vault/proj-2026-08-13.bundle restore
$ git -C restore log --oneline | head -3
2304fd0 Add app.py
20b8927 First commit: add README
$ git -C restore rev-list --count --all
3
```

10. It restored. Now you know. A backup you have never restored is not a backup; it is a hope.

40.13.4 PLAIN – what is really happening inside

1. Git keeps an object only while something can reach it. Refs are the roots. Anything reachable from a ref is kept; anything not is eventually removed by garbage collection.
2. That single rule explains the whole backup problem.
3. When you clone, git asks for the refs it wants and receives everything reachable from them. Objects reachable only from refs you did not ask for simply do not come.

4. A plain clone asks for `refs/heads/*` and tags. So `refs/stash`, `refs/notes/*` and any custom ref namespace are left behind.
5. A mirror asks for `refs/*`, so it gets all of them.
6. Neither asks for the reflog, because the reflog is not a ref. It is a local log file under `.git/logs`, private to one repository, recording where each ref used to point.
7. That matters because the reflog is what saves you from your own mistakes. It is the thing that lets you recover a commit after a bad reset. It is also the thing no backup of the server can give you.
8. Now the deletion problem. Suppose somebody force-pushes over `main`, discarding fifty commits. On the server the old commits become unreachable.
9. Your hourly mirror runs, sees `main` moved, moves its own `main`, and now the mirror is unreachable from those commits too. Run garbage collection on the mirror and they are gone for good.
10. A dated bundle written before the force-push is unaffected, because it is a frozen file, not a live replica. This is the reason bundles exist in a backup plan alongside mirrors.
11. What a bundle actually is: a packfile with a small header listing ref names and, for an incremental bundle, the commits the receiver must already have. `git bundle verify` checks both the pack and those prerequisites.
12. And a bundle is a first-class remote. `git clone file.bundle`, `git fetch file.bundle`, and `git ls-remote file.bundle` all work, which is why the restore drill above is one ordinary clone.

40.13.5 TECHNICAL - the engineer's version

1. Coverage table. This is the section in one picture.

Item	Clone	Mirror	Bundle –all
Branch commits	yes	yes	yes
Tags	yes	yes	yes
refs/stash	no	yes	yes
refs/notes	no	yes	yes
Custom ref namespaces	no	yes	yes
Reflog	no	no	no
Hooks	no	no	no
Repository config	no	no	no
Uncommitted work	no	no	no
Issues, PRs, CI, releases	no	no	no
Survives a force-push	no	no	yes, if dated

2. Incremental bundles work and are small. A real example: the full bundle was 1,497 bytes and the bundle of everything since a tag was 703 bytes.

```
$ git bundle create incr.bundle main --not backup-2026-08-13
$ git bundle verify incr.bundle
incr.bundle is okay
The bundle contains this ref:
dbcd961d... refs/heads/main
The bundle requires this ref:
141f4579...
```

3. "Requires this ref" is the prerequisite. Applying it to a repository that already has that commit works; applying it to an empty one fails cleanly rather than silently producing a broken repository.

```
$ git fetch /tmp/bk/incr.bundle main:restored
$ git log --oneline restored
dbcd961 Fourth
9cbb377 Third
```

```
141f457 Second
69199b0 First
```

4. A backup procedure that actually holds up. Run it from a machine that is not the git server.

```
# once
git clone --mirror git@host:org/proj.git /srv/bk/proj.git

# every hour: refresh the replica
git -C /srv/bk/proj.git remote update --prune

# every night: dated, immutable snapshot
D=$(date +%F)
git -C /srv/bk/proj.git fsck --full --strict
git -C /srv/bk/proj.git bundle create \
    /srv/vault/proj-$D.bundle --all
git -C /srv/bk/proj.git show-ref > /srv/vault/proj-$D.refs
git -C /srv/bk/proj.git bundle verify \
    /srv/vault/proj-$D.bundle
```

5. Then copy `/srv/vault` off-site, keep snapshots for a stated retention period, and delete older ones on a schedule you wrote down.
6. Why each step is there:

Step	Protects against
Mirror refresh	server loss
fsck –full	silent corruption
Dated bundle	force-push, deletion
show-ref file	knowing what was there
bundle verify	an unreadable backup
Off-site copy	site-wide loss

7. The `show-ref` file is cheap and disproportionately useful. When somebody asks "what was `main` pointing at on the ninth?", it answers in one line without unpacking anything.
8. The 3-2-1 convention is worth applying here: three copies, on two different kinds of storage, one of them off-site. It is a convention from the photography world, not a standard, but it is sound.
9. Verification is not optional and is cheap in git, because every object is named by the hash of its own contents. `git fsck --full --strict` recomputes those hashes and reports any mismatch. No other backup system gets this for free.
10. Restore drills should be scheduled, not improvised. Once a quarter, clone the newest bundle into a scratch directory, compare `git rev-parse main` with the recorded `.refs` file, and delete it.
11. Now the part git cannot help with. Back up the platform metadata separately, because none of it is in any bundle:

```
gh api --paginate repos/OWNER/REPO/issues?state=all \
    > vault/issues-$D.json
gh api --paginate repos/OWNER/REPO/pulls?state=all \
    > vault/pulls-$D.json
gh api repos/OWNER/REPO/branches/main/protection \
    > vault/protection-$D.json
gh release list --repo OWNER/REPO \
    > vault/releases-$D.txt
```

12. Secrets cannot be exported at all by design. Keep the authoritative copy in a password manager or a secrets service, and treat the platform's copy as a deployment detail, not as storage.

13. Settings are best kept as code. Branch protection, teams and repository options can be declared in a configuration tool and re-applied to a fresh repository in minutes, which is a better recovery story than a JSON dump nobody knows how to replay.
14. Encryption note: a bundle is not encrypted and not signed. If your history is confidential, encrypt the file before it leaves your control, and record the checksum separately so you can prove the file is intact.
15. Retention worth defaulting to, in the absence of a policy: hourly mirror, nightly bundle kept 30 days, weekly bundle kept 12 weeks, monthly bundle kept 12 months. Adjust for how much you would pay to recover a given day.

40.13.6 WORDS - remember these

1. Reachability — an object is kept if a ref can reach it — the rule that decides what a clone transfers and what garbage collection removes.
2. Dangling object — nothing points at it any more — an object with no ref path to it, reported by `git fsck`, normal after rebase or amend.
3. Bundle — a whole repository in one file — a packfile plus a ref header, usable directly as a remote for clone, fetch and ls-remote.
4. Prerequisite — what an incremental bundle assumes you have — commit identifiers listed in the bundle header and checked by `bundle verify`.
5. Replica — always current, always obedient — a mirror that reproduces deletions and force-pushes as faithfully as it reproduces work.
6. Snapshot — frozen and dated — an immutable copy taken at a known time and kept for a stated retention period.
7. Restore drill — proving the backup works — a scheduled rehearsal that clones the newest snapshot and compares refs against a recorded list.
8. fsck — check every object against its own hash — `git fsck --full --strict`, which detects corruption that ordinary commands might not notice.

40.98 Common wrong ideas

1. Wrong: GitHub stores my code differently from my laptop. Right: it stores the identical objects with the identical hashes. A commit is `2304fd03...` in both places, or one of them is corrupt. GitHub's replication and routing are invisible to the protocol.
2. Wrong: a pull request is a git feature. Right: git has no such concept. `git help -a` lists nothing called pull request. A pull request is a row in a forge's database plus two ref names, exposed to git only as read-only `refs/pull/N/head`.
3. Wrong: forking is a git operation. Right: forking is a forge action. Git's word for taking a copy is `clone`. A fork is a server-side repository plus a recorded parent relationship that git never sees.
4. Wrong: if GitHub goes away my history is lost. Right: every clone holds the complete commit history. What would be lost is the meeting point and the metadata: issues, reviews, build results and releases. Back those up separately, because git cannot.
5. Wrong: a git server needs special server software. Right: it needs a bare repository and a way to reach it. `git init --bare` plus an SSH account with `git-shell` is a complete, secure server.
6. Wrong: a mirror is a backup. Right: a mirror copies deletions and force-pushes too. It protects against the server dying and not against a person making a mistake. Dated bundles protect against the person.
7. Wrong: a clone copies everything in the repository. Right: it copies objects reachable from branches and tags. It leaves behind the reflog, stashes, hooks, local configuration and all uncommitted work.
8. Wrong: `git push` failing means the push did not happen. Right: a failure while reading the reply, such as `send-pack: unexpected disconnect while reading sideband packet`, tells you the answer was lost, not what the server did. Re-query with `git ls-remote` to find out.
9. Wrong: LFS makes big binary files cheap in git. Right: it moves them out of git. The repository holds a 132-byte pointer, and the bytes live on a separate service with separate storage, separate billing and

a separate program that everyone must install.

10. Wrong: a partial or shallow clone is strictly better because it is smaller. Right: it moves cost, it does not remove it. The same `git log -p -300` took 0.197 seconds in a full clone and 106 seconds in a blobless clone, because every missing blob became a network round trip.

40.99 Chapter summary in 20 lines

1. A git server is just another git repository that other machines can reach; there is no special server format and no central authority.
2. The meeting-point copy is normally bare, meaning it has the contents of `.git` at its top level and no working files.
3. `git init --bare proj.git` builds one in about ten milliseconds, and pushing into a non-bare repository's checked-out branch is refused by default.
4. Running the whole loop by hand proves the model: create the bare repository, push, clone elsewhere, commit there, push back, pull into the first copy.
5. A remote is a name for a URL plus a refspec, and a URL is only a path plus a way of reaching it, so `/tmp/lab/srv/proj.git` and `git@host:proj.git` are the same idea.
6. The refspec `+refs/heads/*:refs/remotes/origin/*` says: take every branch, store it under a separate prefix, and allow non-fast-forward updates.
7. Server hooks are real enforcement: `pre-receive` gates the whole push, `update` gates one ref, `post-receive` acts afterwards and cannot refuse.
8. Client hooks are advice, because `--no-verify` skips them, they are not versioned, and a non-executable hook file is silently ignored.
9. GitHub is four layers: hosting for bare repositories, a web interface, identity and permissions, and automation. Founded 2008, bought by Microsoft in 2018 for 7.5 billion United States dollars.
10. The git layer of GitHub is ordinary git; `git ls-remote` against it prints exactly the same shape of output as against a folder in `/tmp`.
11. Pull requests, reviews, branch protection, required checks, CODEOWNERS behaviour, issues, projects, releases, packages, Actions, both APIs and the permission model are all GitHub, not git.
12. The reader's outage proves the boundary: local git kept working, remote git and the whole GitHub API did not, exactly as the local/remote split predicts.
13. `origin/main` reading `0a95cc8` during the outage was a cached photograph, not a live fact, and a failure while reading a reply says nothing about whether the write landed.
14. Self-hosting is real and cheap: SSH plus `git-shell` costs tens of megabytes, Gitea or Forgejo runs in 1 GB, GitLab CE wants 8 to 16 GB.
15. Self-hosting buys control, privacy and no lock-in, and it makes you the operator, responsible for backups, uptime and patches at three in the morning.
16. Migration moves commits perfectly and almost nothing else; issue numbers, cross-references, timestamps, attachments and build history do not survive.
17. Choose the collaboration model by trust: shared repository for a team you trust, forks for strangers; then protect main, or every rule is a suggestion.
18. Git flow, GitHub flow and trunk-based development disagree about exactly one thing: how long a change may live away from the mainline.
19. Large repositories are managed with shallow clones, partial clones, sparse checkout and LFS, and binaries remain git's weak point because they do not delta-compress.
20. A clone is not a backup, a mirror is not a backup, and a backup you have never restored is not a backup either.

41

CHAPTER

KB

CI/CD and What a "Run" Really Is

41.0 What this chapter gives you

1. You will be able to explain what continuous integration is as a practice, and say clearly why it is not the same thing as a tool that runs tests.
2. You will be able to state the difference between continuous delivery and continuous deployment in one sentence each, without muddling them.
3. You will be able to describe what a runner is, what it has installed when it starts, and why a command that works on your laptop fails there.
4. You will be able to read a complete GitHub Actions workflow file line by line, and translate the same pipeline into GitLab CI.
5. You will be able to name every common trigger, and explain why `pull_request_target` is a security trap while `pull_request` is not.
6. You will be able to explain the single most important idea here: a run is attached to one commit hash, never to a branch name.
7. You will be able to verify continuous integration by run ID against an exact head hash, using real commands, and list six ways a branch-level green tick can lie to you.
8. You will be able to explain what a merge queue is, what problem it solves, and what every `mergeStateStatus` value means and implies.
9. You will be able to lay out a sensible pipeline in order, choose a deployment strategy, and handle secrets without leaking them.
10. You will be able to read a failed run methodically instead of guessing.

41.1 What continuous integration means and the problem it solves

41.1.1 PLAIN – in simple words

1. Several people work on the same program at the same time.
2. Each person has their own copy, and each changes different files.
3. Sooner or later all those changes must be joined into one program.
4. Joining them is called **integration**. It is the moment of truth.
5. If everyone waits weeks before joining, the changes have drifted far apart and joining them is slow, painful and full of surprises.
6. Old teams had a word for this: integration hell. It could take days.
7. **Continuous integration** is the opposite habit. Everyone joins their work into the shared copy very often, at least once a day.
8. Small joins are easy. Big joins are hard. So you make the joins small.
9. But joining often is only safe if you also check often.
10. So every time anyone joins, a machine automatically builds the program and runs the tests, and tells you within minutes if it broke.

11. That habit, plus that automatic check, is continuous integration.
12. Notice that the habit comes first. The machine only enforces the habit.

41.1.2 PLAIN - a picture in your head

1. Five students must write one report together, one chapter each.
2. Plan A: everyone writes alone for three weeks, then meets the night before to glue the chapters together.
3. That night is chaos. Two people defined the same term differently. Three people numbered their figures from one. The introduction promises a section that nobody wrote.
4. Plan B: every evening, each student pastes what they have into the one shared document, and one person reads the whole thing top to bottom.
5. Under plan B a clash is found the same evening it is created, while both students still remember what they meant.
6. Now replace "one person reads it top to bottom" with a machine that reads it every single time anyone pastes anything, in two minutes.
7. That machine is the continuous integration server. The evening paste is the practice.

Where this comparison breaks: a report can be a bit inconsistent and still be readable. A program cannot. One mismatched function name and nothing runs at all. Also, a human reader gets bored and misses things on the tenth read. A test suite does not get bored, but it only checks what somebody wrote a test for, which is always less than everything.

41.1.3 PLAIN - a worked example

1. Take a team of five people. Each writes about 200 changed lines a day.
2. Under plan A they integrate once every three weeks, so 15 working days.
3. Lines waiting to be joined: 5 people x 200 lines x 15 days = 15,000 lines.
4. Under plan B they integrate once a day: 5 x 200 x 1 = 1,000 lines.
5. The pain of a merge does not grow in a straight line with size. Two changes can clash, and the number of possible clashing pairs grows roughly with the square of the number of changes.
6. So a batch fifteen times bigger can feel far worse than fifteen times worse.

Integration gap	Lines in flight	Typical feel
Every 3 weeks	15,000	Days of merge work
Every week	5,000	Hours of merge work
Every day	1,000	Minutes, usually clean
Every commit	~200	Almost always clean

7. The numbers in that table are illustrative, not measured. The shape is what matters: smaller batches, cheaper joins.

41.1.4 PLAIN - what is really happening inside

1. You finish a small piece of work and commit it locally.
2. You push it to the shared server. The push is just data transfer.
3. The server notices that a branch moved and fires an event.
4. That event is delivered to whatever is listening. On a hosted service it goes to an internal queue. For an external tool it goes out as a **webhook**, which is a small message posted to a web address you registered.
5. The listener picks a free machine, or starts a fresh one.
6. That machine downloads the exact commit that was pushed. Not the branch. The commit.
7. It installs the tools the project needs, builds the program, and runs the tests.
8. It records success or failure, and posts that result back, attached to the same commit hash it downloaded.

9. The web page you look at then draws a tick or a cross next to that commit.
10. The whole loop is usually two to twenty minutes. Under ten minutes is the target most teams aim at, because longer than that and people stop waiting for the answer and start guessing.

41.1.5 TECHNICAL - the engineer's version

1. The phrase "continuous integration" was coined by Grady Booch in his 1991 book Object-Oriented Design With Applications, where it described the habit of integrating incrementally rather than in a big bang.
2. Extreme Programming, developed by Kent Beck on the Chrysler C3 project from 1996 and set out in Extreme Programming Explained in 1999, made it one of its named practices, with the rule that the team integrates and builds many times a day.
3. Martin Fowler and Matthew Foemmel published the article Continuous Integration in 2000, revised by Fowler in 2006 and again in 2024. That article is where most of the working definition comes from.
4. CruiseControl, written at ThoughtWorks and first released on 30 March 2001, was the first widely used continuous integration server. It polled the version control system on a timer, because webhooks did not exist yet.
5. Hudson was started by Kohsuke Kawaguchi at Sun Microsystems in 2004 and first released in 2005. After a dispute with Oracle, a rename vote was called on 11 January 2011 and approved on 29 January 2011. The renamed project is Jenkins.

Year	Thing	Who
1991	Term coined	Grady Booch
1999	XP names the practice	Kent Beck
2000	The defining article	Fowler and Foemmel
2001	CruiseControl released	ThoughtWorks
2011	Hudson renamed Jenkins	Community vote
2019	GitHub Actions CI GA	GitHub

6. Travis CI launched in 2011 and popularized the idea of a build configuration file living inside the repository. CircleCI launched the same year. GitLab CI began as a separate application in 2012 and was folded into GitLab itself in version 8.0 in September 2015.
7. GitHub Actions was announced as a general automation product in October 2018. Its continuous integration capability was announced in beta on 8 August 2019 and became generally available on 13 November 2019.
8. The practice, in its strict form, has three testable properties: everyone merges to the mainline at least daily, every merge triggers an automated build and test, and a broken mainline is fixed immediately and takes priority over new work.
9. Say this plainly: a team running Jenkins on a branch that merges to main once a quarter is not doing continuous integration. It owns a continuous integration tool. The practice and the tool are different things.
10. Observing tools: `gh run list`, `gh run view`, `git log --oneline --since='1 day ago' origin/main` to see integration frequency, and any dashboard that plots merges to trunk per day.

41.1.6 WORDS - remember these

1. Integration — joining everyone's work into one copy — merging feature branches into the mainline and producing a buildable tree.
2. Integration hell — the painful big join at the end — the superlinear cost of merging long-lived divergent branches.
3. Continuous integration — join often, check automatically every time — the practice of at-least-daily mainline merges verified by an automated build.

4. Webhook — a message the server sends when something happens — an HTTP POST to a registered endpoint carrying a JSON event payload.
5. Mainline or trunk — the one branch everybody joins into — usually `main`, the integration target for all short-lived branches.
6. Build — turning source into a runnable thing — compile, link, package, including dependency resolution.

41.2 Continuous delivery and continuous deployment

41.2.1 PLAIN - in simple words

1. Continuous integration answers one question: does the joined code work.
2. There is a second question: could we give it to users right now.
3. **Continuous delivery** means the answer is always yes. Every change that passes the checks is packaged, tested in a realistic place, and left in a state where one button press would put it in front of users.
4. A human still decides when to press the button.
5. **Continuous deployment** goes one step further. There is no button. If the checks pass, the change goes live automatically.
6. So the difference is exactly one thing: whether a human approves the last step.
7. Continuous delivery is a capability. Continuous deployment is a policy you can only adopt once you have that capability.
8. Both are abbreviated CD, which is why people confuse them constantly.
9. If somebody says "we do CI/CD", you have learned almost nothing. Ask which D they mean, and ask what the last manual step is.

41.2.2 PLAIN - a picture in your head

1. Think of a bakery that supplies a shop.
2. Continuous integration is the taste test after mixing. Is the dough right.
3. Continuous delivery is: the bread is baked, wrapped, labelled, loaded onto the van, and the van is parked outside with the engine running.
4. Any moment the manager says go, it leaves. Nothing more to prepare.
5. Continuous deployment is: the van leaves the moment the loading is finished. No manager, no decision.
6. Notice that a bakery can be excellent at delivery and still choose to send the van only twice a day, for business reasons.
7. That is a real and respectable choice. The point of delivery is that the delay is a decision, not a limitation.

Where this comparison breaks: bread cannot be un-delivered. Software can be rolled back, sometimes. Also the van either goes or does not, whereas software can go to one percent of users first, which has no bakery equivalent.

41.2.3 PLAIN - a worked example

1. Here is one change moving through a delivery pipeline, with times.

Stage	What happens	Time
Commit stage	Lint, build, unit tests	4 min
Package	Build image, push it	2 min
Staging deploy	Install to staging	1 min
Acceptance tests	End-to-end suite	9 min
Ready to release	Waiting for approval	0 min

Stage	What happens	Time
Production deploy	Rolling update	3 min

2. Total machine time is 19 minutes. Under continuous delivery the change sits at "ready to release" until a person approves it.
3. Under continuous deployment, the row "Ready to release" is deleted and the change is live 19 minutes after the commit.
4. The test suites did not change. The pipeline did not change. Only the gate changed.

41.2.4 PLAIN - what is really happening inside

1. The pipeline is a chain of stages. Each stage takes the output of the one before, not a fresh copy of the source.
2. That is the key rule: you build the artifact once, and the same bytes are what gets tested and what gets deployed.
3. If you rebuild before deploying, you have tested something other than what you shipped, and the test result no longer applies.
4. Early stages are fast and cheap and run on everything.
5. Later stages are slow and expensive and only run on things that passed.
6. Each stage either passes the artifact forward or stops it dead.
7. The pipeline itself records which commit each artifact came from, so you can always answer "what is running in production, exactly".
8. Continuous deployment simply removes the human gate from the last hop, and compensates with automatic checks after the deploy: health checks, error rate alarms, and automatic rollback.

41.2.5 TECHNICAL - the engineer's version

1. The deployment pipeline as a named pattern comes from Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation by Jez Humble and David Farley, published in 2010.
2. Its core rules are: build binaries once, deploy the same way to every environment, smoke-test every deployment, and stop the line if any stage fails.
3. Continuous delivery: every change that passes automated checks is a release candidate that could be deployed on demand. Continuous deployment: every such change is deployed without human intervention.
4. The DORA research programme, published in the State of DevOps reports from 2014 and in the 2018 book Accelerate by Nicole Forsgren, Jez Humble and Gene Kim, measures four things.

DORA metric	What it measures
Deployment frequency	How often you ship
Lead time for changes	Commit to production time
Change failure rate	Percent of deploys causing harm
Failed deployment recovery	Time to restore service

5. The consistent finding across those reports is that speed and stability move together rather than trading off. That is a statistical association in survey data, not a controlled experiment. Treat it as strong evidence, not proof.
6. There is genuine expert disagreement about whether continuous deployment is right for every domain. Web services with cheap rollback lean yes. Firmware, medical devices and payment cores lean no, because the cost of a bad release is not symmetric with the cost of a slow one.

7. The honest version: most organizations that say "we do CD" mean continuous delivery to a staging environment and manual promotion to production. That is a fine place to be. It is just not continuous deployment.

41.2.6 WORDS - remember these

1. Continuous delivery — always ready to ship — every passing change is a deployable release candidate, promoted on demand.
2. Continuous deployment — ships by itself — every passing change is deployed to production without manual approval.
3. Deployment pipeline — the chain of stages a change walks — an automated implementation of the path from commit to release, with binary reuse.
4. Release candidate — a version that could go live — an immutable artifact that has passed all preceding pipeline stages.
5. Promotion — moving an artifact to the next environment — deploying the same artifact to a higher environment without rebuilding.
6. Lead time for changes — how long from writing to live — the DORA metric measuring commit timestamp to production deploy timestamp.

41.3 What a runner actually is

41.3.1 PLAIN - in simple words

1. Something has to actually execute your build. That something is a computer.
2. In continuous integration that computer is called a **runner**, or an agent, or an executor, depending on the product.
3. It can be a whole machine, a virtual machine, or a container. From your side it does not matter much: it is a place with a filesystem and a shell.
4. When work arrives, the runner downloads your code at one exact commit and runs your steps one after another, in order, in a shell.
5. It reports the exit code of each step. Zero means success. Anything else means failure, and by default the job stops there.
6. Two kinds exist. **Hosted** runners are supplied by the service, created fresh for your job, and destroyed afterwards. **Self-hosted** runners are machines you own, running an agent program that asks for work.
7. The most important fact about a fresh hosted runner: it contains only what the image ships with. Nothing of yours. No files, no settings, no tools you installed last year and forgot about.
8. That is exactly why builds fail on the runner and pass on your laptop.

41.3.2 PLAIN - a picture in your head

1. Imagine a hotel kitchen you can rent by the minute.
2. It is spotless. Every surface wiped. Standard ovens, standard knives.
3. Whatever you want to cook, you must bring the recipe and the ingredients, or ask the hotel for them at the door.
4. Your own kitchen at home has a jar of your grandmother's spice mix on the top shelf. You use it without thinking. It is not in your recipe.
5. In the hotel kitchen, the dish comes out wrong, and you cannot see why, because the missing thing was never written down anywhere.
6. When your work is finished, the hotel demolishes the kitchen and builds a new one for the next customer. Nothing you left behind survives.

Where this comparison breaks: a real kitchen is never truly identical twice. Runner images are updated every week or two, so tool versions drift under you. And a self-hosted runner is more like your own kitchen: it does keep the spice jar, which is convenient right up until it becomes the bug.

41.3.3 PLAIN - a worked example

1. This step works perfectly on the reader's macOS laptop and fails on a Linux runner.

```
python script.py --config Config.json
```

2. On macOS the default filesystem is APFS, set up case-insensitive. The file on disk is `config.json`, and `Config.json` finds it anyway.
3. On the Linux runner the filesystem is case-sensitive. `Config.json` does not exist. The script exits non-zero. The job fails.
4. The error message is `FileNotFoundError`, which sounds like the file was not committed, so the first hour is spent looking in the wrong place.
5. Here are the usual suspects, all of the same shape.

Works locally because	Runner reality
Case-insensitive filesystem	Case-sensitive filesystem
Tool installed years ago	Only image defaults present
`.env` file on disk	Untracked files do not exist
Local time zone	Runner is UTC
Logged in to a registry	No credentials at all
Dependencies already cached	Cold download every time

41.3.4 PLAIN - what is really happening inside

1. A job goes through the same seven phases every time. Learn them and you can place any failure.

```
1 QUEUE     job waits for a free runner
2 ALLOCATE  a runner is chosen or a VM is created
3 PROVISION boot image, start agent, set environment
4 CHECKOUT  fetch the repository at one exact commit
5 RESTORE   pull caches: packages, compiler output
6 STEPS     run each step in order, stop on failure
7 UPLOAD    save artifacts and logs
8 CLEAN     destroy the machine or wipe the workspace
```

2. Phase 1 costs nothing but wall-clock time. If jobs sit here, you have hit a concurrency limit, not a bug.
3. Phase 3 is where the operating system image is decided. That image is a product of the service and it changes.
4. Phase 4 usually does a shallow fetch of depth one: only the one commit, not the whole history, because history is large and rarely needed.
5. Phase 5 restores a cache keyed on something like the hash of your lock file. A cache miss is slow but correct. A wrong cache hit is fast and wrong, which is worse.
6. Phase 6 runs each step as a separate shell invocation. Variables set with plain `export` in one step do not survive to the next step, because the shell exited.
7. Phase 8 is why "it left a file behind last time" is never an explanation on a hosted runner. There is no last time.

41.3.5 TECHNICAL - the engineer's version

1. GitHub-hosted runners are ephemeral virtual machines in Microsoft Azure, with a fresh instance per job, running the open-source `actions/runner` agent. Figures below were taken from GitHub documentation in August 2026 and change over time.

Label	vCPU	RAM	Disk
`ubuntu-latest` public	4	16 GB	14 GB
`ubuntu-slim` public	1	5 GB	14 GB
`windows-latest` public	4	16 GB	14 GB
`macos-latest` (M1)	3	7 GB	14 GB

2. For private repositories the standard Linux and Windows runners are smaller, at 2 vCPU and 8 GB RAM as of August 2026. Larger runners are a paid option.
3. Hard limits for GitHub Actions, as documented in August 2026: a job may run up to 6 hours on a hosted runner and up to 5 days on a self-hosted runner; a workflow run is cancelled after 35 days; a matrix can generate at most 256 jobs per run; a self-hosted job may sit queued for 24 hours.
4. Concurrent job limits by plan: 20 on Free, 40 on Pro, 60 on Team and 500 on Enterprise. The `GITHUB_TOKEN` is rate limited to 1,000 API requests per hour per repository.
5. GitLab uses a separate `gitlab-runner` binary with pluggable executors: `docker`, `shell`, `kubernetes`, `docker-machine` and others. Jenkins calls the same idea an agent, historically a "slave", connected by JNLP or SSH.
6. Self-hosted runners on a public repository are a documented security hazard: a fork can propose a workflow change that executes on your hardware. GitHub documents this and recommends self-hosted runners only for private repositories, or with ephemeral, isolated, single-use runners.
7. Runner images are built in the open. The `actions/runner-images` repository publishes the exact contents of each image and a changelog per release, which is the correct place to check whether a tool version moved under you.
8. Observing tools: `runs-on` in the workflow file, the "Set up job" log group which prints the image version and the resolved runner name, `uname -a`, `df -h`, `nproc` and `env | sort` as diagnostic steps.

41.3.6 WORDS - remember these

1. Runner — the machine that executes your build — an ephemeral or persistent host running an agent that claims and executes jobs.
2. Hosted runner — supplied and destroyed by the service — a per-job ephemeral VM from a published image, with no state between jobs.
3. Self-hosted runner — your own machine doing the work — a long-lived or ephemeral host running the agent, with persistent state unless wiped.
4. Job — one unit of work on one runner — a named collection of steps sharing a workspace, filesystem and environment.
5. Step — one command or one action inside a job — a separate shell invocation or a container/JavaScript action execution.
6. Workspace — the folder your code is checked out into — the working directory, exposed as `GITHUB_WORKSPACE` or `CI_PROJECT_DIR`.
7. Exit code — the number a command returns — 0 for success, non-zero for failure; the runner uses it to decide whether to continue.

41.4 The workflow file

41.4.1 PLAIN - in simple words

1. Somewhere you must write down what the machine should do. That written description is the **workflow file**.
2. It is a plain text file in YAML, a format that uses indentation instead of brackets to show what belongs to what.
3. On GitHub it lives in the folder `.github/workflows/` inside your repository, and any file there ending in `.yml` or `.yaml` is picked up.

4. On GitLab it is a single file called `.gitlab-ci.yml` at the top of the repository.
5. It sits inside the repository, next to the code, in the same history. That is the important part.
6. Because it is in the history, the workflow that ran for an old commit is the workflow as it existed at that commit. Not today's version.
7. So changing your tests and changing how they run is one change, one review, one commit. They can never drift apart.

41.4.2 PLAIN - a picture in your head

1. Think of a recipe card taped inside the lid of the box that holds the ingredients.
2. If you change the ingredients, you open the box, so you see the recipe and change it too.
3. Compare with a recipe pinned on a noticeboard in another room. It slowly becomes a description of a dish nobody makes any more.
4. A configuration stored in a web interface on the build server is the noticeboard. A workflow file in the repository is the card inside the lid.
5. And if you open a box from three years ago, the card inside tells you exactly how that batch was made.

Where this comparison breaks: the recipe card is not fully self-contained. It says "use action X at version 4", and version 4 is fetched fresh at run time from somewhere else. So an old commit can still build differently today. Section 41.13 explains how pinning fixes that.

41.4.3 PLAIN - a worked example

1. Here is a complete, working GitHub Actions workflow. Every line is explained underneath.

```
name: CI

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

permissions:
  contents: read

concurrency:
  group: ci-${{ github.ref }}
  cancel-in-progress: true

env:
  CI: "true"

jobs:
  lint:
    runs-on: ubuntu-latest
    timeout-minutes: 10
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: "22"
          cache: npm
      - run: npm ci
      - name: Static checks
        run: npm run lint

  test:
```

```
    needs: [lint]
    runs-on: ${{ matrix.os }}
    timeout-minutes: 20
    strategy:
      fail-fast: false
      matrix:
        os: [ubuntu-latest, macos-latest]
        node: ["20", "22"]
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}
          cache: npm
      - run: npm ci
      - name: Unit tests
        run: npm test
        env:
          TZ: UTC
      - name: Save the report
        if: always()
        uses: actions/upload-artifact@v4
        with:
          name: junit-${{ matrix.os }}-${{ matrix.node }}
          path: reports/junit.xml
```

2. `name` is the label shown in the user interface. Cosmetic only.
3. `on` lists the events that start a run. Here: a push to `main`, and any pull request whose target is `main`.
4. `permissions` sets what the automatic token may do. `contents: read` means the job can read the repository and nothing else.
5. `concurrency` groups runs. Two runs in the same group cannot overlap, and `cancel-in-progress: true` kills the older one. This stops five pushes in a minute from starting five full pipelines.
6. `env` at the top sets environment variables for every job.
7. `jobs` holds one or more jobs. `lint` and `test` are job IDs, chosen by you.
8. `runs-on` picks the runner label. `ubuntu-latest` means a hosted Linux VM.
9. `timeout-minutes` kills a job that hangs. Without it a stuck job burns until the six-hour limit.
10. `steps` is an ordered list. Each item is either `uses` or `run`.
11. `uses` runs a prepackaged **action**, here `actions/checkout` at major version 4, which clones the repository at the right commit.
12. `with` passes named inputs to that action.
13. `run` executes a shell command on the runner.
14. `name` on a step is the label in the log. Give every non-obvious step one, because that label is what you will search for when it fails.
15. `needs: [lint]` makes `test` wait for `lint` to succeed. Without `needs`, jobs run in parallel.
16. `strategy.matrix` expands one job into several. Two operating systems by two Node versions gives four jobs.
17. `fail-fast: false` says: if one matrix cell fails, let the others finish. The default, `true`, cancels the siblings, which is faster but tells you less.
18. `if: always()` on the upload step means "run this even if a previous step failed", which is how you get the test report from a failing build.
19. `${{ ... }}` is an expression. It is evaluated by the service before the step runs, and substituted as text.

41.4.4 PLAIN - what is really happening inside

1. Here is the same pipeline written for GitLab CI, in `.gitlab-ci.yml`.

```
stages: [lint, test]

default:
  image: node:22-bookworm
  cache:
    key:
      files: [package-lock.json]
    paths: [.npm/]

variables:
  CI: "true"

lint:
  stage: lint
  script:
    - npm ci --cache .npm --prefer-offline
    - npm run lint

unit-test:
  stage: test
  needs: [lint]
  parallel:
    matrix:
      - NODE_TAG: ["20", "22"]
  image: node:${NODE_TAG}-bookworm
  script:
    - npm ci --cache .npm --prefer-offline
    - npm test
  artifacts:
    when: always
    reports:
      junit: reports/junit.xml
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

2. The differences are worth naming, because they are differences of model, not of syntax.
3. GitLab has a first-class concept of **stages**: named phases that run in order, with all jobs in a stage running together. GitHub has no stages, only `needs` edges between jobs, which form a graph.
4. GitLab jobs run inside a container image by default, set with `image`. GitHub jobs run directly on the runner VM, and containers are opt-in.
5. GitLab has no `uses`. Reuse comes from `include`, `extends` and YAML anchors, plus CI/CD components. GitHub's marketplace of actions has no direct GitLab equivalent.
6. GitLab checks out the code for you automatically. GitHub requires an explicit `actions/checkout` step, and forgetting it is a classic first-day mistake.
7. GitLab's `rules` replaces GitHub's `on` plus `if`, combining trigger and condition into one list evaluated top to bottom.
8. Both put the file in the repository, and both therefore get the same benefit: the pipeline definition is versioned, reviewable and bisectable.

41.4.5 TECHNICAL - the engineer's version

1. GitHub reads every file matching `.github/workflows/*.yml` and `.github/workflows/*.yaml` on the branch that the event applies to. This is a documented behaviour, not a convention.

2. For `push` events the workflow file used is the one at the pushed commit. For `pull_request` events it is the one at the pull request's head. For `pull_request_target` and `schedule` it is the one on the base or default branch. That asymmetry is the root of section 41.5's security discussion.
3. Available contexts inside `${{ }}` include `github`, `env`, `vars`, `secrets`, `job`, `steps`, `runner`, `strategy`, `matrix`, `needs` and `inputs`. Context values are substituted before the shell sees them, which is why interpolating untrusted text directly into a `run` block is a script injection hole. Pass it through `env:` instead.
4. Each `run` block is written to a temporary script and executed with the default shell: `bash -e` on Linux and macOS, `pwsh` on Windows. Because `-e` is set, the first failing command ends the step. Add `set -o pipefail` if you use pipes, since a pipeline's exit code is the last command's.
5. YAML has a real trap here. In YAML 1.1, the bare word `on` is a boolean true, so strict linters flag `on:` as the key `true`. GitHub's parser special-cases it. Quoting as `"on":` also works. This is an implementation detail worth knowing when a linter shouts at a file that runs fine.
6. Reuse mechanisms: composite actions (steps bundled in an `action.yml`), reusable workflows called with `uses` at job level and triggered by `workflow_call`, and Docker container actions. Reusable workflows can be nested up to four levels deep.
7. GitLab equivalents: `include:local`, `include:project`, `include:remote`, `extends`, and CI/CD components introduced in GitLab 16.6 in late 2023.
8. Because the file is versioned, `git log -p .github/workflows/` answers "when did this check start being required" and `git bisect` can walk a pipeline regression the same way it walks a code regression.
9. A practical consequence the reader met directly: a GitHub personal access token needs the `workflow` scope to push a commit that adds or edits a file under `.github/workflows/`. Without it the push is refused, even though the same token can push ordinary code. The workflow file is treated as more dangerous than code, because it is.

41.4.6 WORDS - remember these

1. Workflow file — the written plan for the machine — a YAML document defining triggers, jobs and steps, stored in the repository.
2. YAML — a text format that uses indentation — a data serialization language; GitHub Actions and GitLab CI both use it for pipeline definitions.
3. Action — a reusable prepackaged step — a JavaScript, Docker or composite unit referenced by `uses: owner/repo@ref`.
4. Matrix — one job definition expanded into many — a cross product of variable values generating up to 256 jobs per run.
5. Stage — a named phase in GitLab — an ordered grouping of jobs; GitHub models the same idea as `needs` dependencies instead.
6. Context — the data available to expressions — named objects such as `github`, `matrix` and `secrets` resolved before the step runs.
7. Script injection — untrusted text becoming code — interpolating attacker controlled values into a `run` block, mitigated by passing through `env`.

41.5 Triggers: what starts a run

41.5.1 PLAIN - in simple words

1. A run never starts on its own. Something has to ask for it.
2. The thing that happens is called an **event**. The rule in your file that listens for it is called a **trigger**.
3. The commonest event is a **push**: you send new commits to the shared server.
4. The next commonest is a **pull request**: you propose joining one branch into another.
5. A clock can be the event: run every night at two in the morning.
6. A person can be the event: somebody clicks a "Run workflow" button.

7. Another program can be the event: an outside system sends a message asking for a run.
8. Creating a **tag**, which is a permanent name pinned to one commit, can be the event.
9. Publishing a **release**, which is a tag plus notes and downloadable files, can be the event.
10. Every event carries facts with it: which repository, which branch, which commit, and who caused it.
11. Your workflow file lists the events it cares about. Every other event is ignored by that file.
12. You can narrow an event further: only on this branch, only when these files changed.
13. Narrowing matters because every run costs minutes and money, and most runs on most changes are wasted work.

41.5.2 PLAIN - a picture in your head

1. Picture a workshop with a row of bells along one wall.
2. Bell one rings when a delivery van drops parts at the door. That is `push`.
3. Bell two rings when somebody posts a proposed change to the plans. That is `pull_request`.
4. Bell three is wired to a timer and rings itself at 2 a.m. every night. That is `schedule`.
5. Bell four has a rope hanging down, and any worker may pull it whenever they like. That is `workflow_dispatch`.
6. Bell five is a phone line from the factory next door, so they can call you. That is `repository_dispatch`.
7. Beside each bell hangs a small card saying which deliveries are worth getting up for.
8. One card says "steel only, ignore paint". That is a path filter.
9. Another says "only deliveries addressed to the main workshop". That is a branch filter.

Where this comparison breaks: bells in a workshop ring one at a time. Real events overlap. A single push to a branch that already has an open pull request rings two bells and can start two nearly identical runs on nearly the same code. Also a bell carries no information, while an event carries a whole document describing exactly what happened, which your job can read.

41.5.3 PLAIN - a worked example

1. Here is one `on:` block using most of the common triggers at once.

```
on:
  push:
    branches:
      - main
      - 'release/**'
    tags:
      - 'v*.*.*'
    paths:
      - 'src/**'
      - 'package-lock.json'
  pull_request:
    types: [opened, synchronize, reopened, ready_for_review]
    branches: [main]
  schedule:
    - cron: '17 2 * * *'
  workflow_dispatch:
    inputs:
      environment:
        description: 'Where to deploy'
        type: choice
        options: [staging, production]
        default: staging
      dry_run:
        type: boolean
        default: true
  repository_dispatch:
```

```
    types: [deploy-request]
  release:
    types: [published]
```

2. `push.branches` lists branch name patterns. `release/**` matches `release/2026-08` and `release/hotfix/db`.
3. `push.tags` lists tag patterns. `v*.*.*` matches `v2.4.1`. If both `branches` and `tags` are present, a match on either one starts a run.
4. `push.paths` says: only start if at least one changed file matches. A commit that only edits `README.md` starts nothing.
5. `pull_request.types` lists the sub-kinds of the event. `synchronize` means "the pull request branch got a new push", which is the one people forget.
6. `ready_for_review` fires when a draft pull request stops being a draft. It is not in the default list, so you must ask for it.
7. `schedule.cron` is five fields: minute, hour, day of month, month, day of week. `17 2 * * *` is 02:17 every day, in UTC always.
8. The 17 is deliberate. Everyone writes `0 * * * *`, so the top of the hour is the busiest queue on the service. Pick an odd minute.
9. `workflow_dispatch.inputs` builds a small form in the web page and also accepts the same values from the command line.
10. `repository_dispatch.types` filters on a label chosen by the outside caller, so one repository can serve several kinds of external request.
11. `release.types: [published]` fires when a release is published, which is the usual moment to build and upload artifacts.

Trigger	Typical use
`push`	Test the mainline
`pull_request`	Test a proposal
`schedule`	Nightly, slow scans
`workflow_dispatch`	Manual deploy
`repository_dispatch`	Called by another system
`release`	Publish artifacts

41.5.4 PLAIN - what is really happening inside

1. Now the confusing pair. `pull_request` and `pull_request_target` look almost the same and behave very differently.
2. When somebody outside your team proposes a change from their own copy, that code is a stranger's code. You have not read it yet.
3. A run started by `pull_request` does check out the stranger's code, but it runs with almost no power.
4. It gets a read-only token, and it gets none of your secrets.
5. So even if the stranger's code is hostile, it can make a mess inside its own throwaway machine and steal nothing.
6. That is the safe default. It is also why a first-time contributor often sees "waiting for a maintainer to approve running workflows".
7. But it is inconvenient. A run with no secrets cannot post a comment, cannot upload to your storage, and cannot add a label.
8. So a second trigger exists: `pull_request_target`.
9. `pull_request_target` reads the workflow file from your own base branch, not from the stranger's branch.

10. Because the file is yours and was reviewed by you, the service trusts it and hands it a write token and your secrets.
11. Here is the trap, and it is a sharp one. The workflow file is yours. The code being proposed is still theirs.
12. If your trusted workflow checks out the pull request's head and then executes anything from it, you have just handed a stranger your keys.
13. "Executes anything" is wider than it sounds. It includes `npm test`. It also includes `npm install`, because a dependency can run an install script.
14. It includes a build tool that reads a config file naming a plugin, a `Makefile`, a test fixture, or a linter with a custom rule.
15. GitHub's own security team named this family of bug **pwn requests** in an article published in August 2020.
16. The rule is one sentence. With `pull_request_target`, do not check out the head; and if you truly must, do not execute one byte of it.

```
               pull_request        pull_request_target
               ------------        -------------------
workflow file   PR head branch      base branch
checked out     PR merge result     base branch by default
token           read-only (fork)    read and write
secrets         none (fork)         all of them
safe by         having no power     you not running their code
```

41.5.5 TECHNICAL – the engineer's version

1. GitHub Actions documents more than thirty webhook-backed events plus the non-webhook triggers `schedule`, `workflow_dispatch` and `workflow_call`.

Trigger	Fires when	Note
`push`	commits or tags land	branch, tag, path filters
`pull_request`	PR opened, updated	forks get no secrets
`pull_request_target`	same, base context	write token, dangerous
`schedule`	cron time in UTC	5 minute minimum
`workflow_dispatch`	manual or API call	up to 10 inputs
`repository_dispatch`	external POST	default branch only
`release`	release published	types include `published`
`merge_group`	merge queue asks	`checks_requested`
`workflow_run`	another run ends	base branch file

2. The default activity types for `pull_request` are `opened`, `synchronize` and `reopened`. `synchronize` fires on every new push to the head branch.
3. `schedule` uses five-field POSIX cron, always evaluated in UTC, never in local time. The shortest documented interval is every 5 minutes.
4. Scheduled runs are queued, not guaranteed punctual. Delays of many minutes at the top of the hour are normal and documented.
5. In a public repository, a scheduled workflow is disabled automatically after 60 days with no repository activity. This is documented behaviour and a very common cause of "our nightly build silently stopped".
6. `workflow_dispatch` accepts at most 10 top-level inputs, of types `string`, `choice`, `boolean` and `environment`, with a total payload limit of 65,535 characters. Start it with `gh workflow run` or `POST /repos/{owner}/{repo}/actions/workflows/{id}/dispatches`.

7. `repository_dispatch` is `POST /repos/{owner}/{repo}/dispatches` carrying `event_type` and an optional `client_payload` JSON object, readable as `github.event.client_payload`. It only starts workflows whose file lives on the default branch, which surprises people testing it from a branch.
8. Filter keys are `branches`, `branches-ignore`, `tags`, `tags-ignore`, `paths` and `paths-ignore`. You cannot use a key and its `-ignore` partner for the same event in the same workflow.
9. Glob syntax in these filters: `*` matches any characters except `/`, `**` matches any characters including `/`, `?` matches one character, `+` and `*` act as repeaters, `[]` is a character class, and a leading `!` negates.
10. A documented caveat with real consequences: if a push contains more than 1,000 commits, or GitHub cannot compute the diff because it timed out, path filters are ignored and the workflow runs anyway.
11. The opposite caveat matters more. A path-filtered workflow that does not run produces no check at all, and a required status check that never arrives blocks the pull request forever. Section 41.8 covers the fix.
12. The precise difference between the two pull request triggers:

Aspect	`pull_request`	`pull_request_target`
Workflow file from	PR head	base branch
`GITHUB_REF`	`refs/pull/N/merge`	base branch ref
Fork secrets	none	all
Fork token	read-only	read and write

13. For a fork-originated `pull_request`, the `GITHUB_TOKEN` is issued with read permissions only, and no repository, environment or organization secret is exposed. Public repositories can additionally require maintainer approval before a first-time contributor's workflow runs at all.
14. The dangerous pattern, written out so you can recognize it in review:

```
# DANGEROUS when combined with pull_request_target
- uses: actions/checkout@v4
  with:
    ref: ${{ github.event.pull_request.head.sha }}
- run: npm ci && npm test    # their code, your secrets
```

15. The safe pattern when you genuinely need write access on a fork pull request: split into two workflows. The `pull_request` workflow does the untrusted work with no secrets and uploads an artifact. A second workflow on `workflow_run` has permissions, downloads that artifact, and treats its contents as data, never as code.
16. `workflow_run` carries the same hazard in a different coat: it also runs the file from the default branch with a full token, so an artifact produced by an untrusted run must never be executed or expanded into a path you trust.
17. GitLab expresses all of this through `rules` and the `CI_PIPELINE_SOURCE` variable, whose values include `push`, `merge_request_event`, `schedule`, `web`, `api`, `trigger` and `pipeline`. GitLab further distinguishes merge request pipelines from merged results pipelines, which test the combined result rather than the source branch.
18. Observing tools: `gh run list --json event,headBranch,headSha`, the event name shown at the top of any run page, and `gh api repos/OWNER/REPO/hooks` to see which webhooks a repository sends to outside systems.

41.5.6 WORDS - remember these

1. Event — the thing that happened — a webhook payload describing a repository or platform change.
2. Trigger — the rule that listens — the `on:` key naming events, activity types and filters.
3. Activity type — the sub-kind of an event — such as `opened` or `synchronize` for `pull_request`.

4. Path filter — only run when these files changed — `paths` or `paths-ignore` globs matched against the push diff.
5. `workflow_dispatch` — the manual run button — a non-webhook trigger with up to 10 typed inputs, also callable by API.
6. `repository_dispatch` — an outside system asking for a run — a POST to the repository's dispatches endpoint with an `event_type`.
7. Pwn request — a stranger's code running with your keys — arbitrary code execution obtained by checking out untrusted head under `pull_request_target`.

41.6 The most important idea: a run belongs to a commit, not a branch

41.6.1 PLAIN - in simple words

1. This is the idea the whole chapter is built around. Read it twice.
2. A branch name is not a box holding commits. It is a label stuck on exactly one commit.
3. Chapter 38 put it this way: a branch is a moving pointer. Add a commit and the label moves forward to the new one.
4. A test run is not stuck to a label. It is stuck to one exact commit.
5. The machine downloaded one commit, tested those exact bytes, and wrote the answer against that commit's fingerprint.
6. That fingerprint is the commit hash, a long string of hexadecimal digits computed from the contents.
7. So the sentence "CI passed on main" is short for something much longer.
8. In full it means: "CI passed on whichever commit `main` happened to point at when that run started".
9. If `main` has moved since, the sentence stays true and becomes useless. It is a fact about a commit nobody is looking at any more.
10. The tick you see beside a branch name on a web page is a summary drawn at the moment the page loaded. It is not a promise about now.
11. So never accept a tick beside a branch name as evidence that anything in particular was tested.
12. Get the exact commit hash first. Then ask whether that hash passed.

41.6.2 PLAIN - a picture in your head

1. Think of a hotel with a brass plate on one door reading MANAGER.
2. A health inspector visits, inspects the room behind that plate, and issues a certificate.
3. The certificate does not say MANAGER. It says room 412, and it lists what was in room 412 on that day.
4. Months later the manager moves office. Somebody unscrews the brass plate and screws it onto room 509.
5. Now walk up to the door marked MANAGER and ask: is this room certified?
6. The plate is not the room. The certificate belongs to 412. The door in front of you is 509, and nobody has inspected it.
7. The plate is the branch name. The room number is the commit hash. The certificate is the run.

Where this comparison breaks: a hotel room keeps its contents when the plate moves, so you could at least argue about it. A commit is defined by its contents, so it cannot change at all. Alter one byte and it becomes a different commit with a different hash and no certificate whatsoever. Git is stricter than the hotel, not looser.

41.6.3 PLAIN - a worked example

1. During the reader's outage, the machine reported `origin/main` as `0a95cc8`.
2. That was a cached remote-tracking value written by the last successful fetch, not a live fact about the server.
3. Now watch the sequence that makes a green tick lie. The clock times are invented; the shape is exactly real.

```
09:00  main -> 0a95cc8
09:01  run 1187 starts, head_sha = 0a95cc8
09:07  run 1187 finishes, conclusion = success
09:07  branch page shows:  main  [tick]

09:40  a pull request is merged
09:40  main -> 3f1d02c          (the label moved)
09:40  branch page still shows: main  [tick]   <- nothing
                                                  tested 3f1d02c
09:41  run 1194 starts, head_sha = 3f1d02c
09:53  run 1194 finishes, conclusion = failure
```

4. Between 09:40 and 09:41 the page showed a tick beside `main` while nothing at all had tested `3f1d02c`.
5. At 09:53 the honest summary is: `0a95cc8` passed, `3f1d02c` failed, and `main` is `3f1d02c`.
6. If at 09:40 somebody asked "did main pass?", the correct answer is that the question is badly formed.
7. The well-formed question is "did `3f1d02c` pass?", and at 09:40 the answer was "no run for that commit exists yet".
8. There is a second way to get burned where nobody does anything wrong.
9. The reader rebased a local branch onto a moved `main`, and the branch's hash changed.
10. Rebase does not carry commits across. It builds new commit objects with new parents, and new parents mean new hashes.
11. Every run recorded against the old hashes is still there, still correct, and now attached to commits that no branch points at.
12. The branch label now sits on brand new commits with zero runs. The tick you remember belonged to commits that are effectively gone.

41.6.4 PLAIN - what is really happening inside

1. A run record on the server stores many fields. Two of them matter here.
2. `head_sha` is the exact commit the run checked out. This field is the truth.
3. `head_branch` is the branch name as it read at trigger time. This field is a label, and labels move.
4. When you open a branch page, the service does roughly this: find the commit the branch points at right now, then look up the statuses and check runs attached to that commit.
5. That is correct behaviour, and it gives a right answer for the instant the page was rendered.
6. So where does the lie come from? Three places, none of them a bug.
7. Caching in your own tools. The reader's machine reported `0a95cc8` from a file on disk, either `.git/refs/remotes/origin/main` or an entry in `.git/packed-refs`, written by the last fetch.
8. Human memory. You looked at the tick five minutes ago, then merged three pull requests, and then trusted the memory.
9. Frozen evidence. A screenshot or a chat message saying "CI is green, see attached" is a statement about a moment, not about now.
10. The fix is mechanical and takes one extra command.
11. Never carry a branch name into the verification step. Resolve it to a hash once, from the server, and carry the hash.
12. If somebody pushes between your resolve and your check, your conclusion is still exactly true, because it was a statement about the hash you named.
13. That is the whole difference between a claim you can defend and a claim you merely believe.

41.6.5 TECHNICAL - the engineer's version

1. Step one: get the head SHA from the server, not from your local cache.

```
# live from the remote, no local cache involved
git ls-remote origin refs/heads/main
# 3f1d02c9d5f0a1b2c3d4e5f60718293a4b5c6d7e refs/heads/main
```

```
# or through the API
export SHA=$(gh api repos/OWNER/REPO/commits/main --jq .sha)
echo "$SHA"
```

2. `git rev-parse origin/main` does not do this. It reads the remote-tracking ref from disk and answers from cache. That is what produced `0a95cc8` during the reader's outage while the real server value was unknown.
3. Step two: list the runs filtered by that exact SHA, never by branch.

```
gh run list --commit "$SHA" \
  --json databaseId,workflowName,event,status,conclusion,headSha
```

4. The same query in raw REST, which is what `gh` calls underneath:

```
gh api "repos/OWNER/REPO/actions/runs?head_sha=$SHA" \
  --jq '.workflow_runs[]
        | {id, name, event, status, conclusion, head_sha}'
```

5. Step three: look at the one run you care about, by its numeric ID, and read both its conclusion and its head SHA.

```
gh run view 11947722301 \
  --json status,conclusion,headSha,headBranch,event,url
```

6. Step four: assert rather than eyeball. A human reading two long hex strings side by side will miss a mismatch.

```
gh run view "$RUN_ID" --json headSha,conclusion \
  --jq 'if .headSha == env.SHA and .conclusion == "success"
        then "VERIFIED" else "NOT VERIFIED" end'
```

7. Two commit-level views give the rolled-up answer for that SHA, one per API:

```
# newer checks API, one entry per check run
gh api "repos/OWNER/REPO/commits/$SHA/check-runs" \
  --jq '.check_runs[] | {name, status, conclusion}'

# older commit status API, already rolled up
gh api "repos/OWNER/REPO/commits/$SHA/status" \
  --jq '{state, total_count}'
```

8. For a pull request, ask for the head object ID directly rather than deriving it, and pull the rollup in the same call:

```
gh pr view 42 --json headRefOid,statusCheckRollup \
  --jq '{sha: .headRefOid,
         checks: [.statusCheckRollup[]
                  | {name, conclusion}]}'
```

9. Field naming differs between the two interfaces and it trips people up. The `gh --json` output uses camelCase such as `headSha` and `databaseId`. REST uses snake_case such as `head_sha` and `id`. Same values, two spellings.
10. `GET /repos/{owner}/{repo}/actions/runs` accepts `head_sha`, `branch`, `event`, `status`, `actor`, `created`, `check_suite_id` and `exclude_pull_requests` as query parameters.
11. One commit can carry many runs: one per workflow file, plus every re-run, plus a separate run for the `push` event and the `pull_request` event. So "did this SHA pass" is incomplete without "in which workflow".

12. A subtlety worth memorizing. For a `pull_request` event the run's `head_sha` is the pull request branch head, but the code checked out is the ephemeral merge of that head into the base, which is what `GITHUB_SHA` holds. The run is filed under one hash and tested another.
13. The hash itself is a SHA-1 digest over the commit object in classic repositories, and SHA-256 in repositories created with the newer object format, which Git has supported experimentally since version 2.29 in October 2020. The commit object names its tree and its parents, so the hash transitively covers the entire content and history.
14. Because of that, verifying against a hash is verifying against exact bytes. There is no room left for "roughly the same code".
15. Rebase consequence, stated precisely: rebasing writes new commit objects, so check runs and statuses remain attached to the abandoned hashes. They are not migrated, and there is no mechanism that would migrate them. `git reflog` still finds the old commits locally until garbage collection.
16. The sentence to memorize: a run R proves a property about commit `R.head_sha`, and proves nothing whatsoever about any ref.

41.6.6 WORDS - remember these

1. Commit SHA — the fingerprint of one exact snapshot — a 40-hex SHA-1, or 64-hex SHA-256, digest over the commit object.
2. Ref — a name pointing at a commit — a file under `.git/refs` or a line in `.git/packed-refs`.
3. `head_sha` — the commit the run really tested — the field on a workflow run record, spelled `headSha` in `gh --json` output.
4. Remote-tracking ref — your last known copy of the server's branch — `refs/remotes/origin/main`, updated only by fetch, never live.
5. `git ls-remote` — ask the server what it has right now — a live ref advertisement over the transport, bypassing every local cache.
6. Stale tick — a pass mark for a commit that is no longer the head — a status attached to a superseded SHA and displayed next to a moved branch name.

41.7 Why verifying by run ID against a head SHA is the stronger claim

41.7.1 PLAIN - in simple words

1. In the reader's session, continuous integration was verified by run ID against the exact head hash.
2. It was not verified by glancing at a tick beside a branch name. That is the difference between proof and belief.
3. Here is the procedure in words, four steps.
4. Ask the server what the branch points at right now. Write that hash down.
5. List the runs whose tested commit is that hash.
6. Pick the run you care about and note its number.
7. Read that run's result, and read the commit that run tested, and confirm it equals the hash you wrote down.
8. Only then say "this commit passed".
9. That claim survives questioning, because every step names something that cannot move.
10. A hash cannot move. A run number cannot move. A branch name moves all day.
11. The extra cost is about ten seconds and three commands.
12. The cost it avoids is a bad deploy at six o'clock on a Friday.
13. There is a second reason this method is stronger, and it is the reason engineers care most about.
14. It fails loudly. If nothing tested your commit, the list comes back empty and you cannot mistake an empty list for a pass.
15. A tick beside a branch name never comes back empty. It always shows something, and that is exactly the problem.

41.7.2 PLAIN - a picture in your head

1. A school posts a notice on the board: "Class 10B: PASS".
2. You are in class 10B. Are you finished?
3. Not yet. The board is a summary about a group, at a moment, written by somebody in a hurry.
4. Your own certificate carries your roll number. That number was never reassigned to anybody else.
5. Ask for the certificate bearing your roll number, and read the result printed on it.
6. Then compare the roll number on the certificate with the roll number on your identity card.
7. Two documents, one shared identifier, one comparison. That is the entire method, in school and in continuous integration.

Where this comparison breaks: a class notice can only be wrong in one way, by summarizing. A branch tick can be wrong in at least six different ways, and several of them look identical on the screen. Worse, a school certificate is issued by one office, while a commit status can be posted by anything holding a token, including a person typing an API call by hand.

41.7.3 PLAIN - a worked example

1. Start from the branch name and refuse to trust it.

```
$ export SHA=$(gh api repos/OWNER/REPO/commits/main --jq .sha)
$ echo "$SHA"
3f1d02c9d5f0a1b2c3d4e5f60718293a4b5c6d7e

$ gh run list --commit "$SHA" \
    --json databaseId,workflowName,conclusion,headSha
[]
```

2. An empty list. Nothing has ever tested this commit, even though the branch page a moment ago showed a green tick.
3. That tick belonged to `0a95cc8`, the previous head. The claim "main is green" was true about a commit nobody is deploying.
4. Now the good case, after the run has actually happened.

```
$ gh run list --commit "$SHA" \
    --json databaseId,workflowName,conclusion,headSha
[
  {
    "databaseId": 11947722301,
    "workflowName": "ci",
    "conclusion": "success",
    "headSha": "3f1d02c9d5f0a1b2c3d4e5f60718293a4b5c6d7e"
  }
]
```

5. Now name the run and read it back on its own, so the answer does not depend on the filter having worked.

```
$ gh run view 11947722301 \
    --json conclusion,headSha,status,event,runAttempt
{
  "conclusion": "success",
  "headSha": "3f1d02c9d5f0a1b2c3d4e5f60718293a4b5c6d7e",
  "status": "completed",
  "event": "push",
  "runAttempt": 1
}
```

6. Four things were checked and all four matter.
7. `status` is `completed`, so the run finished rather than still running.
8. `conclusion` is `success`, so it finished well.
9. `headSha` equals the hash you resolved, so it tested the right commit.
10. `runAttempt` is 1, so nobody re-ran it until it went green.
11. Now the sentence you may say out loud: commit `3f1d02c` passed workflow `ci` in run 11947722301 on the first attempt.

41.7.4 PLAIN - what is really happening inside

1. Six ways a branch-level tick can lie. Each is a real, ordinary situation, not a bug and not sabotage.
2. **The run was for an older commit.** The branch moved after the run. The page shows the newest result it can find, and briefly that is the old one.
3. **The run tested the merge result, not your head.** For a pull request the service builds a temporary merge of your branch into the target and tests that. Your head commit was never tested alone.
4. **A required check was skipped, not passed.** A job that never ran can report in a way that satisfies a rule, and a workflow that never started reports nothing at all.
5. **The status came from something other than the run you have in mind.** The name "build" on the screen is just a text label. Anything with a token can post that label.
6. **A re-run replaced the result.** Somebody pressed re-run until it went green. The run ID is the same, the number of attempts is not.
7. **The check belongs to a merge commit that no longer exists.** After a squash or rebase merge, the commit that was tested is not the commit that landed on the target branch.
8. Notice what all six have in common. In every case the tick is truthful about something, and that something is not the thing you are about to deploy.
9. The method in 41.7.1 defeats all six because it never asks a question whose subject can move.

41.7.5 TECHNICAL - the engineer's version

1. The failure modes and the exact field to inspect for each.

Failure mode	Detect by
Run is for an older commit	compare `headSha` to head
Run tested the merge result	read the `event` field
Check skipped, not passed	read `conclusion`
Status posted by another app	list `app.slug` per check
Re-run replaced the result	read `runAttempt`
Merge commit gone after merge	resolve the new head SHA

2. Older commit. `gh run list --commit "$SHA"` returns an empty array when nothing tested that hash. Treat an empty array as a hard failure of verification, never as an absence of bad news.
3. Merge result. If `event` is `pull_request`, the run's `head_sha` is the pull request branch head, but the job checked out `refs/pull/N/merge`, and `GITHUB_SHA` inside the job was that ephemeral merge commit. GitHub's own troubleshooting page states that when the test merge commit has a status, it is the test merge commit that must pass, not the head commit.
4. Skipped versus passed. This is the sharpest edge in the whole area, and GitHub documents three different outcomes:

Situation	Reported as
Workflow skipped by path filter	stays Pending, blocks
Workflow skipped by `[skip ci]`	stays Pending, blocks

Situation	Reported as
Job skipped by an `if:` condition	Success
Job skipped after a failed `needs`	skipped, may not block

5. Read that table again. A job you skipped on purpose reports success and satisfies a required check. A workflow that never started reports nothing, stays pending, and blocks forever. Same English word, opposite consequences.
6. The documented mitigation for the fourth row is to add `if: always()` to a final summary job that depends on the others and explicitly fails if any dependency did not succeed.
7. Wrong poster. Any token with write access can create a commit status with any context and state, including `success`:

```
gh api -X POST repos/OWNER/REPO/statuses/$SHA \
  -f state=success -f context=build \
  -f description='looks fine to me'
```

8. That call is legitimate API use and is indistinguishable on the branch page from a real build result. Inspect the poster:

```
gh api "repos/OWNER/REPO/commits/$SHA/check-runs" \
  --jq '.check_runs[] | {name, conclusion, app: .app.slug}'

gh api "repos/OWNER/REPO/commits/$SHA/status" \
  --jq '.statuses[] | {context, state, creator: .creator.login}'
```

9. Re-runs. A re-run does not create a new run ID. It increments `run_attempt` on the same run, and the interface shows the latest attempt. Fetch an earlier attempt explicitly:

```
gh api repos/OWNER/REPO/actions/runs/11947722301/attempts/1 \
  --jq '{attempt: .run_attempt, conclusion, head_sha}'
```

10. "Re-run failed jobs" is more subtle still: it produces a new attempt in which the previously successful jobs are not executed again. The green you see is assembled from two different points in time.
11. Vanished merge commit. A squash merge creates one brand new commit on the target with a new hash and no checks. A rebase merge creates new commits for the same reason. Only a true merge commit preserves the tested parent.
12. So after any squash or rebase merge, the correct next step is to resolve the new head of the target branch and verify that hash on its own. What passed before the merge is evidence about a commit that is no longer in the first parent line of your branch.
13. Two further modes worth knowing, beyond the six:
14. `continue-on-error: true` on a step or job makes a real failure report as success at the job or run level. Read the step conclusions, not only the run conclusion.
15. A `strategy.matrix` job cancelled by `fail-fast: true` reports `cancelled`, which is not `failure`, and people read the absence of red as green.
16. A script for the whole verification, safe to put in a release checklist:

```
set -euo pipefail
REPO=OWNER/REPO
SHA=$(gh api "repos/$REPO/commits/main" --jq .sha)
RUN=$(gh run list --repo "$REPO" --commit "$SHA" \
        --workflow ci --limit 1 --json databaseId \
        --jq '.[0].databaseId')
test -n "$RUN" || { echo "no run for $SHA"; exit 1; }
gh run view "$RUN" --repo "$REPO" \
  --json conclusion,headSha,status,runAttempt \
```

```
--jq --arg sha "$SHA" '
  if .status=="completed" and .conclusion=="success"
     and .headSha==$sha
  then "VERIFIED \($sha) run \(.runAttempt) attempt(s)"
  else error("NOT VERIFIED") end'
```

17. GitLab equivalent concepts: a pipeline has a `sha`, a `status`, and an `id`. `glab ci status` and the pipelines API filtered by `sha` do the same job. Jenkins records the built revision in the build metadata, reachable through the `api/json` endpoint of a build.

41.7.6 WORDS - remember these

1. Verification by run ID — naming the run rather than the branch — asserting `run.head_sha == resolved_head` and `conclusion == success`.
2. Test merge commit — the temporary join of your branch into the target — the `refs/pull/N/merge` ref that pull request runs actually check out.
3. Run attempt — which try you are looking at — the `run_attempt` counter, incremented by a re-run, with the latest shown by default.
4. Commit status — a small note pinned to a commit — a state plus a context, creatable by any token with write access.
5. `continue-on-error` — keep going after a failure — a step or job setting that turns a real failure into a reported success upstream.
6. Squash merge — many commits collapsed into one new commit — a merge strategy producing a target-branch commit that no check has ever seen.

41.8 Statuses, checks, and the two APIs behind them

41.8.1 PLAIN - in simple words

1. There are two separate systems for pinning a result onto a commit. They were built years apart and both are still alive.
2. The older one is the **commit status**. It is a small note with four parts: a state, a short description, a link, and a name called the context.
3. Its states are `pending`, `success`, `failure` and `error`. That is all it can say.
4. The newer one is the **check run**, part of the checks system.
5. A check run says much more: whether it is queued, running or finished; what the outcome was; when it started and ended; a block of output text; and notes pinned to particular lines of particular files.
6. Check runs are collected into a **check suite**, which is roughly "everything one tool reported about one commit".
7. Both systems attach their results to a commit hash. Both draw ticks and crosses in the same place on the page. That is why they get confused.
8. Branch protection lets you name some of these results as **required**.
9. A required result means: this pull request may not merge until a result with exactly that name reports success.
10. And here is the sharp edge. A required check that never arrives blocks the merge forever.
11. It blocks because "has not arrived" is not the same as "failed". It sits in a third state that means "we are still waiting for this".
12. Nothing times it out. Nothing gives up. It waits, and the merge button stays grey.

41.8.2 PLAIN - a picture in your head

1. Imagine a building site where a flat cannot be handed over until three inspection slips are pinned to the door.
2. Slip one is from the electrician, slip two from the plumber, slip three from the fire officer.

3. The old slips were postcards: a tick or a cross, a line of writing, nothing else.
4. The new slips are full forms: who inspected, when they started, when they finished, and photographs with arrows pointing at specific pipes.
5. The handover clerk does not judge quality. The clerk only checks that all three named slips are present and say "pass".
6. Now suppose the fire officer was never told this flat exists.
7. There is no cross on the door. There is simply no third slip.
8. The clerk waits. Nobody is at fault, no alarm rings, and the flat is never handed over.

Where this comparison breaks: real inspectors eventually notice and telephone somebody. A required check has nobody to telephone. The rule is purely mechanical, and the only cure is a human editing the rule or arranging for the missing slip to appear.

41.8.3 PLAIN - a worked example

1. A commit status, created by hand, exactly as an old-style build tool would:

```
gh api -X POST repos/OWNER/REPO/statuses/$SHA \
  -f state=success \
  -f context='ci/unit-tests' \
  -f target_url="$BUILD_PAGE_URL" \
  -f description='412 tests, 0 failures'
```

2. A check run, as posted by an app, has far more structure:

```
{
  "name": "unit-tests",
  "head_sha": "3f1d02c9d5f0a1b2c3d4e5f6...",
  "status": "completed",
  "conclusion": "failure",
  "started_at": "2026-08-13T09:41:02Z",
  "completed_at": "2026-08-13T09:53:11Z",
  "output": {
    "title": "3 failing",
    "summary": "412 tests, 3 failures, 0 skipped",
    "annotations": [
      {
        "path": "src/cart/total.ts",
        "start_line": 44,
        "annotation_level": "failure",
        "message": "expected 1200, received 1199"
      }
    ]
  }
}
```

3. The annotation is the visible difference. It is what draws a red marker on line 44 of that file inside the pull request diff.
4. Reading both systems for one commit:

```
# newer: individual check runs
gh api "repos/OWNER/REPO/commits/$SHA/check-runs" \
  --jq '.check_runs[] | {name, status, conclusion}'

# older: the rolled-up commit status
gh api "repos/OWNER/REPO/commits/$SHA/status" \
  --jq '{state, statuses: [.statuses[].context]}'
```

5. The rollup rule for the older API is documented and simple: the combined state is `failure` if any context is `error` or `failure`; `pending` if there are no statuses at all or any context is `pending`; `success` only if every context's latest status is `success`.
6. Read that middle clause again. Zero statuses rolls up to `pending`, not to `success`. Silence is treated as "not yet", which is exactly the behaviour that makes a missing required check block forever.

41.8.4 PLAIN - what is really happening inside

1. When a run starts, the service creates a check suite for that commit and one check run per job.
2. Each check run begins as queued, becomes in progress, and finally becomes completed with a conclusion.
3. The conclusion is written once, when the job ends. Until then there is no conclusion at all, only a status.
4. That two-field design is the whole reason "pending" and "failed" are different things: `status` says whether it is over, `conclusion` says how it went, and the second is empty until the first says completed.
5. Branch protection compares a list of required names against the results present on the head commit of the pull request.
6. The comparison is by name, as text. It has no idea which tool produced the result and no idea what the tool did.
7. So renaming a job in your workflow file silently breaks a required check. The old name never arrives again, and every pull request blocks.
8. This is the single commonest cause of "the repository is stuck and nobody changed the rules". Somebody did change something: the job's `name`.
9. Path filters cause the same shape of failure from the other direction. If the workflow does not start, the check does not exist, and the required name never arrives.
10. The cure is a small job that always runs and always reports the required name, whose only work is to look at the other jobs and decide.

41.8.5 TECHNICAL - the engineer's version

1. The commit status API is the older of the two and dates from the early years of the GitHub API. Create with `POST /repos/{owner}/{repo}/statuses/{sha}`. Fields: `state` (one of `error`, `failure`, `pending`, `success`), `target_url`, `description`, and `context` which defaults to the literal string `default`.
2. Documented limit: 1,000 statuses per SHA and context pair in a repository. Read them with `GET /repos/{owner}/{repo}/commits/{ref}/statuses` for the list, or `.../status` for the rollup.
3. In GraphQL the type is `StatusState`, with exactly five values: `ERROR`, `EXPECTED`, `FAILURE`, `PENDING`, `SUCCESS`. `EXPECTED` is the one you cannot post yourself; it is what a required-but-absent check looks like.
4. The checks API arrived with GitHub Apps in 2018 and is richer. A **check suite** is a collection of check runs created by one app for one commit. A **check run** is one individual test of that commit.
5. Only a GitHub App can create check runs. `GITHUB_TOKEN` inside Actions acts as an app installation token, which is why Actions can create them and your personal token generally cannot.
6. In GraphQL, `CheckStatusState` has six values: `QUEUED`, `REQUESTED`, `WAITING`, `PENDING`, `IN_PROGRESS`, `COMPLETED`. The REST spelling is lowercase.
7. `CheckConclusionState` has nine values. This table is the one to keep.

Conclusion	Means	Required check
`success`	it passed	satisfied
`failure`	it failed	blocks
`neutral`	ran, no judgement	satisfied
`cancelled`	stopped early	blocks
`skipped`	did not execute	see 41.7.5
`timed_out`	exceeded its limit	blocks

Conclusion	Means	Required check
`action_required`	needs a human	blocks
`stale`	superseded by GitHub	blocks
`startup_failure`	never got going	blocks

8. Notes on the awkward ones. `neutral` exists so a linter can report advice without blocking; it counts as not-failing. `action_required` is what an app returns when it needs you to authorize something, such as granting access.
9. `stale` can only be set by GitHub itself, and marks a result the platform has decided is no longer meaningful. `startup_failure` means the run could not begin, most often an invalid workflow file or an unavailable runner label.
10. `timed_out` on GitHub Actions arrives at `timeout-minutes` if you set one, and at the platform limit of 6 hours for a hosted job if you did not.
11. Branch protection matches required checks by name string against results on the head commit. Configure with `PATCH /repos/{owner}/{repo}/branches/{branch}/protection` or, in the newer model, with repository rulesets.
12. The permanent-block conditions, all documented, all real:

Cause	Effect
Workflow skipped by filter	check absent, stays Expected
Job `name` renamed	old name never arrives
App uninstalled	its checks never post
Required name never existed	typo blocks every PR

13. `[skip ci]`, `[ci skip]`, `[no ci]`, `[skip actions]` and `[actions skip]` in a commit message, or a `skip-checks: true` trailer, suppress the run entirely. On a repository with required checks that is a self-inflicted permanent block.
14. The standard workaround is a gate job that always runs:

```
ci-gate:
  if: always()
  needs: [lint, test, build]
  runs-on: ubuntu-latest
  steps:
    - name: Fail if any dependency did not succeed
      if: contains(needs.*.result, 'failure')
          || contains(needs.*.result, 'cancelled')
      run: exit 1
```

15. Require `ci-gate` and nothing else. It runs on every event that the workflow runs on, and its name is stable even when the jobs beneath it are renamed.
16. GitLab models the same idea differently: a job has `allow_failure`, a pipeline has a single status, and merge request approval rules reference the pipeline rather than individual named checks. There is no direct equivalent of a required check name, which removes this class of bug and removes the fine-grained control with it.
17. Observing tools: `gh pr checks 42`, `gh pr checks 42 --watch`, `gh api repos/OWNER/REPO/commits/$SHA/check-runs`, `gh api repos/OWNER/REPO/branches/main/protection --jq .required_status_checks`.

41.8.6 WORDS - remember these

1. Commit status — the old-style note on a commit — a `state`, `context`, `description` and `target_url` posted to the statuses endpoint.
2. Context — the name of an old-style status — the string branch protection matches against, defaulting to `default`.
3. Check run — the new-style detailed result — an app-created record with status, conclusion, timings, output and line annotations.
4. Check suite — all check runs from one app for one commit — the grouping the interface shows as a single reporting tool.
5. Annotation — a note pinned to a line of a file — a check run output entry with `path`, `start_line` and `annotation_level`.
6. Required status check — a name that must report success — a branch protection rule compared by text against results on the head commit.
7. Expected — waiting for a result that never came — `StatusState.EXPECTED`, the state that blocks a merge indefinitely.

41.9 Merge queues

41.9.1 PLAIN - in simple words

1. Here is a problem that green ticks cannot solve on their own.
2. Two people each propose a change. Each change is tested alone, and each passes.
3. Both are merged within a minute of each other. The result is broken.
4. Nothing went wrong with either test. The tests answered the questions they were asked.
5. Neither test was ever asked the only question that matters: what happens when both of these land together?
6. Chapter 39 called this a semantic conflict: two changes that do not touch the same lines, and still contradict each other.
7. One person renames a function. The other person adds a new call to the old name. No line overlaps. The program does not build.
8. A **merge queue** exists to ask the missing question before the merge, not after.
9. It works like a queue at a counter. Changes wait in a line, and only one thing is joined at a time.
10. For each change in the line, the machine builds the result you would get if that change landed on top of everything ahead of it.
11. It tests that combined result.
12. Only if that passes does the change actually land.
13. So the thing that is tested is the thing that lands, in the order it lands.
14. That is the whole idea, and it is the only mechanical way to keep a mainline that always builds.

41.9.2 PLAIN - a picture in your head

1. Think of an aeroplane runway with one strip and a queue of aircraft.
2. Each pilot has already done a check on their own aircraft in the hangar. All passed.
3. But the tower does not clear them by hangar checks. The tower clears one aircraft at a time, in a known order, and re-checks the runway state each time.
4. Now add impatience. Waiting for each aircraft to finish before starting the next check is slow.
5. So the tower guesses. It assumes number one will be fine, and starts checking number two against a runway that already has number one on it.
6. It does the same for three and four. Four checks run at once, each assuming everything ahead of it succeeded.
7. If all guesses hold, everything lands in one runway's worth of time instead of four.

8. If number two turns out to be faulty, the guesses made for three and four were built on a false assumption and must be thrown away and redone.
9. That guessing is called speculative execution, and it is the trick that makes queues fast enough to use.

Where this comparison breaks: aircraft are physical and independent; software changes interact. A faulty aircraft does not make the next one faulty, but a bad merge genuinely can invalidate every speculation behind it. Also, a runway handles one landing at a time by physics. A merge queue chooses to serialize, and could choose not to, at the cost of correctness.

41.9.3 PLAIN - a worked example

1. Three pull requests are ready. The test suite takes 12 minutes.
2. The old discipline, "rebase onto the latest mainline and re-run before merging", plays out like this.

```
 0 min  A rebases onto main, starts CI
12 min  A green, A merges. main moves.
12 min  B rebases onto new main, starts CI
24 min  B green, B merges. main moves.
24 min  C rebases onto new main, starts CI
36 min  C green, C merges.
```

3. Total elapsed: 36 minutes, and three humans had to sit and watch.
4. The same three in a merge queue with speculation.

```
 0 min  A, B, C enter the queue in that order
 0 min  build main+A        (entry 1)
 0 min  build main+A+B      (entry 2, speculative)
 0 min  build main+A+B+C    (entry 3, speculative)
12 min  all three green
12 min  A, B, C merge in order. main moves once.
```

5. Total elapsed: about 12 minutes, and nobody watched anything.
6. Now the interesting case. Entry 2 fails, because B contradicts A.

```
12 min  entry 1 green, entry 3 green, entry 2 FAILED
12 min  A merges
12 min  B is removed from the queue and its author told
12 min  entry 3 is discarded: it assumed B would land
12 min  rebuild main+A+C
24 min  green, C merges
```

7. Entry 3's result was correct about a world that never happened, so it is thrown away. That is the cost of speculation.
8. Costs compared, using 12-minute runs and a 5 percent chance that any given change fails when combined:

Approach	3 changes, all pass	Human effort
Merge and hope	12 min	none, breaks main
Rebase and re-run	36 min	three people wait
Queue, no speculation	36 min	none
Queue, speculative	12 min	none

9. The wasted machine time under speculation is real: a failure at position two discards every build behind it. Teams accept that because machine minutes are cheaper than a broken mainline and cheaper than engineers waiting.

41.9.4 PLAIN - what is really happening inside

1. When a pull request is added to the queue, the service does not merge it. It creates a temporary branch.
2. That temporary branch contains the target branch, plus every queued change ahead of this one, plus this one.
3. That temporary branch is the thing your workflows run against. It is a real commit, with a real hash, and the run is filed against that hash.
4. This is why a merge queue needs its own trigger. A workflow that only listens for `pull_request` will never run for a queue entry.
5. When the run for an entry finishes green and everything ahead of it has merged, the entry is merged into the target branch.
6. If the run fails, that entry is removed from the queue, the author is told why, and the temporary branches for everything behind it are rebuilt without it.
7. Batching is the other lever. Instead of one entry per build, group five entries into one temporary branch and test the group.
8. A green batch merges five changes for the price of one build. A red batch tells you only that one of five is bad, and you must split it to find out which.
9. Splitting a failed batch in half repeatedly finds the culprit in about log2(n) further builds. That is the same bisection idea as `git bisect` from Chapter 40.
10. Every serious queue has the same three knobs: how many entries to build at once, how many to merge at once, and how long to wait before giving up on filling a batch.

41.9.5 TECHNICAL - the engineer's version

1. The property a merge queue enforces has a name in the Rust community: the "not rocket science rule of software engineering", a phrase attributed to Graydon Hoare, Rust's original author. Stated plainly: automatically maintain a repository of code that always passes all the tests.
2. History. The Rust project's `bors` bot implemented this from the early 2010s, later reimplemented as `homu` and `bors-ng`. OpenStack's Zuul, in use from 2012, added speculative execution across multiple projects and called the mechanism a gate pipeline. Zuul v3 arrived in 2017.
3. Commercial and hosted implementations followed: Mergify, Aviator, Graphite, and GitLab's merge trains, which appeared in GitLab 11.2 in 2018.
4. GitHub's own merge queue entered public beta on 8 February 2023 and became generally available on 12 July 2023.
5. Mechanically, on GitHub: entries create temporary branches under a reserved prefix, `gh-readonly-queue/`, one per entry, named after the base branch and the pull request. You cannot push to them.
6. Workflows must listen for the `merge_group` event with activity type `checks_requested`, or they will not run for queue entries at all:

```
on:
  pull_request:
    branches: [main]
  merge_group:
    types: [checks_requested]
```

7. Forgetting `merge_group` is the classic first-day merge queue failure. The entry sits waiting for a required check that no workflow will ever produce, which is exactly the permanent-block condition from 41.8.
8. Documented settings, with their ranges as of August 2026:

Setting	Range	Meaning
Build concurrency	1 to 100	speculative builds at once
Maximum to merge	1 to 100	largest batch
Minimum to merge	1 to 100	smallest batch

Setting	Range	Meaning
Wait time	minutes	how long to fill a batch

9. Merge method is configurable as merge, rebase or squash, and applies to the whole queue rather than per pull request.
10. "Only merge non-failing pull requests" controls batch semantics. When on, every entry must pass. When off, a group may merge if the final entry passes for the combined changes, which is faster and admits a change that never passed on its own.
11. On failure, GitHub removes the entry from the queue, records the reason on the pull request timeline, recreates the temporary branch without it, and continues with the rest.
12. Cost model. With batch size `b`, per-run time `T`, and independent failure probability `p` per change, a batch is green with probability `(1 - p)^b`. Expected builds per merged change falls with `b` while all is well, and rises sharply once `b * p` approaches 1. Most teams land on batches of 2 to 8.
13. The older discipline is still available and still correct: branch protection offers "Require branches to be up to date before merging", which forces each author to rebase or merge the latest target and re-run before the merge button unlocks.
14. That setting has two costs a queue does not: it serializes humans rather than machines, and it has a race. Between your last green run and your click, somebody else can merge. The window is small but not zero.
15. It also scales badly. On a busy repository, the mainline moves faster than your test suite runs, so you can never get to the front of the line. Teams hit this at roughly `merges per hour x test minutes > 60`.
16. Comparison of approaches:

Approach	Mainline safety	Cost
Merge on green PR	not guaranteed	cheapest
Up-to-date required	guaranteed	humans serialize
Queue, serial	guaranteed	slow, no waste
Queue, speculative	guaranteed	fast, wastes builds

17. Observing tools: `gh pr merge --auto`, `gh api repos/OWNER/REPO/merge-queue` for queue state, the `merge_group` payload inside the run, and `git ls-remote origin 'refs/heads/gh-readonly-queue/*'` to see live entries.

41.9.6 WORDS - remember these

1. Merge queue — a line of changes joined one at a time — a serializing gate that tests the prospective merged state before merging.
2. Semantic conflict — two changes that clash without touching the same lines — a textually clean merge that fails to build or fails tests.
3. Speculative execution — testing entries assuming the ones ahead succeed — parallel builds of stacked prospective states, discarded on any failure.
4. Batch — several entries tested as one — a group merged together, requiring bisection to locate a culprit when it fails.
5. `merge_group` — the event a queue entry raises — the GitHub Actions trigger with activity type `checks_requested`.
6. Merge train — GitLab's name for the same idea — sequential pipelines on the projected result of merging each queued request.
7. Not rocket science rule — always keep the mainline passing — the invariant a gating queue mechanically enforces.

41.10 `mergeStateStatus`, value by value

41.10.1 PLAIN - in simple words

1. When you ask GitHub about a pull request, one field tells you why the merge button is or is not available. It is called `mergeStateStatus`.
2. It has exactly eight possible values, and they are not all the same kind of thing, which is why people misread them.
3. Some values are about the code itself: can these two branches be joined at all?
4. Some are about the rules: the code is joinable, but a policy says not yet.
5. One is about the pull request's own state: it is a draft.
6. And one is about GitHub itself: I have not worked it out yet, ask again.
7. The reader saw three of them in one session: `CLEAN`, `BLOCKED` and `UNKNOWN`.
8. Two of these are worth learning before the rest.
9. `UNKNOWN` almost never means something is wrong. It means the answer is still being computed. The correct response is to ask again in a second.
10. `BLOCKED` does not mean a merge conflict. It means the branches join fine and a protection rule is not satisfied.
11. Getting those two right removes most of the confusion this field causes.

41.10.2 PLAIN - a picture in your head

1. Think of a clerk at a counter deciding whether your form can be filed.
2. `DIRTY` is: your form physically will not fit in the folder. The paper is the wrong shape. Fix the paper.
3. `BEHIND` is: your form is on an old version of the template. Copy it onto the current template.
4. `BLOCKED` is: the form is perfect and the folder is ready, but a signature is missing. Nothing is wrong with your paper.
5. `UNSTABLE` is: everything is signed, but one optional report attached to it came back with a complaint. The clerk will still file it if you insist.
6. `DRAFT` is: you wrote "draft" across the top yourself. The clerk is obeying you.
7. `CLEAN` is: file it.
8. `UNKNOWN` is: the clerk has not finished reading it yet. Come back in a moment.

Where this comparison breaks: a clerk gives you one reason. `mergeStateStatus` gives you the first blocking reason it finds, in an order GitHub does not publish. A pull request can be behind and blocked and have failing checks all at once, and you will be shown a single word. Never treat that word as a complete list of what is wrong.

41.10.3 PLAIN - a worked example

1. Asking for the field, together with the things that explain it:

```
gh pr view 42 --json \
  number,mergeable,mergeStateStatus,isDraft,headRefOid,\
reviewDecision
```

2. A typical answer while GitHub is still thinking:

```
{
  "number": 42,
  "mergeable": "UNKNOWN",
  "mergeStateStatus": "UNKNOWN",
  "isDraft": false,
  "headRefOid": "3f1d02c9d5f0a1b2c3d4e5f60718293a4b5c6d7e",
  "reviewDecision": "APPROVED"
}
```

3. Wait one or two seconds and ask again. This is not a workaround. It is the documented way the field behaves.

```
{
  "mergeable": "MERGEABLE",
  "mergeStateStatus": "BLOCKED",
  "reviewDecision": "APPROVED"
}
```

4. Read those two lines together, because alone each is misleading.
5. `mergeable: MERGEABLE` says the branches join without conflict.
6. `mergeStateStatus: BLOCKED` says a rule is unsatisfied. The code is fine. The policy is not.
7. To find out which rule, ask the checks, not this field:

```
gh pr checks 42
gh api repos/OWNER/REPO/branches/main/protection \
  --jq .required_status_checks.contexts
```

8. Nine times out of ten the answer is a required check sitting in `Expected`, which is exactly the permanent block from 41.8.

41.10.4 PLAIN - what is really happening inside

1. GitHub does not keep a permanent answer to "can this merge?".
2. It computes one on demand, in the background, by attempting a test merge of the head into the base.
3. That computation takes time, and the API answers immediately whether or not it has finished.
4. So the first response after any change to either branch is very often `UNKNOWN`, with the REST equivalent field `mergeable` set to `null`.
5. Asking again triggers or picks up the computation. This is documented, and it is why every serious script polls this field rather than reading it once.
6. Once the test merge succeeds, GitHub knows the branches join, and moves on to the rules.
7. It then reports the first obstacle it finds: draft status, protection rules, check results, or nothing at all.
8. This is why the values are a mixture of categories. The field is not a classification of the pull request. It is a report of the first thing stopping the button from lighting up.

41.10.5 TECHNICAL - the engineer's version

1. `mergeStateStatus` is a GraphQL enum on the `PullRequest` type. GitHub's documented values and descriptions, with the action each implies:

Value	GitHub's description	Do this
`BEHIND`	head ref is out of date	update the branch
`BLOCKED`	the merge is blocked	find the unmet rule
`CLEAN`	mergeable, status passing	merge it
`DIRTY`	merge commit not creatable	resolve conflicts
`DRAFT`	blocked, PR is a draft	mark ready for review
`HAS_HOOKS`	mergeable, pre-receive hooks	merge, hooks may reject
`UNKNOWN`	state cannot be determined	re-query, conclude nothing
`UNSTABLE`	mergeable, status not passing	fix or override

2. `UNKNOWN`, precisely. The mergeability computation is asynchronous. In REST, `GET /repos/{owner}/{repo}/pulls/{number}` returns `mergeable: null` until the background job completes, and the documentation instructs you to retry. In GraphQL the same condition surfaces as `UNKNOWN`.

3. Therefore: `UNKNOWN` is not an error, not a conflict, and not a policy failure. It is "not yet". Poll with a short delay, a few times, before treating it as anything else. Concluding anything from a single `UNKNOWN` is the commonest mistake made with this field.
4. `BLOCKED`, precisely. It means the branches merge cleanly and a branch protection or ruleset requirement is unmet. It is not a merge conflict. `DIRTY` is the merge conflict value.
5. Common causes of `BLOCKED`, none of which are visible in the field itself: a required review missing, a requested change outstanding, a required status check absent or failing, a CODEOWNERS review missing, a required conversation unresolved, or a required deployment or signature rule.
6. `UNSTABLE` versus `BLOCKED` is the distinction people most often get backwards. `UNSTABLE` means a check is not passing but that check is not required, so a user with permission can still merge. `BLOCKED` means a rule forbids it outright.
7. `BEHIND` appears only when the repository has "Require branches to be up to date before merging" enabled. Without that setting, an out-of-date branch reports `CLEAN` or `UNSTABLE` instead, and merges happily.
8. `HAS_HOOKS` is specific to GitHub Enterprise Server, where an administrator can install pre-receive hooks. It means everything is fine but a server-side hook still gets the final word at push time, so a merge can still be rejected after you click.
9. `DIRTY` means Git cannot construct the merge commit without conflict resolution. Nothing in continuous integration will fix it. A human must rebase or merge and resolve.
10. Companion fields you should always fetch alongside it, because `mergeStateStatus` alone is not diagnostic:

Field	Tells you
`mergeable`	conflict-free or not
`reviewDecision`	approved, changes needed
`statusCheckRollup`	per-check results
`isDraft`	explains `DRAFT`

11. On some GitHub Enterprise Server versions, `mergeStateStatus` sits behind a schema preview and requires the Accept header `application/vnd.github.merge-info-preview+json`. The `gh` CLI sets what it needs; a hand-written GraphQL client may not.
12. Practical polling pattern, which is what a release script should contain:

```
for i in 1 2 3 4 5; do
  S=$(gh pr view 42 --json mergeStateStatus \
        --jq .mergeStateStatus)
  [ "$S" != "UNKNOWN" ] && break
  sleep 2
done
echo "state: $S"
case "$S" in
  CLEAN|HAS_HOOKS) echo "safe to merge" ;;
  BLOCKED)  echo "a rule is unmet; run gh pr checks" ;;
  UNSTABLE) echo "a non-required check is failing" ;;
  BEHIND)   echo "update branch and re-run CI" ;;
  DIRTY)    echo "resolve conflicts by hand" ;;
  DRAFT)    echo "mark ready for review" ;;
  UNKNOWN)  echo "still computing; do not conclude" ;;
esac
```

13. GitHub's own documentation notes that the REST field `mergeable_state`, the lowercase sibling of this enum, has historically been treated as unofficial and subject to change. Prefer the GraphQL enum, and prefer reading the underlying facts over branching on this one word.

41.10.6 WORDS - remember these

1. `mergeStateStatus` — why the merge button is or is not lit — a GraphQL enum on `PullRequest` with eight values.
2. `mergeable` — can the branches be joined at all — a separate tri-state field, `null` or `UNKNOWN` while still computing.
3. `CLEAN` — nothing is in the way — mergeable with passing commit status.
4. `BLOCKED` — a rule says no, not a conflict — branch protection or ruleset requirement unmet.
5. `UNSTABLE` — mergeable but something non-required is failing — merge is permitted, quality is not vouched for.
6. `DIRTY` — the branches genuinely conflict — Git cannot construct the merge commit.
7. `BEHIND` — your branch is on an old base — only reported when "require branches to be up to date" is enabled.
8. `UNKNOWN` — GitHub has not finished computing — re-query; it carries no information about the pull request.

41.11 What a good pipeline actually does, in order

41.11.1 PLAIN - in simple words

1. A pipeline is a list of checks in a chosen order. The order is not decoration.
2. The rule is: cheapest and most likely to fail goes first.
3. A check that takes 8 seconds and catches a typo should never run after a check that takes 20 minutes.
4. So a sensible order looks like this.
5. **Lint**: read the code without running it, and complain about style and obvious mistakes. Seconds.
6. **Build**: turn source into something runnable. If it does not build, nothing after this matters.
7. **Unit tests**: test small pieces on their own, with nothing real attached. Fast, many, precise about where the fault is.
8. **Integration tests**: start real pieces together, a database, a queue, and check they talk correctly. Slower, fewer, vaguer about where the fault is.
9. **Security scanning**: look for known-bad dependencies, leaked keys, and dangerous code patterns.
10. **Artifact build**: package the thing you will actually ship, once, and keep it.
11. **Deploy to staging**: put that exact package on a copy of production.
12. **Smoke tests**: a handful of checks against the running copy. Does the home page load? Can one user log in?
13. **Production**: ship the same package that passed everything above.
14. Notice the package is built once and reused. Rebuilding before production would mean shipping something nobody tested.

41.11.2 PLAIN - a picture in your head

1. Think of security at an airport, in the order they actually use.
2. First a glance at your ticket at the door. One second, catches most people who are in the wrong building.
3. Then the document check. Thirty seconds.
4. Then the bag scanner. Two minutes.
5. Then, for a few people, a hand search. Ten minutes.
6. Nobody hand-searches everybody first. It would be correct and it would be useless, because the queue would never move.
7. Each stage is more expensive and catches fewer people than the one before.

Where this comparison breaks: airport stages are independent, and a pipeline's stages are not. A build failure makes every later stage meaningless, so a pipeline stops, while an airport keeps processing the rest of the queue. Also, the airport cannot rerun you from a cache; a pipeline can and should.

41.11.3 PLAIN - a worked example

1. Real timings from a medium web project. Yours will differ; the shape holds.

Stage	Time	Catches
Lint	25 s	style, unused vars
Build	1 m 40 s	type and syntax errors
Unit tests	3 m	logic faults
Integration	7 m	wiring, SQL, contracts
Security scan	2 m	known bad dependencies
Package	1 m 10 s	nothing, it produces
Staging deploy	2 m	config and secrets faults
Smoke tests	40 s	it is alive

2. Run in strict order, that is about 18 minutes. Too slow.
3. Run lint, unit tests and the security scan in parallel after the build, and the critical path drops to about 12 minutes.
4. Caching the dependency download saves roughly 60 to 90 seconds of that on every single run.
5. A worked cache and artifact pair in GitHub Actions:

```
- uses: actions/cache@v4
  with:
    path: ~/.npm
    key: npm-${{ hashFiles('package-lock.json') }}
    restore-keys: npm-

- run: npm ci && npm run build

- uses: actions/upload-artifact@v4
  with:
    name: app-${{ github.sha }}
    path: dist/
    retention-days: 14
```

6. Note the cache key. It is a hash of the lock file, so a dependency change makes a new key automatically, and a stale cache can never be used.
7. Note the artifact name. It carries the commit SHA, so the package you deploy can be traced back to the exact commit that produced it.

41.11.4 PLAIN - what is really happening inside

1. Two words that sound alike and are not: cache and artifact.
2. A **cache** is a speed trick. It holds things you could recreate, like downloaded dependencies. Losing it costs time and nothing else.
3. An **artifact** is a product. It holds the thing you built and will ship. Losing it means you cannot deploy without rebuilding.
4. Never treat a cache as a source of truth, and never rebuild an artifact between test and deploy.
5. **Parallelism** splits work across machines. Four machines running a quarter of the tests each finish in roughly a quarter of the time, plus the fixed startup cost of each machine.
6. That fixed cost is why splitting a 40-second suite into eight parts makes it slower, not faster.
7. **Matrix builds** are parallelism over configurations rather than over test files: the same suite on three operating systems and two language versions.
8. **Fail-fast** decides what happens when one parallel branch fails. On, the siblings are cancelled, saving money. Off, they finish, giving you the full picture.

9. Use fail-fast on for pull requests, where speed matters, and off for nightly matrix runs, where the full picture matters.
10. A **flaky test** is a test that passes and fails on the same code, with no change in between.
11. Flaky tests do something worse than waste time. They teach people that red means "press the button again".
12. Once a team believes that, red stops carrying information, and the whole pipeline becomes decoration. This is the way continuous integration dies.

41.11.5 TECHNICAL - the engineer's version

1. The stage order above is the pipeline described in Continuous Delivery by Jez Humble and David Farley in 2010, where it is called a deployment pipeline, with the commit stage first and later stages progressively slower and more production-like.
2. The proportions come from the test pyramid, named by Mike Cohn in Succeeding with Agile in 2009: many fast unit tests, fewer integration tests, very few end-to-end tests.
3. The 10-minute build rule comes from Extreme Programming and is repeated in the Continuous Delivery literature. Past roughly 10 minutes, developers stop waiting for the result and context-switch, and the feedback loop that justifies the whole apparatus is gone.
4. GitHub Actions cache limits as documented in August 2026: 10 GB total per repository, with least-recently-used eviction, and entries removed after 7 days without access. Cache is keyed by an exact string, with `restore-keys` giving ordered prefix fallbacks.
5. Artifact retention on GitHub Actions defaults to 90 days, configurable from 1 to 90 days for public repositories and 1 to 400 days for private and enterprise repositories.
6. Caches are scoped by branch: a run on a branch can read caches from that branch and from the default branch, but not from unrelated branches. This is a deliberate isolation boundary, since a cache is a writable path shared between runs and therefore a cross-branch attack surface.
7. Security scanning splits into named categories, and mixing them up leads to thinking you are covered when you are not:

Kind	Looks at	Example tools
SAST	your source code	CodeQL, Semgrep
SCA	your dependencies	Dependabot, Trivy
Secret scanning	committed keys	Gitleaks, TruffleHog
DAST	the running system	OWASP ZAP

8. CodeQL came to GitHub through the acquisition of Semmle in September 2019. Dependabot came through the acquisition of Dependabot in May 2019. Both are now built into the platform.
9. Software bills of materials have two mainstream formats: SPDX, whose version 2.2.1 became ISO/IEC 5962:2021, and CycloneDX, standardized as ECMA-424 in

2024. Producing one is now a common late pipeline stage.

10. Flakiness, with a real figure: Google's testing blog reported in May 2016 that almost 16 percent of their roughly 4.2 million tests showed some level of flakiness, and that flaky failures consumed a substantial share of engineering attention. Treat the exact percentage as specific to Google, and the phenomenon as universal.
11. Standard flakiness controls, in increasing order of honesty: retry the test and hope; quarantine it into a non-blocking suite and file a bug; fix the root cause, which is nearly always shared state, real time, real network, or unordered iteration.
12. Retrying without quarantining is the trap. It hides the signal and lets the flake rate grow until the suite is useless.

13. Required versus optional checks, restated in terms of 41.8: a required check must appear and must report success. An optional check appears and is advisory, and produces `UNSTABLE` rather than `BLOCKED` when it fails.
14. A workable policy: require the gate job, the build, and the unit tests. Leave slow end-to-end suites, performance benchmarks and advisory linters optional, and review them on a schedule instead.
15. Test time budgets, as targets rather than laws:

Stage	Budget	If exceeded
Lint plus build	under 3 min	cache, incremental build
Unit tests	under 5 min	shard across runners
Full PR pipeline	under 10 min	move work to nightly
Nightly full matrix	under 2 h	reduce matrix cells

16. The four DORA metrics from the 2018 book Accelerate by Nicole Forsgren, Jez Humble and Gene Kim are the standard way to judge whether the pipeline is actually helping: deployment frequency, lead time for changes, change failure rate, and time to restore service. Note that two of the four are about recovery, not prevention.
17. Observing tools: `gh run view --log`, the timing breakdown on a run page, `gh api repos/OWNER/REPO/actions/cache/usage`, and any test reporter that emits JUnit XML for trend analysis.

41.11.6 WORDS - remember these

1. Deployment pipeline — the ordered set of checks from commit to production — the staged model set out in Continuous Delivery, 2010.
2. Cache — a recreatable speed-up — keyed storage of dependencies, safe to lose, never a source of truth.
3. Artifact — the built product you will ship — a retained output passed between jobs and to deployment, built once.
4. Matrix build — the same job across configurations — a cross product of variables expanded into parallel jobs.
5. Fail-fast — cancel siblings on first failure — a matrix strategy trading diagnostic completeness for speed and cost.
6. Flaky test — passes and fails on identical code — a non-deterministic test, usually caused by shared state, timing or ordering.
7. SAST, SCA, DAST — code scan, dependency scan, running-system scan — static analysis, software composition analysis, dynamic testing.
8. Test pyramid — many small tests, few large ones — the proportional model named by Mike Cohn in 2009.

41.12 Deployment strategies

41.12.1 PLAIN - in simple words

1. You have a package that passed everything. Now you must put it in front of real users. How you do that is the deployment strategy.
2. **Recreate**: stop the old version, start the new one. Simple, and there is a gap where nothing works.
3. **Rolling**: you have ten copies running. Replace them a few at a time. No gap, but for a while both versions are live at once.
4. **Blue-green**: run two complete environments. Blue is serving users. Deploy to green, test it, then switch all traffic to green in one moment. Blue stays there, ready to take traffic back.
5. **Canary**: send 1 percent of users to the new version. Watch the error rate. If it looks fine, go to 5 percent, then 25, then everyone.

6. **Feature flags**: ship the new code to everyone but keep it switched off, then switch it on for chosen users later, without deploying anything.
7. That last one is the important idea, so say it clearly.
8. Deploying is putting code on machines. Releasing is letting users reach it.
9. Feature flags separate those two things, so a deploy stops being a scary event and becomes routine.
10. And turning a feature off is instant, while undoing a deploy is not.

41.12.2 PLAIN – a picture in your head

1. Picture changing the menu at a restaurant with a hundred tables.
2. Recreate: close the restaurant, change everything, reopen. Everyone waits outside.
3. Rolling: change the menus table by table. Nobody waits, but for twenty minutes two menus exist and the kitchen must cook from both.
4. Blue-green: build an identical second dining room, set it up completely, check it, then walk everyone across in one go. Keep the old room furnished for a week in case.
5. Canary: give the new menu to two tables, watch what they send back, then widen it.
6. Feature flags: print every dish on every menu from the start, but mark the new ones "ask the waiter". Turning them on is a word to the staff, not a reprint.

Where this comparison breaks: menus have no state. Software has databases, and a database cannot be walked across to a new room. That single difference is what makes rollback hard, and it is the subject of 41.12.4.

41.12.3 PLAIN – a worked example

1. A service with 12 identical copies, a 90-second start time, and a rolling update replacing 25 percent at a time.
2. 12 copies, 25 percent surge means 3 new copies start while at most 3 old ones go away.
3. Four waves of 3, each about 90 seconds plus a health check, is roughly 7 to 8 minutes to fully roll out.
4. During those 8 minutes, both versions serve real traffic. Both must be able to read the same data and answer the same requests.
5. Compare the same release under four strategies:

Strategy	Downtime	Rollback speed
Recreate	60 to 120 s	full redeploy
Rolling	none	another full roll
Blue-green	none	seconds, flip back
Canary	none	seconds, shift traffic

6. Blue-green and canary buy fast rollback with double the running cost during the change. That is the trade, stated honestly.
7. A Kubernetes rolling update is configured with two numbers:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 25%
    maxUnavailable: 25%
```

8. `maxSurge` is how many extra copies may exist above the target. `maxUnavailable` is how many may be missing. Both default to 25 percent.
9. Setting `maxUnavailable: 0` guarantees no reduction in capacity and requires spare room to surge into.

41.12.4 PLAIN - what is really happening inside

1. Rollback sounds symmetric. It is not.
2. Code rollback is easy: the old package still exists, so put it back.
3. Data rollback is often impossible, because the new version has already written data in a new shape.
4. Suppose the release renamed a column from `phone` to `phone_number` and copied the values across.
5. New code writes to `phone_number`. Users create records for twenty minutes.
6. Now roll the code back. The old code reads `phone`, which no longer receives writes. Those twenty minutes of phone numbers are invisible.
7. Nothing crashed. The data is simply wrong, which is worse.
8. The discipline that fixes this is called expand and contract, or parallel change, and it takes three releases instead of one.
9. **Expand**: add `phone_number` while keeping `phone`. Write to both. Read from `phone`. Deploy. This release is safe to roll back.
10. **Migrate**: switch reads to `phone_number`. Still writing both. Deploy. Still safe to roll back.
11. **Contract**: stop writing `phone`, then drop it, in a later release, once you are certain you will not roll back past this point.
12. The rule underneath: every deployment must be compatible with the version before it and the version after it, because during any rolling update both are running at once.
13. That rule is why database changes are the hard part of continuous deployment, and why teams that ignore it get exactly one bad night and then adopt it.

41.12.5 TECHNICAL - the engineer's version

1. Blue-green deployment as a named practice was popularized by Dan North and Jez Humble and set out in Continuous Delivery in 2010. Canary releasing takes its name from the caged birds carried into coal mines to detect gas.
2. Feature toggles were described in detail in Pete Hodgson's 2016 article Feature Toggles on Martin Fowler's site, which classifies them into release toggles, experiment toggles, ops toggles and permission toggles, with very different lifetimes.
3. Flickr's 2009 engineering post Flipping Out is the widely cited early description of running continuous deployment behind flags, with more than ten deploys a day.
4. Strategy comparison with the properties that actually decide the choice:

Strategy	Extra capacity	Both versions live
Recreate	none	no
Rolling	small surge	yes, briefly
Blue-green	100 percent	at the switch only
Canary	small	yes, for a long time

5. Canary requires observability that most teams underestimate: per-version metrics. If your error rate is not broken down by build, a canary tells you nothing you did not already know.
6. Typical canary schedule: 1 percent for 10 minutes, 5 percent for 20 minutes, 25 percent for 30 minutes, then 100 percent, with automatic rollback on any guard metric breach. Guard metrics are usually error rate, latency at the 99th percentile, and a business counter such as checkouts per minute.
7. Kubernetes supports `Recreate` and `RollingUpdate` natively; blue-green and canary are built on top with service selectors, ingress weights, or a service mesh. Argo Rollouts and Flagger are the common controllers for this.
8. Expand and contract is documented as the ParallelChange pattern by Danilo Sato in 2014. Its formal requirement is that every schema state supports both the previous and the next application version, sometimes called N-1 compatibility.

9. Migrations that are not reversible in practice: dropping a column, dropping a table, narrowing a type, adding a NOT NULL constraint without a default, and any data transformation that loses the original value. Plan these as separate releases, never bundled with a behaviour change.
10. GitHub Actions models targets as **environments**, which carry their own secrets and variables plus protection rules:

Protection	Effect
Required reviewers	up to 6 approvers
Wait timer	delay up to 43,200 min
Branch policy	only named branches deploy
Custom rules	third-party gates

11. 43,200 minutes is 30 days, the documented maximum wait timer. A wait timer is a real control: it is the mechanical version of "let it bake in staging overnight".
12. A job binds to an environment with `environment: production`, at which point the environment's secrets become available and its protection rules apply before the job starts, not during it.
13. GitLab expresses the same idea with `environment:` plus `when: manual`, and protected environments with approval rules. Argo CD expresses it as a sync policy, manual or automated, per application.
14. Observing tools: `gh api repos/OWNER/REPO/deployments`, `gh api repos/OWNER/REPO/environments`, `kubectl rollout status`, `kubectl rollout undo`, and per-version dashboards in your metrics system.

41.12.6 WORDS - remember these

1. Recreate — stop the old, start the new — a strategy with a deliberate outage window and no version overlap.
2. Rolling update — replace instances a few at a time — controlled by surge and unavailability limits, with both versions live.
3. Blue-green — two full environments, one switch — instant cutover and instant rollback at double running cost.
4. Canary — a small slice of traffic first — progressive delivery gated on per-version guard metrics.
5. Feature flag — code shipped but switched off — runtime toggle decoupling deploy from release, reversible in milliseconds.
6. Expand and contract — change the schema in compatible steps — the ParallelChange pattern giving N-1 compatibility across releases.
7. Environment — a named deployment target with rules — a scope holding secrets, reviewers, wait timers and branch policies.

41.13 Secrets in CI

41.13.1 PLAIN - in simple words

1. A pipeline needs passwords. It signs in to a registry, uploads to a server, talks to a cloud account.
2. Those values cannot live in the repository, because everyone who can read the code can read them, forever, in every clone.
3. So they are stored separately, in a box the service keeps locked, and handed to the machine only while the job runs.
4. Inside the job they appear as environment variables, which are just named values the running program can read.
5. When the job ends, the machine is destroyed and the values go with it.
6. Two rules matter more than everything else.
7. Rule one: never print a secret. Not to the log, not in an error message, not in a debug dump.

8. The service tries to help by blacking out anything matching a known secret, but that is text matching. Change the text even slightly and the blackout fails.
9. Encode a key in base64 and print it, and it goes to the log in full, because the blacked-out pattern no longer matches.
10. Rule two: a stranger's pull request gets no secrets at all.
11. That is why a build from an outside contributor can test the code and cannot publish anything.
12. And the modern advice goes further: stop storing long-lived cloud passwords at all.
13. Instead the job proves who it is, and the cloud hands it a temporary key that expires in an hour.
14. Nothing long-lived exists to steal, which changes the shape of the whole problem.

41.13.2 PLAIN – a picture in your head

1. Think of a building that contractors visit.
2. The old way: give each contractor a copy of the master key. It works forever, it can be copied, and if one is lost nobody knows.
3. The new way: the contractor shows an identity card at reception. Reception checks it against a list, and issues a visitor badge valid for one hour, for one floor.
4. The badge cannot be reused tomorrow. Losing it costs almost nothing.
5. Reception never gives out the master key, because there is no need.
6. That is the difference between a stored cloud password and identity federation.

Where this comparison breaks: the contractor's identity card is a physical thing they carry. In continuous integration the identity is issued fresh by the platform for each job, and its trustworthiness rests entirely on the platform not lying about which repository is asking. So you must pin the trust rule down to an exact repository and branch, or any repository on the platform can walk in with a valid card.

41.13.3 PLAIN – a worked example

1. The wrong way, and it is written like this in real repositories:

```
      - run: |
          echo "Token is ${{ secrets.API_TOKEN }}"
          curl -H "Authorization: Bearer $TOKEN" ...
```

2. Two faults. The value is interpolated into the script text, so it can appear in an error trace. And it is printed on purpose.
3. The right way passes secrets through the environment, never into the script body:

```
      - env:
          API_TOKEN: ${{ secrets.API_TOKEN }}
        run: |
          curl -sS -H "Authorization: Bearer $API_TOKEN" \
               "$API_BASE/v1/publish"
```

4. Now the shell receives a variable name. The value never becomes part of the command text that appears in logs and traces.
5. The identity-federation version needs no stored password at all:

```
    permissions:
      id-token: write
      contents: read
    steps:
      - uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::111122223333:role/ci
          aws-region: ap-south-1
```

6. There is no `secrets.AWS_SECRET_ACCESS_KEY` anywhere. The `id-token: write` permission lets the job ask GitHub for a signed identity document, and the cloud exchanges it for credentials lasting about an hour.
7. Pinning a third-party action to an exact commit, with the version in a comment so tools can still update it:

```
      - uses: actions/checkout@11bd719  # v4.2.2
```

8. That short form is for the page. In a real file, write the full 40-character hash, because a tag can be moved and a full hash cannot.

41.13.4 PLAIN - what is really happening inside

1. A secret is encrypted before it leaves your browser, stored encrypted, and decrypted only at the moment a job that is allowed to see it starts.
2. The service also registers each secret value with the log processor, which scans every line and replaces exact matches with asterisks.
3. That masking is genuinely useful and genuinely shallow. It matches strings.
4. It does not know that `base64 -w0 key.pem` produces a different string containing the same secret. It does not know that printing a JSON object containing the key is the same disclosure.
5. This is exactly how the March 2025 attack on a popular third-party action worked: the malicious code read secrets out of the runner process and printed them in an encoded form, and masking did not catch it.
6. Fork pull requests get no secrets because the code being tested is not yours. Section 41.5 covered why, and this is the other half of the same rule.
7. For identity federation the sequence is short and worth knowing.
8. The job asks GitHub for a signed token describing itself: which repository, which branch, which environment, which workflow.
9. The cloud provider has been configured in advance to trust tokens signed by GitHub, but only when the description matches a rule you wrote.
10. If it matches, the cloud issues short-lived credentials. If it does not, the request is refused and nothing is leaked.
11. The whole exchange takes under a second and leaves nothing behind.

41.13.5 TECHNICAL - the engineer's version

1. GitHub encrypts secrets client-side with a libsodium sealed box before upload, stores them encrypted, and decrypts them only when injecting into a runner. Documented limits as of August 2026: 48 KB per secret, 1,000 organization secrets, 100 repository secrets, 100 environment secrets.
2. A value larger than 48 KB must be encrypted into a file committed to the repository, with only the passphrase stored as a secret. This is the documented workaround, not a hack.
3. Masking is implemented by registering values with the log service. You can register more at runtime with the workflow command `::add-mask::value`, which is how you protect a secret derived inside the job.
4. Masking limitations, all real: it does not survive transformation such as base64 or URL encoding, it does not apply to values split across lines, and it does not protect artifacts, only logs.
5. Scoping: repository secrets, organization secrets with repository allow lists, and environment secrets which are only available to jobs bound to that environment and only after its protection rules pass.
6. `GITHUB_TOKEN` is an automatically issued installation token, valid for the duration of the job, invalidated at the end. Declare `permissions:` at workflow or job level and grant the minimum. For newly created repositories and organizations GitHub changed the default to read-only in 2023; older ones may still default to write, so check rather than assume.
7. OIDC federation, precisely. GitHub Actions gained support in a changelog entry on 27 October 2021 and was declared generally available in November

2021.

8. The job requests a JSON Web Token from the endpoint in `ACTIONS_ID_TOKEN_REQUEST_URL`, authenticating with `ACTIONS_ID_TOKEN_REQUEST_TOKEN`, both injected only when `permissions: id-token: write` is set.
9. The issuer is `https://token.actions.githubusercontent.com`. The claim that matters is `sub`, whose shape is one of:

```
repo:OWNER/REPO:ref:refs/heads/main
repo:OWNER/REPO:environment:production
repo:OWNER/REPO:pull_request
```

10. On AWS the exchange is `sts:AssumeRoleWithWebIdentity` against an IAM OIDC identity provider, producing credentials of at most one hour by default. On Azure it is a federated credential on an app registration. On Google Cloud it is Workload Identity Federation.
11. The critical misconfiguration is a trust policy that checks the issuer but not the `sub` claim, or checks it with a wildcard. That configuration lets any repository on GitHub assume your role. Always match the full `sub`, including the ref or environment.
12. Third-party action supply chain risk, with a real case. In March 2025 the action `tj-actions/changed-files` was compromised: its version tags were retargeted to a malicious commit that read secrets out of runner memory and printed them into the workflow log.

Fact about that incident	Value
Advisory	CVE-2025-30066
Affected versions	up to 45.0.7
Related compromise	CVE-2025-30154
United States CISA alert	18 March 2025

13. The mechanism deserves attention: nothing was pushed to your repository. A tag in someone else's repository was moved. `uses: owner/action@v4` resolves that tag at run time, on every run, with no record of what it pointed at yesterday.
14. Pinning to a full 40-character commit SHA removes that entire class of attack, because a SHA names immutable content. Moving a tag cannot change what a SHA resolves to.

Reference form	Mutable	Verdict
`@main`	yes, any push	never
`@v4`	yes, tag moves	avoid
`@v4.2.2`	yes, tag moves	avoid
`@11bd719...` full SHA	no	correct

15. Pinning has a cost: you stop receiving security fixes automatically. Dependabot understands SHA pins with a trailing version comment and raises pull requests to move them, which restores updates while keeping the review step.
16. Earlier incidents in the same family, worth knowing as precedent: the Codecov bash uploader compromise in April 2021, where a modified script exfiltrated environment variables from customers' CI jobs, and the SolarWinds Orion build system compromise disclosed in December 2020.
17. Other controls that matter: run untrusted work on ephemeral, isolated runners; never use self-hosted runners on public repositories; set `permissions: {}` and add back only what a job needs; and enable secret scanning with push protection so a committed key is rejected at push time.

18. Observing tools: `gh secret list`, `gh api repos/OWNER/REPO/actions/permissions`, `gh api repos/OWNER/REPO/actions/permissions/selected-actions`, and a repository policy allowing only actions from verified creators or a pinned allow list.

41.13.6 WORDS - remember these

1. Secret — a value the pipeline needs and must not reveal — encrypted at rest, injected as an environment variable for one job.
2. Masking — blacking out secret values in logs — exact string replacement by the log processor, defeated by any encoding.
3. `::add-mask::` — register a value for masking at runtime — a workflow command for secrets derived inside a job.
4. OIDC federation — proving identity instead of holding a key — a signed JWT exchanged for short-lived cloud credentials.
5. `sub` claim — which repository, branch or environment is asking — the field the cloud trust policy must match exactly.
6. `id-token: write` — permission to request an identity token — required before the OIDC endpoint variables are injected.
7. SHA pinning — naming an action by immutable content — `uses: owner/act@<40 hex>`, immune to tag retargeting.
8. Supply chain attack — compromising something you depend on — as in the March 2025 `tj-actions/changed-files` incident, CVE-2025-30066.

41.14 Reading a failed run like an engineer

41.14.1 PLAIN - in simple words

1. A run went red. Most people open it, read the last line, and guess.
2. The last line is almost never the fault. It is usually the summary of the fault, several steps downstream.
3. Do this instead, in order.
4. Find which **step** failed. Not which job, not which line. The step.
5. Read that step from its beginning, not from its end.
6. The real message is usually in the first few lines after the step starts, and everything below it is consequence.
7. Ask one question before anything else: does this fail on my machine too?
8. If it fails locally, this is a normal bug and continuous integration did its job. Fix it and stop reading.
9. If it passes locally and fails in the pipeline, you have a different problem, and guessing will waste an hour.
10. Then the difference between the two environments is the fault, and there is a short list of things that difference can be.
11. Work through that list in order rather than changing things at random.
12. And if you still cannot see it, re-run with debug output switched on, which prints what the machine was actually doing.

41.14.2 PLAIN - a picture in your head

1. A car will not start in your driveway but starts fine at the garage.
2. A bad mechanic replaces parts until it works.
3. A good mechanic writes down every difference between the driveway and the garage: colder, on a slope, different fuel, different battery charge.
4. Then tests the differences one at a time.
5. The car is the same. The environment is not. The fault lives in the gap.

Where this comparison breaks: you cannot drive a build to the garage, but you can bring the garage to the build. A pipeline runs a published machine image, so you can start the identical image locally, which is a luxury no mechanic has.

41.14.3 PLAIN - a worked example

1. Getting the logs without opening a browser:

```
# which runs failed recently
gh run list --status failure --limit 5

# the whole log, all jobs, all steps
gh run view 11947722301 --log

# only the failing steps, which is what you want first
gh run view 11947722301 --log-failed

# one job's log
gh run view --job 33417290011 --log
```

2. A real failure the reader would meet, coming from macOS locally and Ubuntu in the pipeline:

```
Run npm test
> jest
 FAIL  src/Cart/index.test.ts
  Cannot find module './cartTotal' from 'src/Cart/index.ts'
```

3. The file on disk is `src/Cart/CartTotal.ts` and the import says `./cartTotal`.
4. On macOS this works, because the default file system is case-insensitive. On the Linux runner it does not, because ext4 is case-sensitive.
5. Nothing is wrong with the test, the runner, or the cache. The two machines disagree about whether two file names are the same name.
6. Confirming it without guessing:

```
git ls-files src/Cart | head
# src/Cart/CartTotal.ts     <- what git actually stores
grep -rn "cartTotal" src/Cart/index.ts
```

7. Git stores the name it was given. Your local file system hid the mismatch. The runner did not.

41.14.4 PLAIN - what is really happening inside

1. Debug output is not on by default because it is enormous and it can leak information.
2. Two switches turn it on, both set as repository or environment variables, or as secrets, with the value `true`.
3. `ACTIONS_STEP_DEBUG` makes each step print its internal decisions.
4. `ACTIONS_RUNNER_DEBUG` makes the runner agent itself print how it set up the job, which is what you need when the failure happens before your code runs.
5. With `gh`, `gh run rerun 11947722301 --debug` sets these for one re-run, which is safer than leaving them on.
6. Reproducing locally has three levels, in increasing fidelity.
7. Level one: run the same command with the same environment variable, since many tools change behaviour when `CI=true` is set.
8. Level two: run the same command inside the same container image the pipeline uses, so the operating system and installed versions match.

9. Level three: run the whole workflow locally with a tool such as `act`, which interprets the workflow file and runs the jobs in containers. It is an approximation, not the real service, and it will differ on some actions.
10. One more trick for the truly stuck: a step that opens an interactive session on the runner, so you can look around while the job is paused. Use it on private repositories only, and never on a job holding production secrets.

41.14.5 TECHNICAL - the engineer's version

1. Log structure. A run contains jobs; a job contains steps; each step is a log group. The interface collapses groups, so the failing step may be closed and invisible until you expand it. `--log-failed` filters to failing steps and is the correct first command.
2. Read the step from the top. A stack trace prints the deepest frame first in some languages and last in others, and the actual message is at the boundary between your code and the library.
3. Exit code 1 is a generic failure. Some codes are informative: 127 is command not found, 126 is found but not executable, 137 is SIGKILL which on a runner almost always means the out-of-memory killer, and 143 is SIGTERM which usually means a cancellation or a timeout.
4. Ordered checklist for a job that fails only in CI. Work down it; do not skip.

#	Check	Command or setting
1	Same commit?	`gh run view --json headSha`
2	Reproduces locally?	run the same command
3	OS differs?	`uname -a`, runner label
4	Tool versions differ?	`node -v`, `python -V`
5	Case sensitivity?	`git ls-files` vs disk
6	File missing?	check `.gitignore`
7	Env var missing?	`env | sort` in a step
8	Secret missing?	fork PRs get none
9	Locale or timezone?	CI runs UTC, C locale
10	Network blocked?	egress rules, proxies
11	Test order or parallelism?	run with a fixed seed
12	Clean checkout?	no untracked local files
13	Stale cache?	change the cache key
14	Resource limits?	memory, disk, `df -h`

5. Rows 5, 6, 9 and 12 are the four that catch most people, and all four have the same root cause: your working copy contains state that the repository does not, or your machine normalizes something the runner does not.
6. Case sensitivity in detail. macOS APFS is case-insensitive but case-preserving by default; Linux ext4 and xfs are case-sensitive; Windows NTFS is case-insensitive. Git stores the exact name. A rename that only changes case may not even be recorded unless you use `git mv` explicitly or set `core.ignorecase false`.
7. Time and locale. Hosted runners run in UTC with a minimal locale. A test that formats a date, sorts strings, or parses a decimal comma will behave differently from a laptop set to Asia/Kolkata. Pin `TZ` and `LC_ALL` explicitly in the workflow rather than hoping.
8. Diagnostic step worth adding permanently to a slow-to-debug pipeline:

```
      - name: Environment fingerprint
        if: always()
        run: |
          uname -a
          node -v || true
```

```
          python3 -V || true
          echo "TZ=$TZ LANG=$LANG"
          df -h /
          free -m || vm_stat || true
          git rev-parse HEAD
```

9. Always upload evidence on failure. A failing run that kept no test report, no screenshot and no server log costs a second run to learn anything:

```
      - uses: actions/upload-artifact@v4
        if: failure()
        with:
          name: failure-evidence-${{ github.run_id }}
          path: |
            reports/
            logs/
            screenshots/
```

10. `if: failure()` and `if: always()` are the two conditions that matter here. Without one of them the upload step is skipped, because a failed previous step stops the job by default.
11. Retention and download: artifacts are downloadable with `gh run download 11947722301`, and logs with `gh run view --log > run.txt` for grepping.
12. Debug variables: `ACTIONS_STEP_DEBUG` and `ACTIONS_RUNNER_DEBUG`, both set to `true`, as repository variables or secrets. `gh run rerun ID --debug` enables them for one attempt. Debug logs are larger and are retained the same way as normal logs.
13. Local reproduction with `act` from the `nektos/act` project runs workflows in Docker containers. It approximates hosted runners; differences in preinstalled tooling, services and some actions are expected. Treat a green `act` run as encouraging, not as verification.
14. GitLab's equivalents: the job log with `CI_DEBUG_TRACE: "true"` for shell tracing, artifacts with `when: always`, and `gitlab-runner exec` for local execution. Jenkins exposes the console output plus the pipeline steps view.
15. One caution on `CI_DEBUG_TRACE` and on `set -x`: shell tracing prints every expanded command, and every expanded command may contain a secret. Never enable shell tracing on a job that holds credentials.

41.14.6 WORDS - remember these

1. Step log — the output of one command in a job — a collapsible log group, filterable with `gh run view --log-failed`.
2. `ACTIONS_STEP_DEBUG` — extra detail about each step — a variable set to `true`, enabling verbose action and expression logging.
3. `ACTIONS_RUNNER_DEBUG` — extra detail about job setup — runner-agent level logging, for failures before your code runs.
4. Exit code 137 — the process was killed — SIGKILL, on a runner almost always the out-of-memory killer.
5. Case sensitivity — whether two names are the same name — differs between APFS, NTFS and ext4; Git stores the exact stored name.
6. `if: always()` — run this step even after a failure — the condition that makes failure evidence get collected.
7. `act` — run workflows on your own machine — a local interpreter of workflow files using Docker, approximate rather than exact.

41.15 The wider landscape

41.15.1 PLAIN - in simple words

1. Every idea in this chapter exists in every tool. Only the words change.
2. **GitHub Actions** lives inside GitHub. You write YAML in your repository and the machines are supplied for you.
3. **GitLab CI** lives inside GitLab and works the same way, except GitLab is also the issue tracker, the registry and the deployment tool, all one product.
4. **Jenkins** is a program you install on your own server. It is older than the others, it does anything, and you maintain all of it yourself.
5. **CircleCI** is a service that only does pipelines. It is not your code host, so you connect it to one.
6. **Buildkite** splits the job in an unusual way: they run the web interface and the queue, you run the machines.
7. **Argo CD** does something different from all of the above. It does not build. It watches a repository describing what should be running, and makes the real system match it.
8. That last idea has a name: **GitOps**.
9. In GitOps, the repository is the statement of what should be true, and a program inside your system continuously fixes any difference.
10. Nobody deploys by pushing. Everybody deploys by changing a file and merging it.
11. The difference matters most for keys. If nothing pushes into your system, your pipeline never needs credentials to it.

41.15.2 PLAIN - a picture in your head

1. Two ways to keep a garden the way you want it.
2. Push: you walk in with tools whenever you decide, and change things. You need a key to the gate, and so does anyone helping you.
3. Pull: you pin a drawing of the intended garden on the gate. A gardener who already lives inside reads the drawing every few minutes and fixes anything that differs.
4. Nobody outside needs a key. The drawing is public and versioned. Every change is an edit to the drawing.
5. If a storm knocks a plant over, the gardener notices and rights it, without anyone asking.

Where this comparison breaks: a gardener can only fix what the drawing describes. GitOps only reconciles resources under its control, so anything created by hand outside that scope drifts unnoticed. And some changes, like data migrations, are not describable as a desired end state at all.

41.15.3 PLAIN - a worked example

1. The same idea, "run the tests", in four configuration languages.

```
# GitHub Actions: .github/workflows/ci.yml
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: npm ci && npm test
```

```
# GitLab CI: .gitlab-ci.yml
test:
  image: node:22
  script:
    - npm ci
    - npm test
```

```
// Jenkins: Jenkinsfile
pipeline {
  agent { docker { image 'node:22' } }
  stages {
    stage('test') {
      steps { sh 'npm ci && npm test' }
    }
  }
}
```

```
# CircleCI: .circleci/config.yml
version: 2.1
jobs:
  test:
    docker: [{ image: cimg/node:22.0 }]
    steps:
      - checkout
      - run: npm ci && npm test
```

2. Four files, one idea: check out the code, get a Node environment, run two commands.
3. The differences that actually matter are not syntax. They are who owns the machines, who owns the upgrades, and what happens when you need something unusual.

41.15.4 PLAIN - what is really happening inside

1. Push deployment and pull deployment are the real architectural fork, and every tool sits on one side of it.
2. In push deployment, the pipeline holds credentials to production and reaches in to change it.
3. That is simple, immediate, and means a compromised pipeline is a compromised production system.
4. In pull deployment, an agent runs inside production, reads a repository, and applies the difference.
5. Nothing outside holds production credentials. The pipeline's only job is to update a file in a repository.
6. The agent also loops forever, so it repairs drift: someone changing something by hand is silently undone within minutes.
7. That loop is the part people underestimate. It converts deployment from an event into a continuously enforced statement.
8. The cost is indirection. You merge a change and then wait for a program you are not watching to notice it, which is harder to debug than a command that either worked or did not.

41.15.5 TECHNICAL - the engineer's version

1. Comparison on the three axes that decide the choice.

Tool	Hosting	Config	Strength
GitHub Actions	hosted or self	YAML	ecosystem, no setup
GitLab CI	hosted or self	YAML	one integrated product
Jenkins	self-hosted	Groovy	plugins, total control
CircleCI	mainly hosted	YAML	caching, orbs, speed
Buildkite	hybrid	YAML	your hardware, their UI
Argo CD	in your cluster	Git manifests	reconciliation, drift

2. Jenkins. Started as Hudson by Kohsuke Kawaguchi at Sun Microsystems in 2004, first released 2005, renamed Jenkins after a community vote concluded on 29 January 2011. Jenkins 2.0, released in April 2016, introduced Pipeline as code with the `Jenkinsfile` in a Groovy-based domain-specific language.
3. Jenkins's strength and weakness are the same fact: well over a thousand plugins. Anything is possible, and the plugin set is your responsibility to patch, test and upgrade forever.

4. GitLab CI. Began in 2012 as a separate application and was merged into GitLab in version 8.0 in September 2015. Uses `.gitlab-ci.yml` with stages, and a separate `gitlab-runner` binary with pluggable executors.
5. CircleCI. Launched 2011, config in `.circleci/config.yml`, with reusable packages called orbs introduced in November 2018. CircleCI disclosed a security incident in January 2023 and instructed customers to rotate all secrets stored in the platform, which is a useful reminder that a hosted CI provider is part of your trust boundary.
6. Buildkite. From Melbourne, Australia, starting in 2013. The distinguishing model is hybrid: Buildkite hosts the control plane and interface, and you run the agents on your own machines, so source code and secrets stay on your infrastructure.
7. Argo CD. Created at Applatix, which Intuit acquired in January 2018. Argo became a CNCF project and graduated, together with Flux, in December 2022.
8. GitOps as a term was coined by Alexis Richardson of Weaveworks in 2017. The OpenGitOps project states four principles: the desired state is declarative; it is versioned and immutable; agents pull it automatically; and agents continuously reconcile actual state towards it.
9. Flux is the other main implementation, also CNCF-graduated. Weaveworks, the company that created Flux and coined the term, ceased trading in early 2024, and the project continued under the foundation. That separation of project from vendor is exactly why foundation governance matters.
10. Others worth recognizing by name: Travis CI, which popularized in-repository configuration from 2011; TeamCity from JetBrains, 2006; Bamboo from Atlassian; Azure Pipelines; AWS CodeBuild and CodePipeline; Google Cloud Build; Tekton, a Kubernetes-native pipeline engine under the CD Foundation; and Concourse, which models everything as resources and tasks.
11. The push versus pull distinction, in security terms:

Model	Credentials live	Blast radius
Push from CI	in the CI system	CI owns production
Pull by agent	inside the cluster	CI owns a Git repo

12. Combining them is the common modern arrangement: continuous integration builds and tests and publishes an image, then updates an image tag in a configuration repository; a cluster agent notices and rolls it out. The pipeline never holds a cluster credential.
13. Portability, honestly stated: pipeline configuration is not portable between these tools, and every migration is a rewrite. What is portable is the shell script your steps call. Teams that keep the real work in scripts and use the YAML only as an orchestrator migrate in days rather than months.
14. Where experts disagree: self-hosted Jenkins for control, or a hosted service for speed. Control genuinely matters in regulated environments and on unusual hardware. Against it, maintaining Jenkins is a permanent part-time job nobody plans for. The deciding factor is whether you have staff to own it.
15. Observing tools across all of them: `gh run`, `glab ci`, the Jenkins `api/json` endpoint per build, `circleci config validate`, `buildkite-agent` logs, and `argocd app get` plus `argocd app diff` for reconciliation state.

41.15.6 WORDS - remember these

1. GitOps — the repository states what should be running — declarative desired state, pulled and continuously reconciled by an in-cluster agent.
2. Reconciliation loop — keep fixing the difference forever — the controller pattern comparing desired state to actual state on a timer.
3. Drift — reality quietly diverging from the file — manual changes to a system under declarative control, reverted on the next reconcile.
4. Push deployment — the pipeline reaches into production — outbound deploys requiring production credentials in the CI system.

5. Pull deployment — production reaches out for its own config — an agent inside the boundary, so no inbound credentials exist.
6. Orb — CircleCI's reusable configuration package — a versioned bundle of jobs, commands and executors, introduced in 2018.
7. `Jenkinsfile` — Jenkins pipeline as code — a Groovy-based declarative or scripted pipeline definition stored in the repository.

41.98 Common wrong ideas

1. Wrong: continuous integration is a tool you install. Right: it is a habit of merging to the mainline at least daily; the tool only enforces the habit, and a team merging quarterly with Jenkins running is not doing it.
2. Wrong: the green tick means this code is good. Right: it means the checks somebody wrote passed on some commit. It says nothing about the tests that were never written, and nothing about the code you are looking at now.
3. Wrong: a passing branch means the head commit passed. Right: a run is bound to one `head_sha`. A branch is a moving label, so a branch-level tick may belong to a commit that is no longer the head.
4. Wrong: CD means continuous deployment. Right: it usually means continuous delivery, which is "always releasable, released by a human decision". Continuous deployment removes the human. The two are different practices with the same initials.
5. Wrong: `pull_request_target` is just `pull_request` with more permissions. Right: it runs your trusted workflow file with your secrets, so checking out the pull request head and executing anything from it hands a stranger your credentials.
6. Wrong: `mergeStateStatus: BLOCKED` means there is a merge conflict. Right: it means the branches merge cleanly and a protection rule is unmet. The conflict value is `DIRTY`.
7. Wrong: `UNKNOWN` means something is wrong. Right: it means GitHub has not finished computing mergeability. Re-query after a short delay and conclude nothing from the first answer.
8. Wrong: a skipped check is the same as a passed check. Right: a job skipped by an `if:` condition reports success, while a workflow skipped by a path filter never reports at all and blocks the merge indefinitely.
9. Wrong: pinning an action to `@v4` is safe because it is a version. Right: a tag is a movable pointer in someone else's repository. Only a full commit SHA is immutable, as the March 2025 `tj-actions/changed-files` compromise showed.
10. Wrong: re-running until it goes green is a fix. Right: it converts a flaky test into a permanent lie, and once people stop believing red, the entire pipeline stops carrying information.

41.99 Chapter summary in 20 lines

1. Continuous integration is a practice, not a product: merge to the mainline at least daily, verify every merge automatically, and fix a broken mainline before starting new work.
2. The term came from Grady Booch in 1991, became an Extreme Programming practice in 1999, and got its working definition from Fowler and Foemmel in

2000.

3. Continuous delivery means every change is always releasable; continuous deployment means every change that passes goes live without a human. Same initials, different practices.
4. A runner is a machine that claims a job, checks out one exact commit, runs your steps, and reports an exit code. Hosted runners are destroyed after each job, which is why "it works on my laptop" fails there.
5. The workflow file lives in the repository, so the pipeline is versioned, reviewable and bisectable like any other code, and a token needs the `workflow` scope to change it.
6. Triggers decide what starts a run: `push`, `pull_request`, `schedule`, `workflow_dispatch`, `repository_dispatch`, tags and releases, narrowed by branch and path filters.

7. `pull_request` runs a fork's code with no secrets and a read-only token. `pull_request_target` runs your file with full secrets, so executing the pull request's code under it is the pwn request vulnerability.
8. The central idea of the chapter: a run is bound to one commit SHA, never to a branch. A branch is a moving pointer, so "CI passed on main" is a statement about whichever commit main pointed at then.
9. The correct verification is four steps: resolve the head SHA from the server, list runs filtered by that SHA, read the specific run by ID, and confirm the run's `head_sha` matches.
10. A branch-level tick can lie in at least six ways: older commit, merge result rather than head, skipped rather than passed, posted by another app, overwritten by a re-run, or attached to a merge commit that no longer exists.
11. Two APIs report results on a commit: the older commit status API with four states, and the newer checks API with check suites, check runs, timings and line annotations.
12. Conclusions are `success`, `failure`, `neutral`, `cancelled`, `skipped`, `timed_out`, `action_required`, `stale` and `startup_failure`, and a required check that never arrives stays Expected and blocks forever.
13. Merge queues solve the semantic conflict: two changes that each pass alone and break together. The queue builds the prospective merged state, tests it, and only then merges, serializing the landing order.
14. Speculative execution tests several queued entries at once assuming those ahead succeed, which is fast when all pass and wasteful when one fails. GitHub's merge queue became generally available on 12 July 2023.
15. `mergeStateStatus` has eight values. `CLEAN` and `HAS_HOOKS` mean go, `BEHIND` means update, `DIRTY` means real conflict, `DRAFT` means you marked it, `UNSTABLE` means a non-required check failed.
16. `BLOCKED` means a protection rule is unmet, not a conflict; `UNKNOWN` means GitHub is still computing mergeability, so re-query and conclude nothing.
17. A good pipeline runs cheapest first: lint, build, unit tests, integration tests, security scanning, artifact build, staging deploy, smoke tests, production, with the artifact built once and reused.
18. Deployment strategies trade downtime against cost: recreate, rolling, blue-green, canary, and feature flags, which separate deploying code from releasing it. Database migrations make rollback hard, so expand and contract.
19. Secrets are injected per job and masked by string matching only; forks get none; OIDC federation replaces long-lived cloud keys with one-hour credentials; and third-party actions must be pinned to a full commit SHA.
20. Read a failure by finding the failing step, reading it from the top, reproducing locally, and working an ordered checklist of environment differences before changing anything at random.

42

CHAPTER

KB

SSH, HTTPS, Keys, Tokens and Scopes

42.0 What this chapter gives you

1. You will be able to explain the two ways git talks to a server, SSH and HTTPS, and say exactly what is different and what is identical.
2. You will be able to say what SSH is, who wrote it, in what year, and what dangerous older tools it was built to replace.
3. You will be able to explain a key pair in plain words and then precisely, including what the private key can do that the public key cannot.
4. You will be able to read a real `ssh-keygen` run, decode the public key file field by field, and choose between RSA, ECDSA and Ed25519 with reasons.
5. You will be able to explain `authorized_keys` and `known_hosts`, including why wrong file permissions break logins silently.
6. You will be able to explain why the reader's SSH on port 22 hung while HTTPS on port 443 worked, and apply both fixes.
7. You will be able to explain personal access tokens, their expiry, their prefixes, and why a leaked token is an emergency rather than an annoyance.
8. You will be able to explain a scope precisely, and explain the exact reason a push touching a file under `.github/workflows/` was refused.
9. You will be able to tell a permission failure from a network failure from a rejected push, even though all three print "push failed".
10. You will be able to diagnose at least eight real git authentication error strings from the text alone, and act on each one correctly.

42.1 The two transports git uses, and why there are two

42.1.1 PLAIN - in simple words

1. Git keeps your project history on your own machine. All of it.
2. Sometimes you want to send that history to a server, or fetch new history from one. That is what `push` and `fetch` do.
3. To send anything over a network, two programs need a pipe between them.
4. Git does not build its own pipe. It borrows one that already exists.
5. There are two pipes it commonly borrows, and you choose which one by the shape of the address you type.
6. An address like `git@github.com:owner/repo.git` uses the **SSH** pipe.
7. An address like `https://github.com/owner/repo.git` uses the **HTTPS** pipe, the same kind of pipe your web browser uses.
8. Here is the part people get wrong. The repository on the server is the same repository in both cases. One box, one set of files.

9. Only the road to it changes. It is like posting a letter versus handing it over in person. Same letter, same house, different delivery.
10. So switching from one to the other never loses a commit, never rewrites history, and never needs a re-clone. It is a one-line change.
11. What does change is how the server works out who you are.
12. Over SSH you prove yourself with a key file on your laptop.
13. Over HTTPS you prove yourself with a token, which is a long secret string that behaves like a password with rules attached.

42.1.2 PLAIN - a picture in your head

1. Imagine one library building with one collection of books.
2. There are two doors. The staff door at the back and the public door at the front. Both lead to exactly the same shelves.
3. The staff door has a lock. You carry a physical key that fits it. If your key fits, you walk in. Nobody asks your name.
4. The public door has a desk. You show a card with your name on it and a list of what you are allowed to do. The clerk reads the card and decides.
5. The books do not care which door you used. A book you borrow through the back door is the same book you would have borrowed through the front.
6. Now suppose the alley behind the building is closed for roadworks. The staff door is fine. You simply cannot reach it.
7. You walk round to the front and use the public door. Nothing about the library changed. Your route changed.
8. That is exactly the reader's situation in this chapter, and exactly the fix.

Where this comparison breaks:

9. A real key and a real card are both things you hand over. A private SSH key is never handed over. It stays in your pocket and answers a question.
10. And a library clerk can see your whole card. A well-built server only sees that your token is valid and what it is allowed to do, not your password.

42.1.3 PLAIN - a worked example

1. Here is the same repository addressed both ways, on a real machine.

```
$ git remote add t git@github.com:octocat/Hello-World.git
$ git remote add t https://github.com/octocat/Hello-World.git
$ git remote add t \
    ssh://git@ssh.github.com:443/octocat/Hello-World.git
```

2. All three name one repository: the account `octocat`, the repository `Hello-World`.
3. The first is the short SSH form. The colon after the host name is not a port. It separates the host from the path.
4. The second is HTTPS, and it is an ordinary web address with a `.git` suffix.
5. The third is the long SSH form, written as a proper address with a scheme, which is the only form where you can write a port number.
6. Changing transport on an existing clone is one command:

```
$ git remote set-url origin https://github.com/owner/repo.git
$ git remote -v
origin  https://github.com/owner/repo.git (fetch)
origin  https://github.com/owner/repo.git (push)
```

7. Nothing else changes. Your branches, your commits and your stash are all still local files under `.git`, untouched.
8. Here is a real clone of a real public repository over HTTPS, from the machine used to write this chapter:

```
$ git clone https://github.com/octocat/Hello-World.git
Cloning into 'Hello-World'...
$ git log --oneline -1
7fd1a60 Merge pull request #6 from Spaceghost/patch-1
```

9. Over SSH the commit `7fd1a60` would be byte-identical, because a commit identity is a hash of its content, not of how it travelled.

42.1.4 PLAIN - what is really happening inside

1. Git itself does not know how to speak to a network. It shells out.
2. When you give it an HTTPS address it runs a small helper program called `git-remote-https`. When you give it an SSH address it runs `ssh`.
3. You can watch it choose. Here is a real trace on the machine used for this chapter, with the timestamps trimmed for width:

```
$ GIT_TRACE=1 git ls-remote \
    https://github.com/octocat/Hello-World.git
trace: built-in: git ls-remote https://github.com/...
trace: run_command: git remote-https https://github.com/...
trace: exec: git-remote-https https://github.com/...
```

4. And here is what git runs for an SSH address. This is the real command line git built, wrapped for width:

```
$ GIT_TRACE=1 git ls-remote \
    ssh://git@ssh.github.com:443/octocat/Hello-World.git
trace: run_command: GIT_PROTOCOL=version=2 ssh
  -o SendEnv=GIT_PROTOCOL -p 443 git@ssh.github.com
  'git-upload-pack '\''/octocat/Hello-World.git'\'''
```

5. Read that last line carefully, because it explains everything about SSH and git in one go.
6. Git asked SSH to log in as the user `git` on the host `ssh.github.com`, and then to run one command there: `git-upload-pack`.
7. SSH is not doing anything git-specific. It is doing what it always does: logging in and running a command on a remote machine.
8. The remote command's input and output are piped straight back down the encrypted connection to your local git.
9. Over HTTPS there is no login shell and no remote command. Instead git makes ordinary web requests. Here are the real ones, captured live:

```
GET /octocat/Hello-World.git/info/refs?service=git-upload-pack
Git-Protocol: version=2
<= HTTP/1.1 200 OK
<= Content-Type: application/x-git-upload-pack-advertisement

POST /octocat/Hello-World.git/git-upload-pack
Content-Type: application/x-git-upload-pack-request
<= HTTP/1.1 200 OK
<= Content-Type: application/x-git-upload-pack-result
```

10. One `GET` to ask what branches exist, then one `POST` carrying the real conversation. That is the whole thing.

11. Underneath both, the same objects flow: the same commits, trees, blobs and the same packfile format. The payload is identical.

42.1.5 TECHNICAL - the engineer's version

1. Git supports four transports. Two matter today.

Transport	Port	Authentication
SSH	TCP 22	public key, user `git`
HTTPS smart	TCP 443	token in Basic auth
git:// (dumb)	TCP 9418	none, removed by GitHub
local file path	none	filesystem permissions

2. GitHub removed support for the unencrypted `git://` protocol as part of a protocol hardening programme that became permanent on 15 March 2022.
3. Over SSH, git uses the SSH connection protocol to request a single command execution, not an interactive shell. The remote side runs `git-upload-pack` for fetches and `git-receive-pack` for pushes.
4. The environment variable `GIT_PROTOCOL=version=2` is passed with `-o SendEnv=GIT_PROTOCOL`. Protocol v2 was introduced in Git 2.18 in June 2018 and made the default for fetches in Git 2.26 in March 2020.
5. Protocol v2 matters on large repositories because it lets the client ask for a filtered reference advertisement instead of receiving every ref.
6. Over HTTPS, git speaks the **smart HTTP** protocol, defined in the Git documentation file `http-protocol.txt`. Two endpoints per operation: `GET $URL/info/refs?service=git-upload-pack` then `POST $URL/git-upload-pack`.
7. The media types are `application/x-git-upload-pack-request` and `application/x-git-upload-pack-result`, and the push equivalents replace `upload` with `receive`.
8. Rewriting one transport into the other is a supported config feature, and it is the cleanest fix for a blocked port:

```
$ git config --global \
    url."https://github.com/".insteadOf "git@github.com:"
$ git config --global \
    url."https://github.com/".insteadOf "ssh://git@github.com/"
```

9. On the machine used for this chapter that rewrite is active, and it is visible in a real check:

```
$ git remote add t git@github.com:octocat/Hello-World.git
$ git remote get-url t
https://github.com/octocat/Hello-World.git
```

10. That is worth knowing as a trap. If a colleague swears their SSH URL works on a machine where port 22 is blocked, check `insteadOf` before believing that port 22 is open.
11. `pushInsteadOf` is the write-only variant, letting you fetch anonymously over HTTPS while pushing over SSH.

42.1.6 WORDS - remember these

Transport — the road the data takes — the protocol carrying the Git wire protocol, SSH or HTTPS.

Remote — a nickname for a server copy — a named URL plus fetch and push refspecs stored in `.git/config`.

`git-upload-pack` — the program that sends you history — the server-side process serving fetch and clone.

`git-receive-pack` — the program that accepts your history — the server-side process serving push, and the one that runs the hooks that can refuse you.

Smart HTTP — git talking over web requests — the `info/refs` plus `git-upload-pack` request pair using `x-git-*` media types.

`insteadOf` — a rule that rewrites addresses — a `url.<base>.insteadOf` config key that substitutes URL prefixes before the transport is chosen.

42.2 What SSH actually is

42.2.1 PLAIN - in simple words

1. SSH stands for Secure Shell. A shell is the text prompt where you type commands.
2. SSH lets you get a shell on a computer that is somewhere else, over a network, safely.
3. "Safely" means two separate promises, and both matter.
4. First, nobody in the middle can read what you type or what comes back. It is scrambled on the wire.
5. Second, you can be sure the machine you reached is the machine you meant. Not an impostor sitting in the middle.
6. Before SSH, people used tools called telnet, rlogin and rsh to do the same job. Those tools sent your password as plain readable text.
7. Anyone able to watch the network could simply read it. Not crack it. Read it, like reading a postcard.
8. In 1995 a researcher at a Finnish university discovered exactly that happening on his own network and wrote SSH in response.
9. Today SSH does far more than shells. It carries file copies, tunnels, and the git pushes in this chapter.
10. But underneath it is still the same thing: a safe way to run one command on another machine.

42.2.2 PLAIN - a picture in your head

1. Think of the old way as shouting your order across a crowded market square.
2. The stallholder hears you. So does everyone else, including the person who writes down what people order and comes back later pretending to be you.
3. Now think of SSH as a sealed pneumatic tube running from your desk to the stall.
4. You put a written note in a capsule, it travels inside the tube, and only the stall can open it. Nobody in the square sees anything.
5. But a tube alone is not enough. What if somebody quietly re-routed your tube to a different stall last night?
6. So SSH adds a second step. Before you send anything, the stall shows you a seal that only that stall could have made.
7. You compare it against the seal you wrote down the first time you visited. If it matches, it is really them.
8. If it does not match, everything stops with a very loud warning.

Where this comparison breaks:

9. A pneumatic tube is a physical private path. SSH runs over the same shared public cables as everything else. The privacy is mathematical, not physical.
10. And the "seal you wrote down the first time" is a real weakness. On that first visit you had no way to check. You simply trusted. We will look straight at that in section 42.5.

42.2.3 PLAIN - a worked example

1. A connection has three visible stages. Here is a real one against a small server, with the client asked to explain itself.

```
$ ssh -v -p 2222 root@127.0.0.1 'echo OK'
debug1: Authenticating to 127.0.0.1:2222 as 'root'
```

```
debug1: kex: algorithm: sntrup761x25519-sha512@openssh.com
debug1: kex: host key algorithm: ssh-ed25519
debug1: kex: server->client cipher:
  chacha20-poly1305@openssh.com MAC: <implicit>
debug1: Server host key: ssh-ed25519
  SHA256:fwKYxD4JEmBcHpHj3WkgxSLrRh6tXIzpi/KOxz2yNrA
debug1: Host '[127.0.0.1]:2222' is known and matches the
  ED25519 host key.
debug1: Authentications that can continue: publickey
debug1: Offering public key: id_ed25519 ED25519
  SHA256:WH1E2gSLhG6FgMsha//GINoqouvX3fLxFRdMA3KMrSU
debug1: Server accepts key: id_ed25519 ED25519
Authenticated to 127.0.0.1 using "publickey".
OK
```

2. Line by line, in order of what happened.
3. `kex` is short for key exchange. Both sides agreed on the maths they would use to create a shared secret. That is stage one.
4. `Server host key` is the server proving who it is. `Host is known and matches` means the client checked its own notes and was satisfied.
5. `Authentications that can continue: publickey` is the server saying which ways of proving yourself it will accept. Here, only keys.
6. `Offering public key` then `Server accepts key` is stage two, you proving who you are.
7. `OK` at the end is stage three, the actual work happening inside the now trusted, now encrypted connection.
8. Notice the order. Encryption is set up first, then the server proves itself, then you prove yourself. Your credentials never travel in the open.

42.2.4 PLAIN - what is really happening inside

1. SSH version 2 is built as three stacked layers. Each has one job.
2. **Layer one, transport.** It runs immediately after the TCP connection opens. It has three tasks.
3. It agrees the algorithms both sides will use, it performs a key exchange to create a shared secret, and it verifies the server's host key.
4. From that point on, everything is encrypted and every message carries an integrity check so tampering is detected.
5. Importantly, the shared secret is created fresh for this connection. It is never sent over the wire, by either side. Both sides compute it.
6. **Layer two, user authentication.** Now that the pipe is safe, you prove who you are. Public key, password, keyboard-interactive, or host-based.
7. This layer runs inside the encrypted tunnel, which is why a password sent over SSH is not exposed the way a telnet password was.
8. **Layer three, connection.** One SSH connection can carry many independent streams, called channels, at once.
9. A shell is a channel. A file copy is a channel. A forwarded port is a channel. They share the one encrypted pipe.
10. That is how you can be editing a file over SSH while a port forward runs in the background, on one connection and one login.
11. For git, exactly one channel is opened, of type "exec", carrying one command: `git-upload-pack` or `git-receive-pack`.

```
    TCP connect (port 22)
         |
         v
 [1] TRANSPORT  algorithm agreement, key exchange,
```

```
                server host key check -> encrypted
       |
       v
[2] USER AUTH  publickey / password / etc.
       |
       v
[3] CONNECTION channels: shell, exec, sftp, forwards
       |
       v
  git-upload-pack runs on the server
```

42.2.5 TECHNICAL - the engineer's version

1. SSH was written by Tatu Ylonen at Helsinki University of Technology in 1995, after a password-sniffing attack on the university network.
2. He released the first version in July 1995, and founded SSH Communications Security in December 1995 to commercialize it.
3. The protocol it replaced was the Berkeley r-commands: `rlogin` (TCP 513), `rsh` (TCP 514) and also `telnet` (TCP 23), all of which sent credentials in clear text.
4. Those tools also trusted a file called `.rhosts`, which authenticated by source IP address alone. Address spoofing therefore meant login.
5. **SSH-1** and **SSH-2** are different protocols, not versions of one protocol. SSH-1 has a structural integrity flaw.
6. SSH-1 used CRC-32 for integrity, which is a checksum, not a cryptographic authentication code. In 1998 this was shown to allow an insertion attack, and in 2001 the widely reported "SSH CRC-32 compensation attack detector" vulnerability made it worse. SSH-1 is dead and should never be enabled.
7. SSH-2 is specified in a set of IETF documents published in January 2006: RFC 4250 assigned numbers, RFC 4251 architecture, RFC 4252 authentication, RFC 4253 transport, RFC 4254 connection.
8. **OpenSSH** is a separate implementation, from the OpenBSD project. The history is precise and worth knowing.

Date	Event
July 1995	Ylonen releases ssh 1.x
early 1999	Gronvall's OSSH from 1.2.12
26 Sep 1999	OpenBSD forks OSSH
1 Dec 1999	OpenSSH 1.2.2 in OpenBSD 2.6
15 Jun 2000	SSH-2 support, OpenSSH 2.0

9. The fork happened because releases after ssh 1.2.12 carried increasingly restrictive licence terms. Bjorn Gronvall's OSSH revived the last freely licensed release, and the OpenBSD team took it from there.
10. SSH-2 support was written by Markus Friedl and shipped in OpenSSH 2.0 with OpenBSD 2.7 on 15 June 2000.
11. Port 22 is the IANA-assigned port for SSH. Ylonen requested it because it sat unused between `ftp` on 21 and `telnet` on 23.
12. Modern algorithm defaults on the OpenSSH 9.6 used for this chapter: key exchange `sntrup761x25519-sha512@openssh.com`, which combines the post-quantum lattice scheme NTRU Prime with X25519, and cipher `chacha20-poly1305@openssh.com`.
13. That hybrid key exchange became the OpenSSH default in version 9.0, in April 2022. The reason given was harvest-now-decrypt-later: an attacker recording traffic today and decrypting it after quantum computers arrive.

42.2.6 WORDS - remember these

SSH — a safe way to use a distant computer — the Secure Shell protocol suite specified in RFC 4251 through RFC 4254.

Key exchange (kex) — the step where both sides build a shared secret — a Diffie-Hellman style agreement producing session keys without transmitting them.

Host key — the server's own identity key — the long-term key pair a server uses to sign the key exchange and prove it is itself.

Channel — one stream inside the connection — a multiplexed logical stream in the SSH connection protocol, with independent flow control.

telnet — an old unsafe remote login tool — a clear-text remote terminal protocol on TCP 23, RFC 854, with no encryption or integrity.

OpenSSH — the SSH program almost everyone runs — the OpenBSD implementation, first released 1 December 1999 with OpenBSD 2.6.

42.3 Public key cryptography for SSH

42.3.1 PLAIN - in simple words

1. A key pair is two files made at the same time, mathematically joined.
2. One is called the **private key**. It stays on your machine and you never send it anywhere, ever.
3. The other is called the **public key**. It is meant to be handed out. You paste it into web forms and email it to people.
4. The private key can do one special thing: it can make a **signature** on a piece of data.
5. The public key cannot make a signature. It can only check one, and answer yes or no.
6. That asymmetry is the whole trick. Publishing the public key gives an attacker nothing, because checking signatures is not making them.
7. So when GitHub wants to know you are you, it does not ask for a password.
8. It sends your computer a piece of data and says: sign this.
9. Your computer signs it with the private key. GitHub checks the signature with the public key it already stored under your account.
10. If the check passes, only the holder of that private key could have produced it. You are in.
11. Nothing secret ever crossed the network. The private key stayed in a file on your laptop the entire time.
12. That is why a public key on a public web page is not a leak, and why a private key in a chat message is a catastrophe.

42.3.2 PLAIN - a picture in your head

1. Imagine a very unusual rubber stamp.
2. Only you own the stamp. It presses a pattern so intricate that nobody can carve a copy from looking at an impression.
3. You give everyone a printed picture of the pattern. That is the public key.
4. Now anyone can send you a fresh sheet of paper with today's date and a random number on it and ask you to stamp it.
5. You stamp it and hand it back. They compare the impression with the printed picture. It matches, so it was you.
6. The random number matters more than it looks. If the paper always said the same thing, someone could keep an old stamped sheet and reuse it.
7. Because the paper is fresh every time, an old stamped sheet proves nothing about today.

Where this comparison breaks:

8. A rubber stamp makes the same mark every time. A real signature depends on both the key and the exact data, so it is different for every message.
9. And a rubber stamp can be physically stolen and used. A private key can be locked with a passphrase, so stealing the file is not always enough.
10. The honest version: some key types, RSA in particular, can also be used the other way round, to encrypt something that only the private key can open. In SSH user authentication, modern practice is signing, not encrypting.

42.3.3 PLAIN – a worked example

1. Here is a real key being made on the machine used for this chapter. Nothing is edited except line width.

```
$ ssh-keygen -t ed25519 -C "kedbyte@laptop"
Generating public/private ed25519 key pair.
Enter file in which to save the key (~/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ~/.ssh/id_ed25519
Your public key has been saved in ~/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:WH1E2gSLhG6FgMsha//GINoqouvX3fLxFRdMA3KMrSU
kedbyte@laptop
The key's randomart image is:
+--[ED25519 256]--+
|   ... o. .+Bo.o |
|. o   o..o Eo+o .|
| + o . .o + *  o |
|..o   oo   o    .|
|. .  .. S    . . |
| . o          o  |
|... = . ..   .   |
|+ .. = o .o .    |
|O+. .   o. .     |
+----[SHA256]-----+
```

2. Two files came out. `id_ed25519` is private, 411 bytes. `id_ed25519.pub` is public, 96 bytes.
3. The fingerprint is a short hash of the public key, for humans to compare by eye. It is not a key and it is not secret.
4. The randomart is the same fingerprint drawn as a picture, because humans spot a changed picture faster than a changed string of letters.
5. Now let us prove what a private key can actually do. Sign a file:

```
$ echo "hello kedbyte" > msg.txt
$ ssh-keygen -Y sign -f ~/.ssh/id_ed25519 -n file msg.txt
Signing file msg.txt
Write signature to msg.txt.sig
```

6. Check it with only the public key:

```
$ ssh-keygen -Y verify -f allowed_signers -I kedbyte@laptop \
    -n file -s msg.txt.sig < msg.txt
Good "file" signature for kedbyte@laptop with ED25519 key
SHA256:WH1E2gSLhG6FgMsha//GINoqouvX3fLxFRdMA3KMrSU
```

7. Now change one word of the message and check the same signature again:

```
$ ssh-keygen -Y verify -f allowed_signers -I kedbyte@laptop \
    -n file -s msg.txt.sig < msg2.txt
```

```
Signature verification failed: incorrect signature
Could not verify signature.
```

8. That is the entire idea, demonstrated. The signature is bound to the exact bytes. Change one character and it stops matching.

42.3.4 PLAIN - what is really happening inside

1. Look at the public key file. It is one line with three fields separated by spaces. Here it is, wrapped for the page:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIIpKlC+6wSQAkCHl
gza1Sb50MnBwsPXPD6FAeelzkUGS kedbyte@laptop
```

2. Field one, `ssh-ed25519`, is the key type. Field three, `kedbyte@laptop`, is a comment. It is a label for humans and has no effect on anything.
3. Field two is the key itself, encoded in base64 so it can be pasted into a text box. Decoded, it is 51 bytes with a strict internal structure.
4. That structure is a repeating pattern: a four-byte length, then that many bytes of data. Here is the real decode of the key above:

```
total bytes: 51
field 1: length 11 -> 'ssh-ed25519'
field 2: length 32 -> 8a4a942fbac124009021e58336b549be
                      74327070b0f5cf0fa14079e973914192
```

5. So the type name appears twice: once in plain text outside the base64, and once inside it. The inside copy is the one that counts.
6. The 32 bytes are the actual Ed25519 public key. That is the whole secret-free half of the pair, and it really is only 32 bytes.
7. Now the login exchange, in order.
8. Your client tells the server: I would like to authenticate as this user with this public key. Would you accept it at all?
9. The server looks the key up in that user's `authorized_keys` file. If it is not there, the server says no and your client tries the next key.
10. If it is there, the server says yes, that key is acceptable. Note that nothing has been proven yet.
11. Your client now builds a block of data to sign. It contains the session identifier from the key exchange, the user name, the service name, and the public key.
12. The session identifier is the crucial part. It is unique to this one connection and could not have been known in advance.
13. Your client signs that block with the private key and sends the signature.
14. The server verifies the signature against the stored public key. If it verifies, you are authenticated.
15. Because the signed block includes the session identifier, a signature captured from an old session cannot be replayed into a new one.
16. And because the server also binds its own host key into the session identifier, a man-in-the-middle cannot forward your signature to the real server. That property is why host key checking is not optional.

42.3.5 TECHNICAL - the engineer's version

1. Three key types are in real use, plus a dead one. Real measurements from the OpenSSH 9.6 machine used for this chapter.

Type	Bits	Public file size
Ed25519	256	96 bytes

Type	Bits	Public file size
ECDSA nistp256	256	182 bytes
RSA 4096	4096	744 bytes
DSA	1024	removed everywhere

2. Generation cost, averaged over five runs on the same machine:

Type	Average generation time
Ed25519	0.010 s
ECDSA nistp256	0.008 s
RSA 2048	0.229 s
RSA 4096	2.160 s

3. **RSA.** Published by Rivest, Shamir and Adleman in 1978. Security rests on the difficulty of factoring a large number into two primes.
4. 1024-bit RSA is broken in practice for well-funded attackers and is refused by modern software. 2048 bits is the floor, and is what NIST Special Publication 800-57 treats as acceptable to roughly 2030.
5. 4096 bits is common because it costs almost nothing for a key you generate once, and pushes the estimated strength well past the 2030 line.
6. Beyond 4096 the returns collapse. RSA key strength grows very slowly with size while signing cost grows fast, which is why 8192 is rare.
7. The critical modern detail is the signature algorithm, not the key size. The old `ssh-rsa` algorithm uses SHA-1, which has been considered unsafe since the 2017 SHAttered collision by CWI Amsterdam and Google.
8. RFC 8332, published in March 2018, defines `rsa-sha2-256` and `rsa-sha2-512` for the same RSA keys. OpenSSH disabled `ssh-rsa` by default in version 8.8 in September 2021.
9. **ECDSA.** The elliptic curve digital signature algorithm, standardized in FIPS 186. In SSH it is used with the NIST curves P-256, P-384 and P-521.
10. ECDSA works and is widely supported, but it has two known unhappy properties. It needs a fresh random value for every signature, and a repeated or biased value leaks the private key outright.
11. That is not theoretical. The 2010 Sony PlayStation 3 signing key recovery was exactly this bug, a fixed nonce.
12. Also, some cryptographers distrust the provenance of the NIST curve constants. This is a genuine expert disagreement. One side says the constants are unexplained and should be avoided, the other says no attack has ever followed from them and the curves are fine.
13. **Ed25519.** EdDSA over the twisted Edwards curve Curve25519, from the 2011 paper "High-speed high-security signatures" by Bernstein, Duif, Lange, Schwabe and Yang. Curve25519 itself is Bernstein's, from 2006.
14. It is the current recommendation for SSH, for four reasons that are concrete rather than fashionable.
15. It is deterministic: the per-signature value is derived from the key and the message, so the ECDSA nonce failure mode cannot happen.
16. It is small and fast: 32-byte public keys, 64-byte signatures, and verification that is quick on tiny hardware.
17. Its parameters were chosen by a published rule with stated rationale, so there are no unexplained constants.
18. Its implementations are naturally constant-time, which resists timing side-channel attacks without special care.
19. Ed25519 in SSH is standardized in RFC 8709, February 2020, though OpenSSH shipped it in version 6.5 in January 2014, six years earlier.

20. **DSA.** `ssh-dss`, 1024 bits, roughly 80-bit security. GitHub removed support for DSA keys entirely, permanently enforced on 15 March 2022.
21. The complete list your OpenSSH understands is one command:

```
$ ssh -Q key
ssh-ed25519
sk-ssh-ed25519@openssh.com
ecdsa-sha2-nistp256
ecdsa-sha2-nistp384
ecdsa-sha2-nistp521
sk-ecdsa-sha2-nistp256@openssh.com
ssh-dss
ssh-rsa
```

22. The `sk-` prefixed types are hardware security key types, added in OpenSSH 8.2 in February 2020. The private key half lives on a physical device such as a YubiKey and cannot be copied off it.
23. Fingerprints default to SHA-256 in base64 since OpenSSH 6.8, March 2015. The older MD5 hex form is still available and still seen in old documentation:

```
$ ssh-keygen -lf id_ed25519.pub
256 SHA256:WH1E2gSLhG6FgMsha//GINoqouvX3fLxFRdMA3KMrSU
kedbyte@laptop (ED25519)

$ ssh-keygen -lf id_ed25519.pub -E md5
256 MD5:74:36:b8:87:74:1d:49:9c:09:81:62:6b:89:eb:26:ad
kedbyte@laptop (ED25519)
```

24. Practical recommendation, and it is not controversial: `ssh-keygen -t ed25519 -C "you@machine"`. Use RSA 4096 only when you must talk to something old that cannot do Ed25519.

42.3.6 WORDS - remember these

Key pair — two matched files, one secret one public — an asymmetric keypair where the public half is derivable from the private half but not the reverse.

Private key — the file you never share — the secret scalar or exponent, stored under `~/.ssh/`, optionally encrypted with a passphrase.

Public key — the file you paste everywhere — the shareable half, safe to publish, stored on the server in `authorized_keys`.

Fingerprint — a short code for comparing keys by eye — the base64 SHA-256 hash of the public key blob, printed by `ssh-keygen -l`.

Signature — proof that the holder of the key saw this data — the output of a signing algorithm over specified bytes, verifiable with the public key alone.

Ed25519 — the recommended modern key type — EdDSA over Curve25519, RFC 8709, 32-byte public keys, deterministic signatures.

Nonce — a number used once — a per-signature random or derived value whose reuse in ECDSA leaks the private key.

42.4 `authorized_keys` and how it works

42.4.1 PLAIN - in simple words

1. On the server there is one plain text file that decides who may log in as a given user with a key.
2. It is called `authorized_keys` and it lives in that user's home directory, inside a folder called `.ssh`.

3. Each line of the file is one public key. If your public key is on a line, you may log in as that user.
4. That really is the entire mechanism. There is no database and no clever service. It is a text file that gets read.
5. To give someone access, you append their public key as a new line. To remove access, you delete that line.
6. There is one rule that catches everybody. The file and its folder must not be writable by anyone other than the owner.
7. If they are, the server refuses to use the file at all, and says nothing useful about why.
8. The reason is sound. If any other user could edit that file, they could add their own key and become you. So SSH refuses to trust it.
9. The failure is silent from the client's side. You get "permission denied" and no explanation.
10. GitHub has a variant of this idea for a single repository, called a **deploy key**. Same file format, narrower reach.

42.4.2 PLAIN - a picture in your head

1. Think of a guest list on a clipboard at the door of a private room.
2. The doorman does not know anyone's face. He only checks the list.
3. To let someone in, you write their name on the list. To stop them, you cross it off. There is no other step.
4. Now add the rule that catches people out. The clipboard hangs on a hook in the public corridor.
5. If anyone walking past can write on it, the list is worthless. Anyone could add themselves.
6. So the doorman has a standing instruction: if the clipboard is not locked in its case, ignore the list entirely and let nobody in.
7. And he does not explain. He just says "not on the list", because telling you about the broken case would help an attacker more than it helps you.

Where this comparison breaks:

8. A name on a list can be claimed by anyone who knows the name. A public key on a line can only be used by whoever holds the matching private key.
9. So the list being publicly readable is fine. Only the list being publicly writable is fatal.

42.4.3 PLAIN - a worked example

1. This is a real server, a real key, and a real failure, produced on the machine used for this chapter.
2. First, correct permissions. The connection works.

```
$ ls -l ~/.ssh/authorized_keys
-rw------- 1 root root 96 Aug 13 08:22 authorized_keys
$ ssh -i ~/.ssh/id_ed25519 -p 2222 root@127.0.0.1 'echo OK'
OK
```

3. Now make the file writable by everyone, which is exactly the mistake people make when they get frustrated and run `chmod 666`.

```
$ chmod 666 ~/.ssh/authorized_keys
$ ls -l ~/.ssh/authorized_keys
-rw-rw-rw- 1 root root 96 Aug 13 08:22 authorized_keys
$ ssh -i ~/.ssh/id_ed25519 -p 2222 root@127.0.0.1 'echo OK'
root@127.0.0.1: Permission denied (publickey).
```

4. The key did not change. The user did not change. Only the permission bits changed, and the login broke.
5. Nothing in the client message says so. It says `Permission denied (publickey)`, which is the same message you get for a completely wrong key.

6. Now the server's own log, from the same moment:

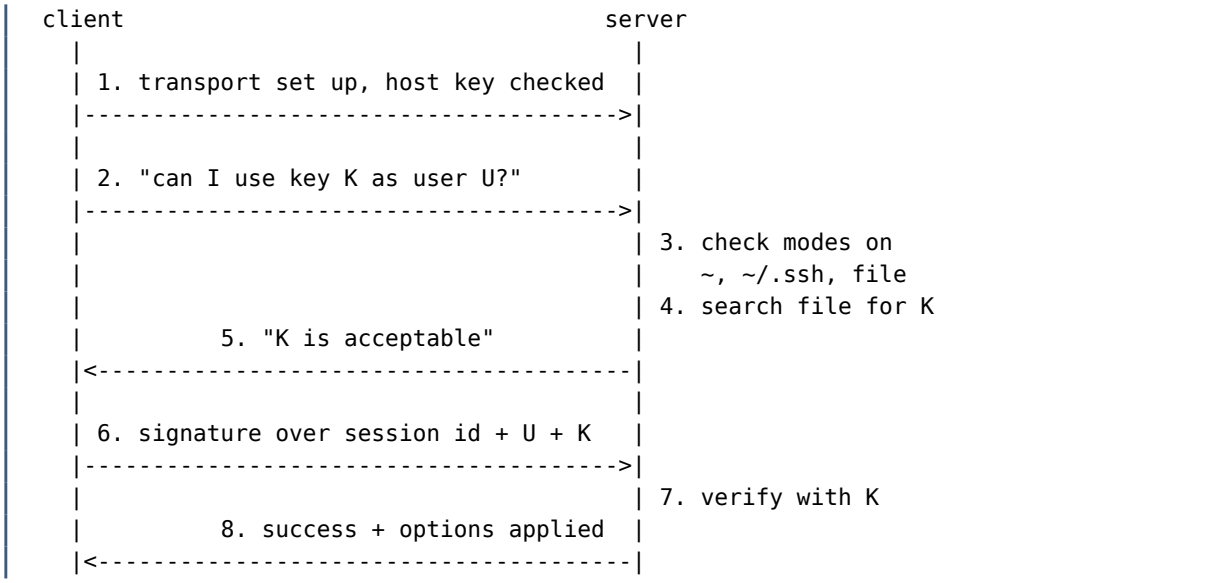
```
Connection from 127.0.0.1 port 48576 on 127.0.0.1 port 2222
Authentication refused: bad ownership or modes for file
  /root/.ssh/authorized_keys
Failed publickey for root from 127.0.0.1 port 48576 ssh2:
  ED25519 SHA256:WH1E2gSLhG6FgMsha//GINoqouvX3fLxFRdMA3KMrSU
Connection closed by authenticating user root [preauth]
```

7. There it is: `bad ownership or modes`. The truth exists, but only on the server side.
8. This is the single most useful piece of knowledge in this section. When public key login fails and you control the server, read the server log.
9. The fix is two commands:

```
$ chmod 700 ~/.ssh
$ chmod 600 ~/.ssh/authorized_keys
```

42.4.4 PLAIN - what is really happening inside

1. Here is the exchange as a numbered sequence, with a diagram after it.
2. The client opens TCP to port 22 and the two sides complete the encrypted transport layer, including the server proving its host key.
3. The client sends an authentication request naming the user, the service, the method `publickey`, and one public key. A flag says "this is only a query".
4. The server takes the user name and looks up that user's home directory.
5. It checks the ownership and permission bits on the home directory, on `~/.ssh`, and on `~/.ssh/authorized_keys`.
6. If any of them is writable by group or by others, the server abandons the file and behaves as if no key matched. This is the silent failure.
7. Otherwise it reads the file line by line, skipping blanks and comments, looking for a line whose key blob equals the one offered.
8. If none matches, the server replies "failure" and the client tries its next key, then finally gives up with `Permission denied (publickey)`.
9. If one matches, the server replies "this key is acceptable, now prove it".
10. The client builds the signature block, signs it, and sends it. The server verifies with the stored public key.
11. On success the server applies any options written in front of that key on that line, then starts the session.



12. Steps 3 and 4 are where nearly all real-world failures live, and neither of them tells the client anything.

42.4.5 TECHNICAL - the engineer's version

1. Default path is `~/.ssh/authorized_keys`, set by the `AuthorizedKeysFile` directive in `sshd_config`. It accepts multiple paths and the tokens `%h` for home directory and `%u` for user name.
2. Required modes, enforced when `StrictModes yes` is set, which is the default:

Path	Required
`~`	not group/other writable
`~/.ssh`	700 recommended
`~/.ssh/authorized_keys`	600, owner only
`~/.ssh/id_*` private key	600, client side too

3. The client enforces its own modes as well. A private key readable by others produces `UNPROTECTED PRIVATE KEY FILE` and is refused.
4. Each line is: optional comma-separated options, then key type, then base64 key, then optional comment. No line continuation exists, so a wrapped paste is the second most common cause of failure after permissions.
5. The options that matter in production:

Option	Effect
`command="..."`	force this command only
`from="host,pattern"`	restrict source addresses
`no-port-forwarding`	block tunnels
`no-agent-forwarding`	block agent reuse
`no-pty`	no interactive terminal
`restrict`	all restrictions at once
`expiry-time="YYYYMMDD"`	key stops working

6. `restrict` was added in OpenSSH 7.2, February 2016, and is the safe default because it enables all current and future restrictions, letting you re-enable individually with `permit*` options.
7. A real forced command, tested on the machine used for this chapter:

```
$ cat ~/.ssh/authorized_keys
command="echo forced: you may only run this",
no-port-forwarding,no-agent-forwarding,no-pty
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAA... kedbyte@laptop

$ ssh -i ~/.ssh/id_ed25519 -p 2222 root@127.0.0.1 \
    'rm -rf /important'
forced: you may only run this
```

8. The destructive command was sent and simply not run. The original command is still available to the forced command in the environment variable `SSH_ORIGINAL_COMMAND`, which is how restricted git servers work.
9. That is precisely the mechanism behind `git-shell`, the login shell that permits only `git-upload-pack`, `git-receive-pack` and `git-upload-archive`.
10. **Deploy keys** are GitHub's per-repository variant. The public key is attached to one repository, not to a user account.
11. Deploy keys are read-only by default and can be granted write access when added. One key cannot be reused across repositories: GitHub rejects a duplicate.

12. GitHub lists five alternatives to deploy keys: SSH agent forwarding, HTTPS with OAuth tokens, GitHub App installation access tokens which expire after one hour, machine users, and personal access tokens.
13. On GitHub, all SSH access uses the single account name `git`. Your identity comes entirely from which key signed, not from the user name in the URL.
14. That is why `git@github.com` is the same for every person on earth, and why `ssh -T git@github.com` greets you by your own name.

42.4.6 WORDS - remember these

`authorized_keys` — the list of keys allowed in — a per-user file of public key lines consulted by sshd during publickey authentication.

`StrictModes` — the rule that file permissions must be tight — an sshd directive making the server refuse key files writable by group or others.

Forced command — a key that can only do one thing — the `command="..."` option overriding whatever the client requested.

`SSH_ORIGINAL_COMMAND` — what the client actually asked for — the environment variable holding the requested command when a forced command is in effect.

Deploy key — a key for one repository, not one person — a public key attached to a single GitHub repository, read-only unless granted write.

`git-shell` — a login shell that only allows git — a restricted shell shipped with Git permitting only the three pack transfer commands.

42.5 `known_hosts` and host key verification

42.5.1 PLAIN - in simple words

1. Everyone thinks about proving themselves to the server. Almost nobody thinks about the server proving itself to them.
2. But that half exists, it runs first, and it is the half that stops somebody sitting invisibly in the middle of your connection.
3. Every SSH server has its own key pair, separate from any user's. It is called the **host key**.
4. During setup, the server signs part of the conversation with its private host key. Only the real server could do that.
5. Your client checks that signature against a copy of the server's public key it saved earlier, in a file called `known_hosts`.
6. And here is the awkward truth. The first time you ever connect, your client has no saved copy, so it cannot check anything.
7. So it asks you. It shows you a fingerprint and asks whether to trust it. Most people type yes without looking.
8. That approach has a name: **trust on first use**. You are trusting that nobody was interfering during that one first moment.
9. After that first yes, the key is saved and every later connection is properly checked.
10. If the saved key ever stops matching, SSH stops everything and prints a very loud warning. That warning is doing its job. Read it, do not delete it.

42.5.2 PLAIN - a picture in your head

1. You are meeting someone you have only spoken to by letter, and you must be sure it is really them.
2. The first meeting is the hard one. They show you a birthmark. You have no way to check it, so you write it down and decide to believe it.
3. Every meeting afterwards is easy. You look for the birthmark you wrote down. Match means it is them.

4. Now imagine you turn up and the birthmark is different.
5. Three explanations. They had it removed and told nobody. They have an identical twin who came instead. Or somebody is impersonating them.
6. You cannot tell which from looking. What you can do is stop, and go and check through a different channel before saying anything private.

Where this comparison breaks:

7. A birthmark cannot be changed at will. A host key can be legitimately replaced, and sometimes must be. So a mismatch is not automatically an attack.
8. And a person can be phoned to confirm. A server cannot. You have to find the fingerprint published somewhere trustworthy instead.

42.5.3 PLAIN – a worked example

1. All of the following is real output from the machine used for this chapter, against a small server started for the purpose.
2. The first connection, with nothing saved yet:

```
$ ssh -p 2222 root@127.0.0.1
The authenticity of host '[127.0.0.1]:2222' can't be
established.
ED25519 key fingerprint is
SHA256:nu2laRCiSw3wMTbG6hoH8tKukC952d+/PbROtgsD5vU.
This key is not known by any other names.
Are you sure you want to continue connecting
(yes/no/[fingerprint])?
```

3. Answer yes and the client writes one line into `known_hosts`. Here it is, truncated for page width:

```
$ cat ~/.ssh/known_hosts
[127.0.0.1]:2222 ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAA
IJJmXI2wR7keztPI78nlxu2ooGFa2+d/ke5qLMBiy5...
```

4. Three fields: which host, which key type, and the key. Note the host is in square brackets with the port, because it is not port 22.
5. Now the server's host key was deliberately replaced and the same connection tried again. This is the real, complete warning:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@   WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now
(man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ED25519 key sent by the remote
host is
SHA256:fwKYxD4JEmBcHpHj3WkgxSLrRh6tXIzpi/KOxz2yNrA.
Please contact your system administrator.
Add correct host key in ~/.ssh/known_hosts to get rid of
this message.
Offending ED25519 key in ~/.ssh/known_hosts:1
  remove with:
  ssh-keygen -f '~/.ssh/known_hosts' -R '[127.0.0.1]:2222'
Host key for [127.0.0.1]:2222 has changed and you have
requested strict checking.
Host key verification failed.
```

6. Notice the message even tells you the exact line number and the exact command to remove it. That convenience is why people delete first and think never.
7. Notice also that the connection stopped. No password prompt appeared. SSH refused to send anything to a server it could not identify.

42.5.4 PLAIN - what is really happening inside

1. The host key check happens during the key exchange, before any authentication method is even offered.
2. The two sides compute a shared secret, then a hash covering both version strings, both algorithm lists, both key exchange values and the shared secret. That hash is the **session identifier**.
3. The server signs the session identifier with its private host key and sends the signature plus its public host key.
4. The client verifies the signature using the public host key it was just sent. That proves the far end holds the matching private host key.
5. But it does not prove the far end is the right machine. Anyone can generate a key pair and sign with it.
6. So the client separately compares that public host key against `known_hosts`. That is the step that ties the maths to a name you care about.
7. Four outcomes. Not found means prompt the user. Found and matching means proceed. Found and different means abort loudly.
8. The fourth outcome is subtler: found for a different key type. If you have an Ed25519 entry and the server offers RSA, OpenSSH says "this key is not known by any other names" rather than warning of a change.
9. When a mismatch happens, SSH refuses to continue in the default configuration, because continuing means handing your credentials to something unverified.
10. If you use password authentication and click through such a warning, you have just typed your password into whatever is in the middle. With key authentication you would have leaked a signature, which is far less valuable, but still not nothing.

42.5.5 TECHNICAL - the engineer's version

1. `known_hosts` lives at `~/.ssh/known_hosts` for the user and `/etc/ssh/ssh_known_hosts` for the system. Controlled by `UserKnownHostsFile` and `GlobalKnownHostsFile`.
2. Line format: host patterns, key type, base64 key, optional comment. Host patterns may be comma-separated, may use `*` and `?`, and non-default ports are written `[host]:port`.
3. `HashKnownHosts yes` stores hosts as HMAC-SHA-1 hashes so a stolen file does not enumerate every server you use. Debian and Ubuntu enable this by default; upstream OpenSSH does not.
4. `StrictHostKeyChecking` has four values, and choosing correctly matters more than people think:

Value	Behaviour on new host
`ask`	prompt (upstream default)
`accept-new`	add, but warn on change
`no`	add silently, allow change
`yes`	refuse unless already known

5. `accept-new` was added in OpenSSH 7.6, October 2017, and is the correct choice for scripts. `no` is dangerous because it also tolerates a changed key, which is the case you actually care about.
6. Useful commands, all real:

```
$ ssh-keyscan -t ed25519 -p 2222 127.0.0.1 >> known_hosts
$ ssh-keygen -F '[127.0.0.1]:2222' -f known_hosts
# Host [127.0.0.1]:2222 found: line 1
$ ssh-keygen -R '[127.0.0.1]:2222' -f known_hosts
```

7. A warning that must be stated plainly: `ssh-keyscan` asks the server what its key is. If something is impersonating the server, `ssh-keyscan` will faithfully record the impostor's key.
8. So `ssh-keyscan` is a convenience for populating a file, not a verification tool. Verification means comparing against a fingerprint published by the operator through a different channel.
9. **The GitHub 2023 rotation is the textbook legitimate change.** At approximately 05:00 UTC on 24 March 2023, GitHub replaced its RSA SSH host key.
10. The cause was not a compromise of GitHub's systems. GitHub's RSA SSH private key had been briefly exposed in a public GitHub repository, and they rotated it as a precaution.
11. Every user with an RSA entry for `github.com` in `known_hosts` then saw the REMOTE HOST IDENTIFICATION HAS CHANGED warning. The remedy GitHub published was `ssh-keygen -R github.com` followed by re-adding the new key.
12. Users whose `known_hosts` held the Ed25519 or ECDSA entry saw nothing, because only the RSA key changed.
13. GitHub's current published fingerprints, which you should compare against rather than trusting a prompt:

Type	SHA-256 fingerprint
RSA	uNiVztksCsDhcc0u9e8Bu…
ECDSA	p2QAMXNIC1TJYWeIOttr…
Ed25519	+DiY3wvvV6TuJJhbpZis…

14. The full Ed25519 fingerprint is `SHA256:+DiY3wvvV6TuJJhbpZisF/zLDA0zPMSvHdkr4UvCOqU`, and the full RSA one is `SHA256:uNiVztksCsDhcc0u9e8BujQXVUpKZIDTMczCvj3tD2s`.
15. GitHub publishes these machine-readable at the `/meta` endpoint of its API, under the `ssh_keys` field, which is what automation should consume.
16. When is a changed host key benign? When the operator announced it, when the server was rebuilt or migrated, when you connected to a load-balanced pool with inconsistent keys, or when your DNS now resolves to a different machine because you changed networks.
17. When is it alarming? When nothing changed on your side, when it happens on an untrusted network such as public wifi, when only some connections show it, or when the operator's published fingerprint does not match.
18. The stronger long-term answer is **SSH certificates**. A certificate authority signs host keys, clients trust the CA once with a `@cert-authority` line, and individual host key changes stop mattering. OpenSSH has supported this since version 5.4, March 2010.

42.5.6 WORDS - remember these

`known_hosts` — your notebook of server identities — a client-side file mapping host patterns to accepted host public keys.

Host key — the server's identity key — the long-term key pair sshd uses to sign the session identifier during key exchange.

Trust on first use — believing the first answer you get — the TOFU model, where the initial unverified key is pinned and all later ones checked against it.

`ssh-keyscan` — a tool that fetches a server's key — a scanner that records whatever key is offered, with no verification of authenticity.

Session identifier — a fingerprint of this one connection — the exchange hash from the first key exchange, bound into every authentication signature.

`@cert-authority` — trust a signer instead of each server — a `known_hosts` marker line naming a CA key that may vouch for host keys.

42.6 The SSH agent

42.6.1 PLAIN - in simple words

1. A private key should be protected by a passphrase, so that stealing the file is not enough to steal the key.
2. But that means typing a long passphrase every single time you connect.
3. Git makes this unbearable. A fetch, a push, a status check against the server: each one is a fresh connection and a fresh prompt.
4. People solve this the worst possible way. They remove the passphrase.
5. The **SSH agent** is the right solution. It is a small program that runs in the background and holds your unlocked keys in memory.
6. You type the passphrase once. The agent keeps the unlocked key. Every later connection asks the agent instead of asking you.
7. Here is the important detail. The agent does not hand the key to `ssh`.
8. `ssh` sends the agent the data that needs signing. The agent signs it and sends back only the signature.
9. So the key never leaves the agent's memory, not even to another program on your own computer.
10. The agent forgets everything when it stops, which normally means when you log out or shut down.

42.6.2 PLAIN - a picture in your head

1. Think of a notary who keeps your official seal in a locked drawer.
2. You unlock the drawer once in the morning with your combination.
3. All day, whenever you need a document sealed, you hand the document to the notary. He seals it and hands it back.
4. You never carry the seal around, and nobody else ever holds it. They only ever receive sealed documents.
5. At the end of the day the drawer locks itself and the combination is forgotten.
6. Now consider **agent forwarding**, which is the dangerous feature.
7. It means: when I visit another office, let that office phone my notary and ask for seals on their documents.
8. That is wonderfully convenient. It also means anyone with power over that office, while you are visiting, can phone your notary and get anything sealed.
9. They cannot steal the seal. They can use it, on whatever they like, for as long as you are connected.

Where this comparison breaks:

10. A notary would notice being asked to seal something outrageous. The agent signs opaque data and cannot tell a git push from a login to your bank's build server.

42.6.3 PLAIN - a worked example

1. Real output, in order, from the machine used for this chapter.
2. Start an agent and look inside it. Empty.

```
$ eval "$(ssh-agent -s)"
Agent pid 14148
$ ssh-add -l
The agent has no identities.
```

3. Add a key that genuinely has a passphrase. The passphrase is typed once, here and nowhere else.

```
$ ssh-add ~/.ssh/id_pass
Enter passphrase for ~/.ssh/id_pass:
Identity added: ~/.ssh/id_pass (with-passphrase)
$ ssh-add -l
```

```
256 SHA256:cY2TWmZ8NY6LT0x9/BYw2D8XxqpmW4+M/C5K7i+nnOI
with-passphrase (ED25519)
```

4. Now connect with no `-i` flag naming a key, and with prompting explicitly disabled so it cannot secretly ask for anything.

```
$ ssh -o BatchMode=yes -p 2222 root@127.0.0.1 \
    'echo CONNECTED USING AGENT KEY'
CONNECTED USING AGENT KEY
```

5. That worked with no passphrase typed and no key file named. The agent supplied the signature.
6. Add a key with an automatic expiry, which is a habit worth building:

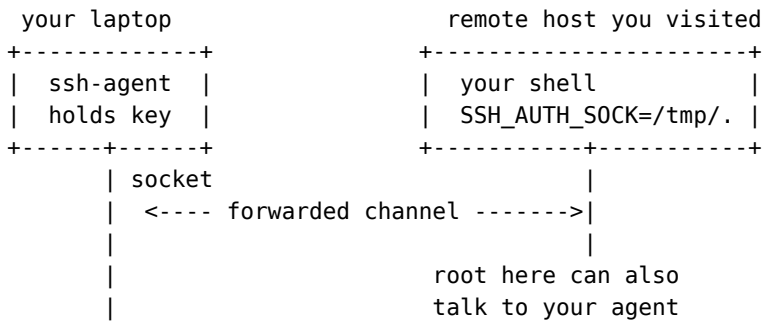
```
$ ssh-add -t 300 ~/.ssh/id_pass
Identity added: ~/.ssh/id_pass (with-passphrase)
Lifetime set to 300 seconds
```

7. And empty the agent when you walk away:

```
$ ssh-add -D
All identities removed.
```

42.6.4 PLAIN - what is really happening inside

1. The agent listens on a Unix domain socket, which is a file on disk used as a communication endpoint between programs on the same machine.
2. Its path is put in the environment variable `SSH_AUTH_SOCK`. That variable is how every other program finds it. There is no other discovery mechanism.
3. When `ssh` needs to authenticate, it connects to that socket and asks: what identities do you hold? The agent replies with public keys only.
4. `ssh` offers those public keys to the server. If the server accepts one, `ssh` builds the data to be signed and sends it to the agent.
5. The agent signs and returns the signature. `ssh` forwards it to the server. At no point does `ssh` see the private key.
6. Agent forwarding, the `-A` flag, changes one thing: on the remote machine, `sshd` creates a new socket and sets `SSH_AUTH_SOCK` to point at it.
7. Anything on that remote machine that can read that socket can ask your agent for signatures.
8. Crucially, the socket is a file, and the root user of that machine can read any file. So root on the remote host can use your keys.
9. Not copy them. Use them, silently, for as long as your session lasts, against every server your keys open.
10. That is why agent forwarding to a machine you do not fully control is a genuine security decision, not a convenience toggle.



42.6.5 TECHNICAL – the engineer's version

1. The agent protocol is specified in RFC 4251's ecosystem as `draft-miller-ssh-agent`, published as RFC 9578 in June 2024. Before that it was a de facto standard for over twenty years.
2. Core commands and what they do:

Command	Effect
`ssh-add`	add default keys
`ssh-add -l`	list fingerprints held
`ssh-add -L`	list full public keys
`ssh-add -d KEY`	remove one key
`ssh-add -D`	remove all keys
`ssh-add -t 3600`	add with 1 hour expiry
`ssh-add -c`	confirm each use

3. `ssh-add -c` is underused. It makes the agent pop a confirmation dialog for every signature, which turns silent abuse over a forwarded agent into something you would notice immediately.
4. The relevant client configuration keys:

Key	Meaning
`AddKeysToAgent yes`	auto-add on first use
`ForwardAgent yes`	enable forwarding
`IdentityAgent path`	use a specific agent
`IdentitiesOnly yes`	only offer named keys

5. `IdentitiesOnly yes` solves a real problem. Without it, `ssh` offers every key the agent holds, in order. Servers commonly refuse after six failed attempts with `Too many authentication failures`.
6. On macOS, the system agent is integrated with the login keychain. Since macOS 10.12.2 the correct configuration is explicit:

```
Host *
    UseKeychain yes
    AddKeysToAgent yes
    IdentityFile ~/.ssh/id_ed25519
```

7. `UseKeychain yes` stores the passphrase in the login keychain and retrieves it automatically, so the agent is repopulated after a reboot. It is a macOS-only option and will error on Linux.
8. On Linux, `gnome-keyring` and KDE's `ksshaskpass` provide similar integration. On Windows, OpenSSH ships an `ssh-agent` Windows service.
9. The safer alternative to agent forwarding is `ProxyJump`, added in OpenSSH 7.3, August 2016:

```
Host internal
    HostName 10.0.5.20
    ProxyJump bastion.example.com
```

10. `ProxyJump` tunnels a new SSH connection through the intermediate host. The intermediate host sees only encrypted bytes and never touches your agent.
11. A complete, realistic `~/.ssh/config`. This file must be mode 600 and OpenSSH applies the first matching value for each key, so specific hosts must come before `Host *`:

```
# GitHub over the HTTPS port, for blocked networks
Host github.com
    HostName ssh.github.com
```

```
    Port 443
    User git
    IdentityFile ~/.ssh/id_ed25519_github
    IdentitiesOnly yes

# A work server on an odd port, through a bastion
Host build
    HostName 10.0.5.20
    Port 2222
    User deploy
    ProxyJump bastion.example.com
    IdentityFile ~/.ssh/id_ed25519_work
    IdentitiesOnly yes

# Everything else
Host *
    AddKeysToAgent yes
    ServerAliveInterval 60
    ServerAliveCountMax 3
    HashKnownHosts yes
```

12. You never have to guess what that file computed. `ssh -G` prints the final resolved settings. Real output from this chapter's machine:

```
$ ssh -G demo
user root
hostname 127.0.0.1
port 2222
identitiesonly yes
identityfile /tmp/sshdemo/.ssh/id_ed25519
```

13. `ServerAliveInterval 60` with `ServerAliveCountMax 3` sends a keepalive every 60 seconds and gives up after three unanswered ones. This is the fix for connections silently dying behind a NAT device that expires idle entries.

42.6.6 WORDS - remember these

SSH agent — a background helper holding unlocked keys — a daemon exposing a signing oracle over a Unix socket, never releasing key material.

`SSH_AUTH_SOCK` — the address of your agent — the environment variable naming the agent's Unix domain socket path.

Agent forwarding — letting a remote host use your agent — an SSH channel exposing your local agent socket on the remote machine, usable by its root.

`ProxyJump` — hop through a middle machine safely — a client directive tunnelling a fresh SSH session through an intermediate host without agent exposure.

`IdentitiesOnly` — only try the key I named — a directive suppressing the agent's other keys, avoiding `Too many authentication failures`.

`UseKeychain` — let macOS remember the passphrase — an Apple-specific `ssh_config` option storing and retrieving passphrases from the login keychain.

42.7 Why SSH on port 22 can be blocked when 443 is not

42.7.1 PLAIN - in simple words

1. A port is a number attached to a connection saying which service you want on that machine.
2. Web traffic over HTTPS uses port 443. SSH uses port 22.
3. Many networks do not let you reach every port on the outside world. They allow a short list and block the rest.
4. Port 443 is always on that list. If you block 443, the web stops working, and somebody complains within seconds.
5. Port 22 is often not on the list. Blocking it upsets developers, and only developers, and usually not loudly enough to matter.
6. There are also two real reasons a network operator gives for filtering 22.
7. First, port 22 is scanned constantly by automated programs looking for servers with weak passwords. Filtering it reduces that noise.
8. Second, an outbound SSH connection can carry anything at all inside it, in a form nobody can inspect. It is a perfect way to move data out unseen.
9. So the block is not aimed at you. You are collateral.
10. And the way it fails is cruel. A blocked port 22 usually does not say "no". It says nothing at all, and your command hangs.
11. This is exactly the shape of the reader's own fault in this book, and the fix is one of two lines of configuration.

42.7.2 PLAIN - a picture in your head

1. A building with many numbered delivery bays. Bay 443 is the main goods entrance, open all day, huge queues, everyone uses it.
2. Bay 22 is a small side entrance used by maintenance staff.
3. Management decides to weld bay 22 shut. Almost nobody notices. Bay 443 cannot be welded shut, because then the business stops.
4. Now, welding shut has two possible styles.
5. The polite style is a sign on the door: "closed, use the main entrance". You read it, turn around, and lose ten seconds.
6. The unhelpful style is no sign at all. You stand there. You knock. You wait. You knock again. Eventually you give up, knowing nothing.
7. Networks overwhelmingly choose the second style, on purpose, because a sign tells a stranger which doors exist.

Where this comparison breaks:

8. A welded door is permanent and obvious on inspection. A network filter can be selective by time, by source address or by destination, so the same command can work for your colleague and hang for you.

42.7.3 PLAIN - a worked example

1. This is real, measured on the machine used to write this chapter, which happens to sit behind exactly this kind of filtering.
2. Port 22 to GitHub, with a generous timeout:

```
$ time ssh -o ConnectTimeout=30 -T git@github.com
ssh: connect to host github.com port 22: Connection timed out

real  0m30.057s
```

3. Thirty seconds, and then a timeout. Nothing came back at any point.
4. Port 443 to the same host name, in the same shell, one second later:

```
$ time bash -c 'exec 3<>/dev/tcp/github.com/443 && echo OPEN'
OPEN

real  0m0.021s
```

5. Twenty-one milliseconds. The network is fine. DNS is fine. GitHub is fine. Only port 22 is unreachable.
6. Now compare that with a port where something actively refuses. Nothing is listening on port 2200 on the local machine:

```
$ time ssh -o ConnectTimeout=10 -p 2200 root@127.0.0.1
ssh: connect to host 127.0.0.1 port 2200: Connection refused

real  0m0.015s
```

7. Fifteen milliseconds and a clear refusal. That is what a healthy "no" looks like.
8. Put the three side by side and the diagnosis writes itself.

Case	Message	Time
Nothing listening	Connection refused	0.015 s
Port 443 reachable	connects	0.021 s
Port 22 filtered	Connection timed out	30.057 s

9. The timing is the tell. Milliseconds means somebody answered. Tens of seconds means silence.

42.7.4 PLAIN - what is really happening inside

1. Chapter 34 of this book covered two ways to block a packet, and the difference decides your whole afternoon. Here it is again in one line each.
2. **REJECT** means refuse and say so. The filter sends back a TCP reset or an ICMP error. Your program fails instantly with "connection refused".
3. **DROP** means refuse and say nothing. The packet is deleted. No reply is generated at all.
4. With DROP, your computer does not know anything happened. It assumes the packet was lost, so it does what TCP always does: it sends it again.
5. The retries are spaced out with doubling gaps. One second, two, four, eight, sixteen. Then it gives up and reports a timeout.
6. That is why a blocked port 22 hangs for tens of seconds rather than failing instantly. Every second of that wait is your machine politely retrying.
7. Firewall administrators choose DROP deliberately. A REJECT confirms that a host exists at that address, which helps someone mapping your network.
8. So silence is a security choice that costs you diagnosis time.
9. Applying it to the reader's own session: `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and then produced nothing for fifteen seconds. That is the same signature at a different port.
10. What that proves is narrow and must be stated carefully. It proves no response arrived. It is consistent with a silent drop somewhere on the path. It does not identify who dropped it, and it does not prove the server was down.
11. What made the reader's case conclusive was the same request succeeding over mobile data seconds later. Same destination, different path, different result.

42.7.5 TECHNICAL - the engineer's version

1. The exact test to determine whether port 22 is blocked, in order, and each step distinguishes something specific.

```
# 1. Does the name resolve? (rules out DNS)
$ dig +short github.com
20.207.73.82

# 2. Can TCP reach port 22 at all? (the real question)
$ nc -vz -w 5 github.com 22

# 3. Same for 443, as a control
$ nc -vz -w 5 github.com 443

# 4. If nc is absent, bash can do it
$ time bash -c 'exec 3<>/dev/tcp/github.com/22'

# 5. GitHub's own SSH check, most direct of all
$ ssh -vT -o ConnectTimeout=10 git@github.com
```

2. Read the results this way:

Observation	Conclusion
22 refused fast	reachable, nothing there
22 silent, 443 ok	port 22 filtered
both silent	path or host problem
both fast, auth fails	network fine, key wrong

3. **Fix one: switch the remote to HTTPS.** One command per repository, or a global rewrite for all of them.

```
$ git remote set-url origin \
    https://github.com/owner/repo.git

# or globally, for every GitHub remote at once
$ git config --global \
    url."https://github.com/".insteadOf "git@github.com:"
```

4. **Fix two: use GitHub's SSH over the HTTPS port.** GitHub runs an SSH endpoint on port 443 at a different host name.
5. The host name is `ssh.github.com`, not `github.com`. This matters. GitHub's documentation states it explicitly because people get it wrong.
6. Test it first:

```
$ ssh -T -p 443 git@ssh.github.com
Hi USERNAME! You've successfully authenticated, but GitHub
does not provide shell access.
```

7. Then make it permanent with the exact block GitHub publishes, in `~/.ssh/config`:

```
Host github.com
    Hostname ssh.github.com
    Port 443
    User git
```

8. After that block, your existing `git@github.com:owner/repo.git` remotes need no change at all. `ssh` rewrites the destination before connecting.
9. This is real SSH, not SSH wrapped in TLS. It is the SSH protocol speaking on TCP port 443. A firewall filtering by port number is satisfied.
10. Therefore it does not help against a proxy that inspects protocols, only against a port-number filter. GitHub's own documentation notes that proxy servers may interfere.

11. Comparison of the two fixes, since they are not equivalent:

Aspect	HTTPS remote	SSH on 443
Credential	token	SSH key
Works via HTTP proxy	yes	usually not
Survives deep inspection	yes	no
Config scope	per repo or global	one config block

12. A third option exists and is worth knowing: `ProxyCommand` with a tool such as `corkscrew` or `nc` tunnels SSH through an HTTP proxy using the `CONNECT` method. It is fiddly and usually not needed if fix one works.
13. On the honest side: none of this is a way around a policy that forbids the traffic. Using SSH on 443 to evade a corporate control is a conversation with your employer, not a technical question.

42.7.6 WORDS - remember these

Port — a number naming a service on a machine — a 16-bit TCP or UDP endpoint identifier; 22 for SSH, 443 for HTTPS by IANA assignment.

DROP — block and say nothing — a filter action discarding the packet with no reply, producing a client-side timeout after retries.

REJECT — block and say so — a filter action returning a TCP RST or ICMP unreachable, producing an immediate `Connection refused`.

Egress filtering — controlling what may leave a network — outbound firewall policy, commonly permitting only 80, 443 and DNS.

`ssh.github.com` — GitHub's SSH server on the web port — the host name serving the SSH protocol on TCP 443 for networks blocking 22.

`ProxyCommand` — a custom way to reach a host — an `ssh_config` directive running an external program to establish the transport.

42.8 HTTPS authentication for git

42.8.1 PLAIN - in simple words

1. When git talks over HTTPS, it makes ordinary web requests, and it proves who you are the way web requests always have: a user name and a secret.
2. For many years that secret was your actual GitHub account password.
3. That stopped. On 13 August 2021, GitHub stopped accepting account passwords for git operations entirely.
4. The reason is straightforward. Your account password unlocks everything you own, cannot be limited, cannot be given to one script, and cannot be cancelled without changing it everywhere.
5. In its place you use a **token**, which is a long random string that acts like a password but has rules attached to it.
6. A token can be limited to certain actions. It can expire. It can be cancelled on its own without touching anything else.
7. Now the practical problem. You cannot type a forty-character random string before every push.
8. So git has **credential helpers**. A credential helper is a small program that stores your token and hands it back when git asks.
9. Where it stores it depends on the helper, and the differences matter. Some put it in your operating system's encrypted vault. One writes it to a plain text file.

10. Knowing which helper you use, and being able to look inside it and clear it, is a basic skill.

42.8.2 PLAIN - a picture in your head

1. Think of a building where the receptionist checks a pass every time you go through a door.
2. Carrying the pass in your hand and showing it forty times a day is tiring.
3. So you give it to a locker at the entrance. Whenever you approach a door, the locker sends the pass details for you.
4. There are different lockers. A steel one with a combination that only opens when you are logged in. And a cardboard box on the counter with your pass sitting in it, visible to anyone who looks.
5. Both work. Both save you the same effort. Only one is a good idea.
6. And when you lose your pass, or you suspect somebody copied it, you need to do two separate things.
7. Cancel the pass at the security office, so the old one stops working. And empty the locker, so your own building stops trying to use the dead one.

Where this comparison breaks:

8. A physical pass is one object. A token can be copied silently and used from anywhere in the world at the same time as you are using it, with no sign at the door.

42.8.3 PLAIN - a worked example

1. Real output from the machine used for this chapter, using git's own `credential` commands. These are the exact operations a helper performs.
2. Choose a helper. Here the insecure `store` helper is chosen deliberately, so we can look at what it writes:

```
$ git config --global credential.helper \
    'store --file=/tmp/demo/.git-credentials'
```

3. Save a credential. This is what happens automatically the first time you push and type a token:

```
$ printf 'protocol=https\nhost=github.com\nusername=reader
password=ghp_EXAMPLEexampleEXAMPLEexample1234\n\n' \
  | git credential approve
```

4. Now look at what landed on disk:

```
$ ls -l /tmp/demo/.git-credentials
-rw------- 1 root root 63 Aug 13 08:24 .git-credentials
$ cat /tmp/demo/.git-credentials
https://reader:ghp_EXAMPLEexampleEXAMPLE...@github.com
```

5. That is your token, in a plain readable file. The permissions are 600, owner only, but there is no encryption at all.
6. Ask git what it would send for a given host. This is the command to run when you suspect a stale credential:

```
$ printf 'protocol=https\nhost=github.com\n\n' \
  | git credential fill
protocol=https
host=github.com
username=reader
password=ghp_EXAMPLEexampleEXAMPLEexample1234
```

7. And clear it, which is what you do the moment a token is rotated or leaked:

```
$ printf 'protocol=https\nhost=github.com\nusername=reader
password=ghp_EXAMPLEexampleEXAMPLEexample1234\n\n' \
  | git credential reject
```

```
$ wc -c < /tmp/demo/.git-credentials
0
```

8. The file is now empty. Note carefully: this cleared the local copy. It did nothing whatsoever to the token on GitHub's side.

42.8.4 PLAIN – what is really happening inside

1. Git never asks a helper for a credential until it needs one.
2. It first makes the request without any credential. If the server is happy, nothing else happens. This is why cloning a public repository never prompts.
3. If the server responds with HTTP status 401 and a `WWW-Authenticate` header, git knows a credential is required.
4. Git then runs the configured helper with the subcommand `get`, feeding it the protocol, host and path on standard input.
5. The helper replies with `username=` and `password=` lines, or says nothing if it has none stored.
6. If no helper has one, git prompts you on the terminal.
7. Git then retries the request with an `Authorization: Basic` header. That header is the user name and token joined with a colon and base64-encoded.
8. Base64 is an encoding, not encryption. It hides nothing. What protects the token is that the whole request travels inside TLS.
9. If the retried request succeeds, git runs the helper again with `store`, so a working credential is remembered.
10. If it fails with 401, git runs the helper with `erase`, so a wrong credential is not offered forever.
11. That last step is why a bad cached token often self-corrects, and why a token that is valid but lacks a permission does not: that is a 403, not a 401, so git does not erase it.

```
git ---> GET /repo.git/info/refs        ---> server
git <--- 401 + WWW-Authenticate         <--- server
git ---> helper "get" (host, protocol)
git <--- helper "username=..., password=..."
git ---> GET again + Authorization: Basic ...
git <--- 200 OK
git ---> helper "store"  (remember this one)
```

42.8.5 TECHNICAL – the engineer's version

1. The credential helper protocol is a documented text protocol. A helper is any executable named `git-credential-NAME`, invoked with one of `get`, `store` or `erase`.
2. The helpers you will actually meet:

Helper	Storage	Encrypted
`osxkeychain`	macOS keychain	yes
`manager` (GCM)	OS vault per platform	yes
`libsecret`	GNOME keyring	yes
`wincred`	Windows Cred Manager	yes
`cache`	memory, 900 s default	n/a
`store`	`~/.git-credentials`	no

3. `store` is documented by git itself as insecure. Its own manual page says the credentials are stored unencrypted. Use it only on a machine where the whole disk is encrypted and single-user, and preferably not then.
4. `cache` keeps credentials in memory in a background daemon with a default timeout of 900 seconds, adjustable with `--timeout`.

5. Git Credential Manager, formerly Git Credential Manager Core, is the cross-platform helper maintained by GitHub. It handles OAuth device flow, so you get a browser login instead of pasting a token.
6. Configuration is layered, and this catches people out:

```
$ git config --show-origin --get-all credential.helper
file:/etc/gitconfig     manager
file:~/.gitconfig       osxkeychain
```

7. Multiple helpers are tried in order until one returns a credential. An empty value, `credential.helper=`, resets the list, which is the way to disable an inherited system-level helper.
8. Per-host configuration is possible and is the right way to keep work and personal accounts separate:

```
[credential "https://github.com"]
    username = work-account
    helper = osxkeychain
```

9. Inspecting and clearing on macOS, which is the reader's platform:

```
# see it
$ printf "protocol=https\nhost=github.com\n\n" \
  | git credential-osxkeychain get

# delete it
$ printf "protocol=https\nhost=github.com\n\n" \
  | git credential-osxkeychain erase
```

10. The graphical route on macOS is the Keychain Access application, searching for `github.com` and deleting the internet password entry.
11. On Windows: `cmdkey /list` in a command prompt, or Credential Manager in Control Panel, under Windows Credentials.
12. On Linux with libsecret: `secret-tool search server github.com`.
13. The HTTP conversation is worth seeing once. Real capture from this chapter's machine against a public repository, so no credential was needed:

```
GET /octocat/Hello-World.git/info/refs?service=git-upload-pack
Git-Protocol: version=2
<= HTTP/1.1 200 OK
<= Content-Type: application/x-git-upload-pack-advertisement
```

14. Against a repository the requester cannot see, unauthenticated, the same endpoint returns 401 rather than 404. Real result:

```
$ curl -s -o /dev/null -w "status=%{http_code}\n" \
  "https://github.com/octocat/not-a-real-repo.git/info/refs\
?service=git-upload-pack"
status=401
```

15. That 401 for a repository that does not exist is deliberate. GitHub will not confirm or deny the existence of a private repository to an anonymous requester. Remember this when we reach `Repository not found`.
16. The user name in an HTTPS git URL is almost always ignored by GitHub when a token is supplied. `x-access-token`, your login, or anything else usually works. The token identifies you.
17. Never put a token in the remote URL itself. It lands in `.git/config` in plain text, in your shell history, and in the output of `git remote -v`, which people paste into bug reports.

42.8.6 WORDS - remember these

Credential helper — a program that remembers your token — an executable implementing git's `get/store/erase` credential protocol.

Basic authentication — user name and secret sent in a header — HTTP authentication, RFC 7617, sending base64 of `user:secret`, safe only inside TLS.

401 versus 403 — "who are you" versus "not allowed" — 401 means no or invalid credential and triggers a prompt; 403 means the credential is valid but insufficient.

`store` helper — the one that writes plain text — a helper persisting credentials unencrypted in `~/.git-credentials`.

Keychain — the operating system's encrypted vault — macOS Keychain, Windows Credential Manager or a libsecret provider, unlocked with your login.

`git credential fill` — ask git what it would send — a plumbing command running the helper chain and printing the resolved credential.

42.9 Personal access tokens

42.9.1 PLAIN - in simple words

1. A token is a long random string that stands in for you when a program talks to a service.
2. It is not derived from your password. Cancelling it does not affect your password, and changing your password does not necessarily cancel it.
3. That independence is the point. A token is a separate identity you can hand to one script, on one machine, for one purpose.
4. A token has three properties a password does not have, and each one matters.
5. It can be **scoped**, meaning limited to a specific list of things it is allowed to do.
6. It can **expire**, meaning it stops working on a date you chose, whether or not you remember it exists.
7. It can be **revoked individually**, meaning you cancel this one without disturbing the other nine tokens on your other nine machines.
8. GitHub offers two kinds. The older **classic** tokens, which use a list of broad permission names. And **fine-grained** tokens, which pick individual repositories and individual permissions.
9. A leaked token is an emergency, not an inconvenience. Anyone holding it is you, for everything that token can reach, immediately.
10. There is no clever recovery. You revoke it, you create a new one, you replace it everywhere, and you check what was done while it was out.

42.9.2 PLAIN - a picture in your head

1. Think of a hotel with two kinds of key.
2. The master key opens every room, the safe, the office and the roof. That is your account password.
3. The other is a plastic card cut for room 412 only, valid until Friday, and the front desk can cancel it in one click without recutting anyone else's.
4. If a cleaner needs into room 412 on Tuesday, you give them the card, not the master key.
5. If the card is lost, you cancel it. Nothing else changes. Nobody else is inconvenienced.
6. If the master key is lost, the entire hotel has to be rekeyed.
7. Now add the part that has no hotel equivalent. A copied card is invisible. The lock cannot tell the difference and neither can you.
8. So the card also has a printed expiry date, so that a copy you never learned about dies on its own.

Where this comparison breaks:

9. A hotel card must be physically present at the door. A token can be used from another continent, by a program, thousands of times a minute.
10. And a hotel lock logs the card number. Token use is logged too, but you have to go and look, and almost nobody does until something breaks.

42.9.3 PLAIN - a worked example

1. GitHub tokens begin with a short prefix that tells you what kind they are. This is a deliberate design so that scanners can find them.

Prefix	What it is
`ghp_`	classic personal token
`github_pat_`	fine-grained personal token
`gho_`	OAuth app user token
`ghu_`	GitHub App user token
`ghs_`	GitHub App server token
`ghr_`	refresh token

2. Using one at the command line, once, without saving it anywhere:

```
$ git clone https://TOKEN@github.com/owner/repo.git
```

3. That works, and it is a bad habit, because the token is now in the shell history and in `.git/config`.
4. The better single-use form pipes it in and never stores it:

```
$ echo "TOKEN" | gh auth login --with-token
```

5. Rotating a token is a fixed procedure. Do these in order, because the order prevents an outage:
6. One: create the new token with the same scopes, and note the expiry.
7. Two: update every place that uses it. Your keychain, your CI secrets, the `.netrc` on the build box, the deployment config.
8. Three: verify the new one works, with an actual operation, not a guess.
9. Four: only now, revoke the old one.
10. Five: watch for failures for a day, because there is always one forgotten machine.
11. If instead the token has leaked, that order inverts. Revoke first, immediately, and accept the outage. A live leaked token is worse than downtime.

42.9.4 PLAIN - what is really happening inside

1. A token is a record in GitHub's database. It has an owner, a creation time, an expiry time, a list of permissions and a last-used timestamp.
2. What you are shown once, at creation, is the token string. GitHub does not store that string in a readable form; it stores a hash of it.
3. That is why the page says you will never see it again. They genuinely cannot show it to you.
4. When you use it, the server hashes what you sent and looks up the record. If the record exists, is not expired and is not revoked, you are authenticated.
5. Then, and only then, does it check whether the permissions on that record allow the specific operation you asked for. Those are two separate steps.
6. Keeping them separate explains the whole of section 42.10. Being authenticated does not mean being allowed.
7. **Secret scanning** is GitHub scanning content pushed to it, looking for strings that match known credential patterns.
8. This is why the prefixes exist. `ghp_` followed by the right number of base62 characters with a checksum is machine-recognisable with almost no false positives.

9. GitHub runs the same scanning for over five hundred partner patterns: cloud provider keys, payment provider keys, and so on.
10. When GitHub finds its own token in a public repository, it revokes it automatically. When it finds a partner's, it notifies the partner, who typically revokes it too.
11. **Push protection** is the same detection moved earlier. Instead of finding the secret after it is pushed, the push itself is refused.
12. That distinction is enormous. A secret that reached the server must be treated as leaked forever, even after you rewrite history, because the object may have been fetched or cached.
13. A secret blocked by push protection never arrived. Nothing leaked. You fix your commit and move on.
14. Push protection has been enabled by default for public repositories since 2024. For private repositories it is part of GitHub's paid secret protection product.

42.9.5 TECHNICAL - the engineer's version

1. Classic versus fine-grained, precisely:

Property	Classic	Fine-grained
Scope model	broad named scopes	per-permission
Repository limit	all you can access	chosen list
Owner limit	all orgs	one user or org
Org approval	no	can be required
Expiry	optional	policy-controlled

2. GitHub's own documentation states the case bluntly: a classic token "will grant access to all repositories within the organizations that you have access to, as well as all personal repositories".
3. That sentence is the whole argument for fine-grained tokens. A classic token on a laptop is a key to everything that laptop's owner can reach.
4. Fine-grained tokens are limited to resources owned by a single user or organization, can be limited to specific repositories, and are granted specific permissions rather than broad scopes.
5. Organization owners can require approval before a fine-grained token may act on organization resources, which is a control classic tokens never had.
6. Expiry: GitHub supports preset periods and a custom date. Fine-grained tokens gained optional non-expiring lifetimes and rotation policies in a change announced on 18 October 2024, and enterprises can enforce maximum lifetimes.
7. GitHub automatically removes personal access tokens that have not been used for a year. That is a useful backstop, not a strategy.
8. Practical expiry guidance, and this is a convention rather than a rule:

Use	Suggested lifetime
Interactive laptop use	90 days
A one-off script	7 days
A long-running service	do not; use an App
CI pipeline	do not; use OIDC

9. Token format, since GitHub's April 2021 redesign: a fixed prefix, an underscore, then base62 characters, ending in a checksum. The checksum lets a scanner reject a random-looking string cheaply before doing any lookup.
10. Auditing what a token can do, without guessing. The response headers of an authenticated API call name the scopes:

```
$ curl -sI -H "Authorization: Bearer $TOKEN" \
    https://api.github.com/user | grep -i x-oauth
x-oauth-scopes: repo, read:org
x-accepted-oauth-scopes:
```

11. `x-oauth-scopes` is what your token has. `x-accepted-oauth-scopes` is what that particular endpoint would accept. Comparing the two diagnoses most permission failures in one command.
12. With the GitHub CLI the equivalent is `gh auth status`, which prints the scopes of the stored token.
13. Leak response, in order, and treat this as a checklist:
14. Revoke the token in GitHub settings. This is instant and unconditional.
15. Rotate anything the token could have reached: deploy keys, downstream secrets, database passwords stored in repositories it could read.
16. Review the account's security log and, for organizations, the audit log, filtering by the period the token was live.
17. Check for new SSH keys, new deploy keys, new webhooks, new collaborators and changed workflow files. Those are the standard persistence mechanisms.
18. Rewriting git history does not undo a leak. Assume the secret is public from the moment it was pushed.
19. Never commit a token, even to a private repository. Private repositories get forked, made public, and cloned onto laptops that later get lost.

42.9.6 WORDS - remember these

Personal access token (PAT) — a password substitute you can limit and cancel — a server-side credential record with scopes, expiry and independent revocation.

Classic token — the older broad kind — a PAT carrying OAuth-style scopes across every repository and organization the user can reach.

Fine-grained token — the newer narrow kind — a PAT limited to one owner, a chosen repository list and individual permissions.

Revocation — cancelling one credential — deleting the server-side token record, taking effect immediately and independently of other credentials.

Secret scanning — the service that hunts for leaked credentials — pattern matching over pushed content, with automatic revocation for GitHub's own tokens.

Push protection — refusing the push instead of cleaning up after it — pre-receive secret detection that blocks the write, so nothing ever leaks.

42.10 Scopes, properly

42.10.1 PLAIN - in simple words

1. A **scope** is a named permission attached to a credential.
2. The name is a short string, like `repo` or `workflow`. It sits in the token's record on the server, not in anything you hold.
3. The server checks it at the exact moment you try to do something. Not when you log in. At the moment of the action.
4. That timing is the thing to understand. You can be perfectly, provably authenticated and still be refused.
5. Authentication answers "who are you". Authorization answers "may you do this". They are separate questions asked at different times.
6. So a token with no `workflow` scope will happily let you clone, fetch, and push ordinary changes. It will refuse one specific kind of push.
7. The reader hit exactly this. A push that changed a file under `.github/workflows/` was refused, while everything else worked.

8. The reason is not arbitrary. A file under `.github/workflows/` is not ordinary content. It is instructions for GitHub's own machines.
9. Those machines run with access to the repository's secrets. So changing that file is changing what code runs with those secrets.
10. GitHub therefore treats it as a separate, higher permission that you must ask for on purpose.

42.10.2 PLAIN - a picture in your head

1. Imagine an office pass that opens the front door, the stairs and your own floor.
2. You walk in, you go up, you work. Everything is fine. You are definitely a recognised employee.
3. Then you try the server room. The reader flashes red. Not because your pass is fake, but because your pass does not include that room.
4. Nobody has questioned your identity. They know exactly who you are. The answer is still no.
5. Now the important part: why is the server room separate at all?
6. Because inside it you could reconfigure the machines that everyone else's work runs on, and give yourself access to everything they touch.
7. That is precisely the relationship between a normal file and a CI workflow file. One is a document. The other is a change to the factory.

Where this comparison breaks:

8. A door either opens or it does not, and it decides in one instant. A scope check can depend on the content of what you are pushing, which path it touches, and even whether an identical file already exists on another branch.

42.10.3 PLAIN - a worked example

1. The scopes that actually matter in daily work. These are the exact names GitHub uses.

Scope	What it grants
`repo`	full read/write, private too
`public_repo`	same, public repos only
`workflow`	add/update workflow files
`read:org`	read org and team membership
`write:packages`	publish packages
`read:packages`	download packages
`admin:repo_hook`	manage repository webhooks
`gist`	write access to gists

2. GitHub's own words for the two that cause the most confusion.
3. `repo` "grants full access to public and private repositories including read and write access to code, commit statuses, repository invitations, collaborators, deployment statuses, and repository webhooks".
4. `workflow` "grants the ability to add and update GitHub Actions workflow files". And then a detail almost nobody knows, quoted exactly.
5. "Workflow files can be committed without this scope if the same file (with both the same path and contents) exists on another branch in the same repository."
6. That exception explains a confusing symptom. Merging a branch that already contains the workflow file can succeed while the original push failed.
7. Note that `repo` does not include `workflow`. It is not a hierarchy. The broadest normal scope still stops at the workflow directory.
8. `read:org` is the one people forget. Without it, a token cannot list the organizations you belong to, so tools report you as having no organizations at all.

42.10.4 PLAIN - what is really happening inside

1. Here is the reader's exact failure, and the message is worth reading character by character.

```
$ git push origin main
Enumerating objects: 7, done.
Writing objects: 100% (4/4), 412 bytes, done.
To https://github.com/owner/repo.git
 ! [remote rejected] main -> main (refusing to allow a
   Personal Access Token to create or update workflow
   `.github/workflows/ci.yml` without `workflow` scope)
error: failed to push some refs to
'https://github.com/owner/repo.git'
```

2. Read what already succeeded before the refusal, because it tells you where the failure was not.
3. `Enumerating objects` and `Writing objects` both completed. The network worked. TLS worked. Authentication worked.
4. Your commits were transmitted to GitHub. All of them. The bytes arrived.
5. The refusal came from `git-receive-pack` on GitHub's side, after receiving everything, when it inspected what was about to be applied.
6. Then it declined to update the branch reference. Nothing was written. The objects were transmitted and discarded.
7. That is what `! [remote rejected]` means, and it is why this is not a transport failure. A transport failure cannot produce a sentence about scopes, because a broken transport cannot produce sentences at all.
8. The phrasing `refusing to allow` is also deliberate. GitHub is not saying it failed. It is saying it decided.
9. Here is the same shape reproduced locally on the machine used for this chapter, with a server-side hook standing in for GitHub's own check. Real output:

```
$ git push origin main
remote: error: refusing to allow a Personal Access Token
remote: to create or update workflow
remote: `.github/workflows/ci.yml` without `workflow` scope
To /tmp/gitgate/srv/repo.git
 ! [remote rejected] main -> main (pre-receive hook declined)
error: failed to push some refs to '/tmp/gitgate/srv/repo.git'
```

10. And the push immediately before it, which touched only `README.md`, succeeded:

```
$ git push origin main
To /tmp/gitgate/srv/repo.git
 * [new branch]      main -> main
```

11. Same server, same credential, same network, same second. The only variable was which file the commit touched.
12. That comparison is the proof that this is a content-dependent authorization decision, taken on the server.

42.10.5 TECHNICAL - the engineer's version

1. The fix, in order, and it is short:
2. Open GitHub, Settings, Developer settings, Personal access tokens, Tokens (classic). Select the token in use.
3. Tick `workflow`. Save. The scope is added to the existing token; you do not have to create a new one for classic tokens.
4. If the token was already cached, git will keep using the same string, and it now has the scope. No local change is needed.

5. If you create a new token instead, clear the cached one first, or git will keep sending the old one:

```
$ printf "protocol=https\nhost=github.com\n\n" \
  | git credential reject
```

6. With the GitHub CLI, re-authorize with the extra scope in one command:

```
$ gh auth refresh -h github.com -s workflow
```

7. For fine-grained tokens, the equivalent is the repository permission "Workflows: Read and write".
8. For a GitHub App, it is the `workflows` repository permission, and the app must be installed on the repository.
9. Verify before retrying, rather than pushing and hoping:

```
$ curl -sI -H "Authorization: Bearer $TOKEN" \
    https://api.github.com/user | grep -i x-oauth-scopes
x-oauth-scopes: repo, workflow
```

10. **Now generalise, because all three failures print "push failed".** This table is the most useful thing in the chapter.

Layer	Symptom	Proof
Network	hangs then times out	no bytes sent
Auth	asks again, or 401	`Invalid username`
Authorization	refused with a reason	`[remote rejected]`
Non-fast-forward	refused, no credential talk	`[rejected]`

11. Distinguish them by four observations, in this order.
12. Did objects transfer? `Writing objects: 100%` means the network and the credential both worked. Everything after that is a policy decision.
13. Was a credential requested? A prompt or a 401 means authentication failed, not authorization.
14. Is there a reason in parentheses after `! [remote rejected]`? A sentence means a server rule. A bare `[rejected]` with `non-fast-forward` means your branch is behind and no permission is involved.
15. How long did it take? Instant means somebody answered. Fifteen to thirty seconds of silence means the network.
16. A worked triage, applying the table to the four cases:

```
Case A: 30 s, "Connection timed out"
  -> network. Port filtered. See 42.7.

Case B: "remote: Invalid username or password"
  -> authentication. Token wrong, expired or revoked.

Case C: "! [remote rejected] ... without `workflow` scope"
  -> authorization. Credential fine. Add the scope.

Case D: "! [rejected] ... (non-fast-forward)"
  -> neither. Fetch and rebase or merge.
```

17. One further separation the reader met in the same session. Plain git kept working while the GitHub API did not.
18. That is the same split at a larger scale: git the tool, running locally against local objects, versus GitHub the service, reachable only over the network and enforcing its own rules.

19. The honest version: the `workflow` scope is not a complete defence. Someone with `repo` scope can still change a file that a workflow executes, such as a build script the workflow calls. The scope protects the workflow definition, not everything the workflow does.
20. That is why serious repositories also use branch protection, required reviews on the `.github/` path with a CODEOWNERS file, and environment protection rules on the jobs that hold real secrets.

42.10.6 WORDS - remember these

Scope — a named permission on a credential — a string in the token record that the server checks per operation, such as `repo` or `workflow`.

Authentication — establishing who you are — verifying a credential against a stored record; failure produces 401.

Authorization — establishing what you may do — checking permissions for a specific operation; failure produces 403 or a rejection message.

`! [remote rejected]` — the server chose not to apply this — a per-ref failure from `git-receive-pack`, always accompanied by a reason.

Non-fast-forward — your branch is behind, not forbidden — a rejection because the update would discard commits, unrelated to any permission.

`x-oauth-scopes` — the header that lists what your token has — a GitHub API response header naming the token's granted scopes.

42.11 OAuth, tokens, SSH keys and GitHub Apps

42.11.1 PLAIN - in simple words

1. There are four different ways a program or a person can be identified to GitHub, and people mix them up constantly.
2. An **SSH key** identifies a machine you sit at. It lives in a file on that machine and never moves.
3. A **personal access token** identifies you as a person, to a program you gave it to. It is a string, so it can be copied.
4. An **OAuth authorization** is what happens when you click "sign in with GitHub" on someone else's website. You grant that site permission to act as you.
5. A **GitHub App** is a separate identity of its own. It is not you. It is installed on specific repositories and acts in its own name.
6. The difference that matters most is **blast radius**: if this credential leaks, how much can the finder reach, and for how long.
7. An SSH key with no passphrase, sitting on a stolen laptop, reaches every repository you can reach, forever, until you notice.
8. A classic token reaches the same, from anywhere in the world, with no laptop required.
9. A GitHub App installation token reaches only the repositories the app is installed on, only with the permissions granted, and it dies after one hour.
10. For automated systems there is a fifth and better option that involves no stored secret at all. We come to it at the end of this section.

42.11.2 PLAIN - a picture in your head

1. Four ways to get a package delivered into a building.
2. The SSH key is a key cut for your own front door. It works only when you are standing there. Losing it is bad but local.
3. The personal token is your signature on a delivery authorization. Anyone with a photocopy can use it anywhere, until you cancel it.

4. The OAuth grant is you signing a standing instruction that a courier company may collect on your behalf. You can withdraw it, but while it stands, they act as you.
5. The GitHub App is a contractor with their own uniform, their own badge, and a written list of which rooms they may enter. When they do something, the log says the contractor did it, not you.
6. That last property is underrated. When a person leaves the company, their key, token and OAuth grants all die with their account. The contractor keeps working.

Where this comparison breaks:

7. A contractor can be phoned and asked what they are doing. An app acts in milliseconds, thousands of times, and the only record is the audit log.

42.11.3 PLAIN – a worked example

1. The four identities compared on the properties that decide which to use.

Identity	Acts as	Typical lifetime
SSH key	you	years, until removed
Classic PAT	you	30 to 90 days
OAuth grant	you, via an app	until revoked
App install token	the app	1 hour

2. And the blast radius if each one leaks:

Identity	Reach if leaked
SSH key	all repos you can access
Classic PAT	all repos, plus scopes
Fine-grained PAT	listed repos only
App install token	installed repos, 1 hour

3. Choosing, as a short decision list:
4. A human on their own laptop, working daily: an SSH key with a passphrase, held in the agent.
5. A human running a script occasionally: a fine-grained token, short expiry, limited to the repositories the script touches.
6. A server that must pull one repository: a deploy key, read-only, on that server only.
7. A product integrating with GitHub for many customers: a GitHub App.
8. A CI pipeline talking to a cloud provider: neither of these. Use OIDC, which is the next block.

42.11.4 PLAIN – what is really happening inside

1. The problem with a long-lived credential in a CI system is that it must be stored somewhere the CI system can read.
2. Which means anyone who can make the CI system print things, or run arbitrary code, can read it. And CI systems exist to run code that people wrote today.
3. The modern answer removes the stored secret entirely. It is called **OIDC federation**, and the idea is simple even if the acronyms are not.
4. When a job starts, GitHub creates a short-lived signed statement describing that exact job. Which repository, which branch, which workflow, which run.
5. That statement is signed by GitHub, and anyone can check the signature against GitHub's published verification keys.
6. The job hands that statement to the cloud provider and says: I am this workflow in this repository, please give me credentials.

7. The cloud provider has been told in advance which statements to accept. For example, only jobs from `owner/repo` on the `main` branch.
8. It verifies the signature, checks the description matches its rule, and issues its own short-lived credential, valid for perhaps an hour.
9. Nothing long-lived was ever stored. There is no secret in the repository, no secret in the CI settings, and nothing to rotate.
10. If someone steals the statement, it expires in minutes and only works for that one narrow rule.

```
GitHub Actions job
  |  1. request token for audience "aws"
  v
GitHub OIDC issuer
  |  2. signed JWT: repo, ref, workflow, run_id
  v
the job
  |  3. present JWT to the cloud provider
  v
cloud provider
  |  4. verify signature, match trust policy
  |  5. issue short-lived credentials
  v
the job now has 1-hour credentials, no stored secret
```

42.11.5 TECHNICAL - the engineer's version

1. **SSH keys** authenticate a client to a server using public key cryptography. On GitHub they are attached to a user account or, as deploy keys, to a repository. They carry no scopes: an account SSH key can do whatever that account can do over git.
2. That is a genuine limitation. There is no such thing as a read-only account SSH key on GitHub. If you need read-only, you need a deploy key or a fine-grained token.
3. **Personal access tokens** are bearer credentials. Possession is authorization. They carry scopes and expiry as covered in 42.9 and 42.10.
4. **OAuth** is the delegation framework in RFC 6749, October 2012, with the current best-practice profile in RFC 9700, January 2025. Three parties: you the resource owner, the app, and GitHub the authorization server.
5. The user-facing difference from a token is that you never hand the app your credential. You are redirected to GitHub, you approve a named list of scopes, and the app receives a token bound to that grant.
6. OAuth App tokens act entirely as you. Every action appears in the audit log as your action. Revoking the grant kills the token.
7. **GitHub Apps** are first-class actors. Key properties:

Property	Value
Identity	the app, not a user
Install scope	chosen repositories
Permissions	per-resource read/write
Install token life	1 hour
Rate limit	scales with installation

8. A GitHub App authenticates in two stages. It signs a JSON Web Token with its own private key to prove it is the app, then exchanges that for an installation access token scoped to one installation.
9. That installation token expires after one hour, which is stated in GitHub's own documentation as one of the reasons to prefer apps over deploy keys.
10. Apps also survive staff turnover. A token or key tied to a departing employee dies when the account is deprovisioned, usually at three in the morning, in the middle of a deployment.

11. **OIDC federation for CI.** OpenID Connect is an identity layer on top of OAuth 2.0. GitHub Actions runs an OIDC provider that issues a JSON Web Token per job.
12. Claims in that token include `sub`, which encodes the workflow context such as `repo:octo-org/octo-repo:environment:prod`, plus `aud`, `repository`, `actor`, `ref`, `workflow` and `run_id`.
13. The cloud side is configured with a trust policy matching those claims. A correct policy pins at least the repository and usually the branch or environment.
14. GitHub states three benefits: no cloud secrets duplicated as long-lived GitHub secrets, granular control through the provider's own access tools, and automatic rotation because each token expires when the job ends.
15. A minimal workflow needs one permission block and one action:

```
permissions:
  id-token: write
  contents: read
```

16. Without `id-token: write` the job cannot request an OIDC token at all, and the failure message is unhelpful, so this is worth memorizing.
17. `GITHUB_TOKEN`, the automatic token inside every Actions run, is itself an installation token for a GitHub App. Its permissions are set by the `permissions:` key, and any action in the job can read it via `github.token` even if you never pass it.
18. A common trap follows from that. The `workflow` scope discussion in 42.10 applies here too: `GITHUB_TOKEN` deliberately cannot push changes to workflow files, and cannot trigger further workflow runs, to prevent infinite loops and privilege escalation.
19. The honest version, and it is a real trade-off. OIDC federation is more setup than pasting a token. For a one-person hobby project a short-lived fine-grained PAT is a reasonable choice. For anything with real secrets, the setup cost is paid once and removes a whole class of incident.

42.11.6 WORDS - remember these

Bearer credential — whoever holds it, is you — a token requiring no proof of possession beyond presenting it.

OAuth — letting an app act for you without your password — the delegation framework of RFC 6749, exchanging a user approval for a scoped token.

GitHub App — a robot with its own identity and badge — an installable actor with per-repository permissions and one-hour installation tokens.

Installation access token — an app's short-lived key to one installation — a token valid for 1 hour, scoped to the repositories the app is installed on.

OIDC federation — proving who you are instead of holding a secret — exchanging a signed job description for short-lived cloud credentials.

Blast radius — how much damage one leak causes — the set of resources and the time window a compromised credential can affect.

42.12 Practical hardening for a developer's own machine

42.12.1 PLAIN - in simple words

1. Most security advice is too long to follow. Here is the short version that actually pays for itself.
2. Put a passphrase on every private key. It is the difference between a stolen laptop being embarrassing and being catastrophic.
3. Use the SSH agent so the passphrase costs you one typing per day, not one per push.

4. Make a different key on every machine. Never copy a private key between machines, not even your own.
5. The reason is simple. If you lose one machine, you remove one key. If you copied one key everywhere, you have to replace it everywhere at once, in a hurry, while panicking.
6. Look at your list of keys and tokens once or twice a year and delete the ones you cannot account for.
7. On any server you run yourself, turn password logins off entirely. Keys only.
8. Install a small program that bans addresses which repeatedly fail to log in. It is not clever, but it removes most of the noise.
9. Moving SSH to a port other than 22 stops the automated scanning almost completely. Be honest that this hides the door, it does not lock it.
10. And know in advance what you will do when a key or token leaks, because the first ten minutes decide how bad it gets.

42.12.2 PLAIN - a picture in your head

1. Think of the difference between a bicycle lock and a bicycle in a locked shed.
2. A passphrase on your key is the lock. Someone who takes the bicycle still cannot ride away immediately.
3. Separate keys per machine is owning three cheap bicycles instead of one expensive one. Losing one is annoying, not ruinous.
4. Disabling password login on your server is removing the letterbox that people keep pushing lock-picks through.
5. Banning repeat offenders is a doorman who remembers faces.
6. Changing the port is painting the shed to look like a hedge. Any determined person who walks the whole street still finds it.
7. That last one is worth doing anyway, not because it stops attackers, but because it stops your log file filling with thousands of pointless entries and hiding the one that matters.

Where this comparison breaks:

8. A thief must be physically present. An attacker scans the entire internet's address space in hours, so "nobody would bother with my little server" is simply false. Everything is scanned, all the time, automatically.

42.12.3 PLAIN - a worked example

1. A complete setup for a new machine, in order, with real commands.

```
# 1. one key, this machine, with a passphrase
$ ssh-keygen -t ed25519 -C "you@laptop-2026"

# 2. keep it in the agent so you type it once
$ ssh-add -t 8h ~/.ssh/id_ed25519

# 3. check what the agent holds
$ ssh-add -l
256 SHA256:WH1E2gSLhG6FgMsha//GINoq... (ED25519)

# 4. add the public half to GitHub, then test
$ ssh -T git@github.com
```

2. A minimal safe `~/.ssh/config` for that machine:

```
Host *
    AddKeysToAgent yes
    IdentitiesOnly yes
    ServerAliveInterval 60
    HashKnownHosts yes
```

3. On a server you own, the three lines in `/etc/ssh/sshd_config` that matter most:

```
PasswordAuthentication no
KbdInteractiveAuthentication no
PermitRootLogin prohibit-password
```

4. Always test in a second terminal before closing the first one. Locking yourself out of a remote machine is a rite of passage that is best skipped.
5. Adding a passphrase to a key that has none, without regenerating it:

```
$ ssh-keygen -p -f ~/.ssh/id_ed25519
Enter old passphrase:
Enter new passphrase:
```

6. Checking whether a key file is encrypted at all. An unencrypted OpenSSH private key contains the string `none` twice near the start of the base64, for the cipher and the key derivation function.

42.12.4 PLAIN - what is really happening inside

1. A passphrase does not scramble the whole key file with the passphrase directly. That would be weak, because people choose short passphrases.
2. Instead the passphrase is fed through a slow, deliberately expensive function to derive an encryption key. Then that derived key encrypts the private key.
3. Slow is the point. If each guess costs a fraction of a second instead of a microsecond, a guessing attack becomes hundreds of thousands of times more expensive.
4. When you type the passphrase, the same derivation runs, the file is decrypted in memory, and the plain key is handed to the agent.
5. From then until the agent is emptied or stopped, the unlocked key exists in that process's memory and nowhere else on disk.
6. A ban tool such as `fail2ban` works by reading the authentication log, matching failure lines with a pattern, counting them per source address, and inserting a firewall rule when a threshold is crossed.
7. It is not intelligent and it does not need to be. Automated scanners do not adapt; they move to the next address.
8. Changing the port works for the same unglamorous reason. Mass scanners prioritise port 22 because that is where the yield is. A different port is still found by a full port scan, just far less often.

42.12.5 TECHNICAL - the engineer's version

1. Key encryption in the OpenSSH format uses `bcrypt_pbkdf`, a password-based key derivation function built on bcrypt, introduced with the new private key format in OpenSSH 6.5, January 2014.
2. The work factor defaults to 16 rounds and is adjustable at generation time:

```
$ ssh-keygen -t ed25519 -a 100 -C "you@laptop"
```

3. `-a 100` sets 100 KDF rounds. It makes unlocking take noticeably longer for you, and proportionally longer for anyone brute-forcing the file.
4. Since OpenSSH 7.8, August 2018, the OpenSSH format is the default for all key types, replacing the old PEM format which used a weak MD5-based derivation.
5. Server hardening, the directives that matter and what each removes:

Directive	Value	Removes
`PasswordAuthentication`	`no`	guessing attacks
`PermitRootLogin`	`prohibit-password`	direct root by password
`MaxAuthTries`	`3`	long guessing sessions
`AllowUsers`	list	logins by other accounts

Directive	Value	Removes
`PermitEmptyPasswords`	`no`	the worst case
`LoginGraceTime`	`30`	held-open connections

6. Validate before restarting, every time:

```
$ sudo sshd -t
$ sudo systemctl reload ssh
```

7. `fail2ban` reads `/var/log/auth.log` on Debian-family systems or the journal on others. A typical jail:

```
[sshd]
enabled  = true
port     = 22
maxretry = 4
findtime = 600
bantime  = 3600
```

8. That bans an address for one hour after four failures within ten minutes. With `PasswordAuthentication no` already set, the practical benefit is log volume rather than security, which is still a real benefit.
9. `sshguard` and `CrowdSec` are alternatives; `CrowdSec` additionally shares reputation data across participants.
10. Changing the port. State the honest position clearly, because both exaggerations are common.
11. It is **security through obscurity**. It provides no protection against anyone who scans your ports, which any real attacker does.
12. It does reliably remove the overwhelming majority of automated noise, because mass scanners target 22 by default. Measured on typical internet- facing hosts, failed login attempts commonly drop by more than 95 percent.
13. So the correct framing is: it is a log hygiene measure with a security side effect, and it must never be the only measure.
14. Note also that ports below 1024 require root to bind, so a high port such as 2222 is bindable by an unprivileged process, which is a small argument against picking a high port on a shared machine.
15. Key rotation, as a routine rather than an emergency:

Credential	Suggested cycle
SSH key, laptop	yearly, or on loss
SSH key, server	on rebuild
Classic PAT	90 days
Deploy key	yearly
App private key	yearly

16. **Leak response**, and this is the part to rehearse. The order matters more than the speed.
17. For an SSH key: remove the public key from every account and server that has it. GitHub's SSH key page shows a last-used date for each key, which is the fastest way to spot one you do not recognise.
18. For a token: revoke it in settings. Instant.
19. Then, in both cases, review what was done. Check the account security log, the organization audit log, and the repository's recent pushes.
20. Look specifically for the standard persistence tricks: a newly added SSH key, a new deploy key, a new webhook, a new collaborator, a changed workflow file, or a new GitHub App installation.
21. Then rotate everything the compromised credential could read. A token with `repo` scope could read every secret ever committed to any repository it reached, including ones you deleted from the working tree but not from history.

22. Finally, write down what happened and how the credential got out. Almost every leak is one of four things: committed to a repository, pasted into a chat or ticket, printed in CI logs, or left on a machine that was lost.

42.12.6 WORDS - remember these

Passphrase — a secret that unlocks your key file — a user secret fed through a KDF to derive the key that encrypts the private key at rest.

`bcrypt_pbkdf` — the slow function guarding your key file — the KDF used by the OpenSSH private key format, with an adjustable rounds parameter.

`fail2ban` — a doorman that remembers repeat offenders — a log-scanning daemon inserting temporary firewall bans after repeated authentication failures.

Security through obscurity — hiding rather than locking — a measure that raises the cost of finding a service without raising the cost of attacking it.

Key rotation — replacing credentials on a schedule — periodic reissue and retirement, limiting the useful lifetime of an undetected compromise.

Persistence — what an attacker leaves behind — added keys, hooks, collaborators or workflow changes that survive the original credential being revoked.

42.13 A diagnosis table for git authentication failures

42.13.1 PLAIN - in simple words

1. Almost every git authentication problem announces itself with one specific sentence.
2. If you learn to read those sentences, you stop guessing and start fixing.
3. There are three questions to ask in order, and they narrow it down fast.
4. First: how long did it take? Instant means somebody replied. Half a minute of silence means the network.
5. Second: did it ask for a user name or password? That means it wanted a credential and either had none or was told the one it had was wrong.
6. Third: did it mention SSH keys, or a repository name, or a scope? Each of those points at a different layer.
7. The most misleading message of all is `Repository not found`, because it usually does not mean the repository is missing.
8. It usually means your credential is not allowed to see it, and the server is refusing to confirm that it exists.

42.13.2 PLAIN - a picture in your head

1. Think of a doctor listening to a description of symptoms.
2. "It hurts when I press here" narrows things down enormously, far more than "I feel unwell".
3. Error messages are the same. `Permission denied (publickey)` is "it hurts when I press here". `Push failed` is "I feel unwell".
4. So the first job is always to find the most specific line in the output, not the last line.
5. Git prints the summary last and the cause first. People read the last line and miss the answer three lines above it.

Where this comparison breaks:

6. A patient can be asked follow-up questions. A one-shot error message cannot, which is why the verbose flags exist.

42.13.3 PLAIN - a worked example

1. Get more information before guessing. Two flags do almost everything.

```
# for SSH remotes, ask ssh to narrate
$ GIT_SSH_COMMAND="ssh -vvv" git push origin main

# for HTTPS remotes, ask git to narrate
$ GIT_TRACE=1 GIT_CURL_VERBOSE=1 git push origin main
```

2. And the single most useful check for SSH, which tests only the connection and touches no repository:

```
$ ssh -T git@github.com
Hi USERNAME! You've successfully authenticated, but GitHub
does not provide shell access.
```

3. That message is a success. The words "does not provide shell access" alarm people every time. GitHub is not a shell server; it only runs git commands.
4. If that greets you by name, your key and your network are both fine, and any remaining problem is about a specific repository or a specific push.

42.13.4 PLAIN - what is really happening inside

1. The reason one problem produces several different messages is that several independent programs are involved, each with its own vocabulary.
2. Your kernel produces `Connection refused` and `Connection timed out`. These are TCP-level outcomes and mention no names.
3. `ssh` produces `Permission denied (publickey)` and `Host key verification failed`. These are SSH-level and mention keys.
4. `curl` inside git produces HTTP status codes and TLS errors.
5. GitHub's own server produces every line beginning with `remote:`. Those are the only messages that reflect a decision by GitHub rather than a failure of plumbing.
6. So the prefix tells you the layer. A line starting with `remote:` came from the far end and means you got there. A line starting with `ssh:` means you did not.

42.13.5 TECHNICAL - the engineer's version

1. The table. Each row is an exact string you will see.

Error text	Meaning
`Permission denied (publickey)`	key not accepted
`Invalid username or password`	token wrong or gone
`without workflow scope`	scope missing
`Host key verification failed`	known_hosts mismatch
`Connection timed out`	packets dropped
`port 22: Connection refused`	reached, nothing there
`Repository not found`	usually no permission
`non-fast-forward`	branch behind, not auth

2. Now each one in full, with the action to take.
3. **`git@github.com: Permission denied (publickey).`** SSH connected and the server rejected every key offered. Causes, in order of likelihood: the public key was never added to the account; the agent is empty; the wrong key is being offered; `IdentityFile` points at the public key by mistake; on your own server, wrong permissions on `~/.ssh` or `authorized_keys`. Check with `ssh -T git@github.com`

and `ssh-add -l`. On a server you control, read the server log, which is the only place `bad ownership or modes` appears.

4. **`remote: Invalid username or password.`** or **`remote: Support for password authentication was removed on August 13, 2021.`** The HTTPS credential is not valid. Either it is an account password, which has not worked since that date, or the token expired, was revoked, or was for a different account. Clear the cached credential with `git credential reject` or the platform keychain command, then retry so you are prompted afresh.
5. **`! [remote rejected] main -> main (refusing to allow a Personal Access Token to create or update workflow ... without workflow scope)`** The credential is valid. The push touched `.github/workflows/`. Add the `workflow` scope to the token, or the Workflows write permission to a fine-grained token, then push again. Nothing was written; the branch is unchanged.
6. **`Host key verification failed.`** The server's key does not match the saved one, or you declined the prompt. Compare the printed fingerprint against the operator's published one before doing anything else. If and only if it matches, remove the stale entry with `ssh-keygen -R github.com` and reconnect. Never delete the file wholesale to make a warning go away.
7. **`ssh: connect to host github.com port 22: Connection timed out`** Nothing replied. Port 22 is filtered on this network, or the path is broken. Test 443 as a control. Fix with an HTTPS remote or the `ssh.github.com` port 443 configuration in 42.7. Expect roughly 15 to 130 seconds before this message appears, depending on the system.
8. **`ssh: connect to host X port 22: Connection refused`** The opposite of the above and far friendlier. Something answered and said no. The host is reachable and nothing is listening on 22. Either sshd is not running, or it listens on another port. Measured in milliseconds, not seconds.
9. **`remote: Repository not found.`** followed by **`fatal: repository ... not found`**. Read this as "not found for you". Causes: a typo in the owner or repository name; the repository is private and your credential lacks access; you are authenticated as a different account than you think; a classic token without `repo` scope, which cannot see private repositories at all; an organization with SAML single sign-on where the token has not been authorized for that organization.
10. Why 404 rather than 403? Because returning 403 would confirm the repository exists. GitHub deliberately returns "not found" to anyone not permitted to know. We measured this directly in 42.8: an anonymous request for a nonexistent repository returned 401, not 404.
11. **`! [rejected] main -> main (non-fast-forward)`** or **`(fetch first)`**. Not an authentication problem at all. Your branch is behind the remote. `git fetch` then rebase or merge. If you push after a rebase, use `--force-with-lease`, never plain `--force`.
12. **`send-pack: unexpected disconnect while reading sideband packet`** The connection died while git was reading the server's response. This is the error the reader saw twice during a network outage.
13. It deserves a paragraph of its own because it is the most dangerous message in this list, and for a reason that is not obvious.
14. It tells you the response was lost. It tells you nothing about whether the write landed. The server may have applied your push completely and then failed to tell you.
15. So never assume either outcome. Re-query the actual state: `git ls-remote origin refs/heads/main` and compare the hash against your local commit.
16. Do not trust `origin/main` in your local repository either. That is a remote-tracking reference, which is a cached copy of what the remote said last time you successfully spoke to it. During an outage it is stale by definition.
17. Two more you will meet:
18. **`Too many authentication failures`** The agent offered more keys than the server's `MaxAuthTries` allows before you got to the right one. Fix with `IdentitiesOnly yes` and an explicit `IdentityFile` for that host.
19. **`sign_and_send_pubkey: signing failed: agent refused operation`** The agent has the public key but cannot use the private key. Usually a key added from a file that has since changed, or a hardware key that needs a touch. Run `ssh-add -D` then re-add.

20. A general ordering rule that resolves most confusion: read the output from the top, find the first line that names a cause, and ignore the summary at the bottom. `error: failed to push some refs` is never the reason.

42.13.6 WORDS - remember these

`remote:` prefix — a line that came from the server — output relayed from the far end, proving the connection and authentication both worked.

Sideband — the side channel carrying server messages — the multiplexed stream in the pack protocol used for progress and error text.

Remote-tracking reference — your cached note of what the server said — `refs/remotes/origin/main`, updated only on a successful fetch or push.

`--force-with-lease` — force push, but safely — a push that refuses if the remote moved since your last fetch, unlike plain `--force`.

`ls-remote` — ask the server what it actually has — a command querying live references without touching your working tree.

SAML SSO authorization — an extra approval step for org access — an organization requirement that a token be separately authorized before it can reach that organization's repositories.

42.98 Common wrong ideas

1. Wrong: SSH and HTTPS reach different copies of the repository, so switching loses work. Right: it is one repository. Only the road changes. Switching is one `git remote set-url` and nothing is lost.
2. Wrong: the private key is sent to the server, so publishing the public key is risky too. Right: the private key never moves. The server sends data, your machine signs it, and only the signature crosses the network. The public key is designed to be pasted into web forms on purpose.
3. Wrong: `Permission denied (publickey)` means the key is wrong. Right: it also appears when `~/.ssh` or `authorized_keys` has loose permissions, when the agent is empty, and when too many wrong keys were offered first. Only the server log distinguishes them.
4. Wrong: the host key warning is a nuisance, so delete `known_hosts`, and anyway a change always means an attack. Right: both halves are wrong. It is the one warning that detects an interceptor, so compare the fingerprint against a published one and remove only the stale entry. And GitHub legitimately rotated its RSA host key at about 05:00 UTC on 24 March 2023.
5. Wrong: port 22 being blocked means GitHub is down, and a blocked port fails immediately. Right: on the machine used for this chapter, port 22 timed out after 30.057 seconds while port 443 connected in 0.021 seconds to the same host. A DROP sends nothing back, so your machine retries with doubling gaps and hangs. Only a REJECT fails fast, with `Connection refused`.
6. Wrong: a token is just a password with extra steps. Right: a token can be scoped to specific actions, expire on a date, and be revoked alone. A password can do none of those things, which is why GitHub stopped accepting passwords for git on 13 August 2021.
7. Wrong: the `repo` scope includes everything, so a push cannot be refused for permissions. Right: `repo` does not include `workflow`. A push touching `.github/workflows/` is refused without it, even though authentication succeeded and every object transferred.
8. Wrong: `remote rejected` means the network dropped the push. Right: the word `remote:` proves you reached the server and it answered in sentences. That is a policy decision, not a transport failure.
9. Wrong: `Repository not found` means the repository does not exist. Right: it usually means your credential may not see it. GitHub returns "not found" rather than "forbidden" so it does not confirm that a private repository exists.

10. Wrong: changing the SSH port secures a server, and agent forwarding is just a convenience toggle. Right: a different port only removes automated scanning noise, since a real attacker scans all ports. And root on the machine you forwarded to can use your agent silently, against every server your keys open, for as long as you stay connected. Prefer `ProxyJump`.

42.99 Chapter summary in 20 lines

1. Git borrows two transports: SSH on TCP 22 and HTTPS on TCP 443. The repository is identical either way; only the road and the credential differ.
2. Over SSH, git logs in as the user `git` and runs one command remotely, `git-upload-pack` or `git-receive-pack`. Over HTTPS it makes one GET and one POST.
3. SSH was written by Tatu Ylonen at Helsinki University of Technology in 1995 after a password-sniffing attack, replacing telnet, rlogin and rsh, which sent passwords in clear text.
4. SSH-2 replaced the structurally flawed SSH-1 and is specified in RFC 4251 to RFC 4254. OpenSSH forked from Bjorn Gronvall's OSSH on 26 September 1999 and shipped with OpenBSD 2.6 on 1 December 1999.
5. SSH-2 has three layers: transport for encryption and host key verification, user authentication, and connection for multiplexed channels.
6. A key pair splits into a private key that signs and a public key that only verifies. Publishing the public half is safe; the private half never leaves your machine.
7. Authentication is challenge-response: the client signs a block containing the session identifier, so an old signature cannot be replayed and a man-in-the-middle cannot forward yours.
8. Ed25519 is the current recommendation: deterministic signatures, 32-byte public keys, published parameter rationale, RFC 8709. RSA 2048 is the floor and 4096 is common. DSA is removed everywhere.
9. `authorized_keys` is a plain text file of public keys. Wrong permissions make the server ignore it and return `Permission denied (publickey)` with no explanation to the client.
10. Options in front of a key restrict it: `command=`, `from=`, `no-port-forwarding`, and `restrict` for everything at once. Deploy keys are the per-repository variant.
11. `known_hosts` is the other half nobody reads. The server proves itself first, and the first connection is trust on first use, which is the model's honest weakness.
12. GitHub rotated its RSA host key at about 05:00 UTC on 24 March 2023 after the private key was briefly exposed publicly. That is what a legitimate change looks like.
13. The agent holds unlocked keys and signs on request, so the key never leaves its memory. Agent forwarding lets root on the remote host use your keys and should be replaced by `ProxyJump`.
14. Port 22 is commonly filtered outbound because it is heavily scanned and useful for exfiltration, while 443 must stay open or the web breaks. A DROP hangs for tens of seconds; a REJECT refuses in milliseconds.
15. The two fixes are switching the remote to HTTPS, or using GitHub's SSH endpoint `ssh.github.com` on port 443 with a four-line `~/.ssh/config` block.
16. GitHub removed password authentication for git operations on 13 August 2021. Credential helpers now store tokens, in the OS keychain if you are careful and in plain text if you use `store`.
17. A token can be scoped, can expire and can be revoked alone. Its prefix, `ghp_` or `github_pat_`, exists so that secret scanning can find it, and push protection blocks it before it ever leaks.
18. A scope is a named permission checked at the moment of the operation. Pushing a file under `.github/workflows/` needs `workflow`, because changing CI changes what runs on GitHub's runners with the repository's secrets.
19. Distinguish the three "push failed" cases by evidence: silence and a long wait is network, a credential prompt or `Invalid username` is authentication, and `! [remote rejected]` with a sentence is authorization.
20. Four identities with four blast radii: SSH keys, personal access tokens, OAuth grants and GitHub Apps. CI should hold no long-lived secret at all and use short-lived OIDC-federated credentials instead.

43

CHAPTER

KB

Cloud Computing, the Bezos Mandate, AWS and Azure

43.0 What this chapter gives you

1. You will be able to say what "the cloud" actually is in one plain sentence, without any mystery, and say exactly what it changed and what it did not.
2. You will be able to describe a real datacentre from the outside wall to the individual server, including power, cooling and what a rack unit is.
3. You will be able to explain regions, availability zones and edge locations precisely, and name the real regions the big three run inside India.
4. You will be able to explain what a hypervisor does, why virtual machines became cheap, and how special hardware removed the last of the overhead.
5. You will be able to state clearly that a container is a process, not a small virtual machine, and name the exact Linux features that make it work.
6. You will be able to read a Dockerfile, a Kubernetes manifest and a Terraform file line by line, and say what each line causes to happen.
7. You will be able to quote the 2002 Bezos API mandate, explain why it forced Amazon to build a platform, and say honestly what it cost.
8. You will be able to tell the true story of how AWS started, and correct the popular myth about spare retail capacity with evidence.
9. You will be able to name the eighteen AWS services you meet in real work, say what problem each solves, and map each to Azure and Google Cloud.
10. You will be able to read a cloud bill, predict where it will explode, and argue both sides of the buy-versus-rent question with real numbers.

43.1 What the cloud actually is

43.1.1 PLAIN – in simple words

1. The cloud is somebody else's computers, sitting in somebody else's building.
2. You rent them by the hour or by the second, instead of buying them.
3. You reach them over the internet, and you never touch them physically.
4. That is the whole idea. There is nothing floating in the sky.
5. The word "cloud" came from old network drawings, where people drew a fuzzy cloud shape to mean "the internet, and we are not going to draw the details".
6. So "in the cloud" originally just meant "over there, past our own wires".
7. Four things genuinely changed when this became a business you could buy.
8. First, **elasticity**: you can ask for 200 machines at 9am and give them back at 11am, and pay only for those two hours.

9. Second, **pay per use**: no big purchase up front, no machine sitting idle.
10. Third, **self-service**: you get the machines by making a request from a program, in seconds, with nobody's approval and no phone call.
11. Fourth, somebody else deals with the building, the power, the cooling, the broken disks and the security guard at the door.
12. Here is what did not change. These are still real physical computers.
13. They still overheat, lose power, lose disks and lose network cables.
14. Your program can still be slow, still be wrong, and still fall over.
15. The cloud moved the pain. It did not delete the pain.

43.1.2 PLAIN - a picture in your head

1. Think about how you get electricity in your home.
2. You do not own a generator. You do not buy fuel or service a machine.
3. You plug in, you use what you need, and once a month a meter reading turns into a bill.
4. If you buy an air conditioner, you do not have to also buy a bigger generator. You just use more, and the bill goes up.
5. If you go away for a month, you use almost nothing, and you pay almost nothing.
6. Somebody else runs enormous power stations so that you can avoid running a small one badly.
7. Renting computers works the same way. Amazon, Microsoft and Google run the enormous machine rooms so that you can avoid running a small one badly.
8. This comparison is not new. The writer Nicholas Carr made it the centre of his 2008 book The Big Switch, comparing cloud computing with the way factories stopped running their own power plants a century earlier.

Where this comparison breaks: electricity is one product. One volt is like any other volt, and you can change supplier without rewiring your house. Cloud services are all different from each other, and moving from one supplier to another means rewriting real code. Also, electricity is metered on one dimension. A cloud bill meters compute time, storage, requests, data movement, addresses and dozens of other things at once, and the surprise is almost always in a dimension you were not watching.

43.1.3 PLAIN - a worked example

1. You are two students launching a website for a college festival.
2. Old way: you buy one server. Say it costs 90,000 rupees, plus a rack space rental, plus a person to look after it.
3. You must guess the busiest moment months in advance and buy for that.
4. On the festival night 40,000 people arrive at once. Your one server dies.
5. For the other 360 days of the year, the server is 3 per cent busy, and you have paid for all of it anyway.
6. Cloud way: you rent two small machines for the quiet months. On festival night you rent forty machines for six hours, then give them back.

Approach	Quiet month	Festival night
Own one server	Paid in full	Site falls over
Rent, fixed size	Pay 2 machines	Site falls over
Rent, elastic	Pay 2 machines	Pay 40 for 6 hours

7. Real prices, from the AWS public price list for the North Virginia region in August 2026: a small `t3.medium` machine costs 0.0416 US dollars per hour.
8. Forty of them for six hours is 40 x 6 x 0.0416 = 9.98 US dollars.
9. That is the actual thing that changed. Ten dollars, decided at 7pm, with no purchase order and no meeting.

43.1.4 PLAIN - what is really happening inside

1. You send a small message over the internet to the provider's control system. It says, in effect, "give me one machine of this size, in this place".
2. That message is a normal web request, signed with your secret key so the provider knows it is really you.
3. A piece of software called the **control plane** takes the request. Its job is to decide and to record, not to run your program.
4. It finds a physical server in the right building that has room, and picks it.
5. On that physical server, a program called a **hypervisor** carves out a slice: some processor time, some memory, some disk, one network address.
6. It boots an operating system into that slice from a saved disk image.
7. Within about 30 to 60 seconds, that slice answers to a network address, and you can log in as if it were a computer on your desk.
8. Everything after that is ordinary computing. The slice does not know or care that it is a slice.
9. A separate meter records that this slice existed, second by second, and that record eventually becomes a line on your bill.
10. When you say "delete it", the hypervisor throws the slice away and the physical server is offered to the next customer.

43.1.5 TECHNICAL - the engineer's version

1. The working definition of cloud computing is not marketing. It is a standard document: NIST Special Publication 800-145, published September 2011 by the United States National Institute of Standards and Technology.
2. It names five essential characteristics. A service is only cloud computing if it has all five.
3. On-demand self-service: the consumer provisions capacity unilaterally, with no human interaction with the provider.
4. Broad network access: capabilities are available over the network through standard mechanisms.
5. Resource pooling: physical and virtual resources are pooled and assigned to many consumers, with location independence at some level of abstraction.
6. Rapid elasticity: capabilities can be scaled outward and inward, in some cases automatically, and appear unlimited to the consumer.
7. Measured service: use is metered and reported, giving transparency to both sides.
8. The same document names three service models, IaaS, PaaS and SaaS, and four deployment models: private, community, public and hybrid.
9. That test is genuinely useful. A rented server with a two-week ticket queue fails point three and point six, and is therefore hosting, not cloud.
10. Scale of the industry, from published company filings. Amazon reported AWS net sales of 128,725 million US dollars for the full year 2025, with segment operating income of 45,606 million.
11. In the quarter ending June 2026 AWS reported 42.2 billion US dollars, up 37 per cent year on year, an annualized run rate near 169 billion.
12. Microsoft first disclosed Azure's own revenue for its fiscal year 2025, at 75 billion US dollars, and reported that Azure passed 100 billion for fiscal 2026, growing 41 per cent.
13. Alphabet reported Google Cloud revenue of 17.664 billion US dollars for the quarter ending December 2025, up 48 per cent, and said the business ended 2025 at an annual run rate above 70 billion.

Provider	Latest disclosed figure	Period
AWS	128.7 bn USD	Full year 2025
Azure	Over 100 bn USD	Fiscal year 2026
Google Cloud	17.7 bn USD	Q4 2025 quarter

14. Commands that make the abstraction visible: `aws ec2 describe-instances`, `az vm list`, `gcloud compute instances list`. Each is a signed web request to a control plane, nothing more.

43.1.6 WORDS - remember these

1. Cloud — renting computers you reach over the internet — on-demand, self-service, metered, pooled and elastic computing per NIST SP 800-145.
2. Elasticity — getting more and giving it back quickly — automatic or manual scaling of provisioned capacity in both directions.
3. Control plane — the part that decides and records — the API and orchestration layer that provisions and tracks resources.
4. Data plane — the part that does the actual work — the running instances, packets and stored bytes themselves.
5. Provisioning — asking for and receiving a resource — the allocation of capacity to a tenant in response to an API call.
6. Tenant — one paying customer sharing the hardware — an isolated account whose workloads share pooled physical resources.

43.2 Inside a real datacentre

43.2.1 PLAIN - in simple words

1. A datacentre is a warehouse full of computers, built so they never stop.
2. Inside, computers are not on desks. They are bolted into tall metal frames called **racks**, like books on a shelf but lying flat.
3. Each computer is a flat metal tray about as wide as a shelf and as thick as a few textbooks. Rows of them fill each rack from floor to head height.
4. Every rack has a small switch at the top that all its computers plug into.
5. Those switches plug into bigger switches, which plug into the internet.
6. Power comes in from the electricity grid, on two separate paths, so that one failure does not stop the room.
7. Between the grid and the computers sit big batteries, which cover the few seconds of a cut, and diesel generators, which cover the hours.
8. All that electricity turns into heat, so a large part of the building is machinery for taking heat away.
9. Cold air is pushed up through the floor into one aisle, sucked through the computers, and comes out hot into the next aisle.
10. Getting in is hard on purpose: fences, cameras, guards, card and fingerprint doors, and a locked cage around your own rack.
11. These buildings are enormous. A big one holds tens of thousands of computers and uses as much electricity as a small city.

43.2.2 PLAIN - a picture in your head

1. Think of a very large library.
2. The racks are the shelves, standing in long parallel rows.
3. Each server is one book lying flat on the shelf, and the shelf spacing is fixed so books of standard thickness always fit.
4. Every row of shelves has a librarian at the end of it, the top-of-rack switch, who handles all messages in and out of that row.
5. The librarians all report to a small group of head librarians, and any librarian can reach any other through any head librarian.
6. The reading rooms alternate: one cold room, one warm room, one cold, one warm, because readers give off heat and you keep the cold air away from the warm exhaust.

7. The building has its own power station in the yard, in case the city cuts supply, and a battery room that keeps the lights on while it starts.

Where this comparison breaks: a library's books are the point of the library. Here, the servers are interchangeable and expected to die. Roughly two to five per cent of servers fail in a year, and staff replace them without anybody noticing. Also, a library is quiet. A datacentre hall is loud enough that long-stay staff wear ear protection, because thousands of fans are running.

43.2.3 PLAIN - a worked example

1. Take one standard rack. The width of the mounting rails is 19 inches, which is 482.6 millimetres. This is the EIA-310 standard, and it dates from telephone equipment in the 1920s.
2. Height is measured in **rack units**, written U. One U is 1.75 inches, which is 44.45 millimetres, exactly.
3. A common rack is 42U tall, about 1.87 metres of usable height.
4. A basic web server is 1U thick. So 42 of them fit, minus space for switches and cable management. In practice you plan for about 38 servers plus 2 switches.
5. Now the power. Suppose each server draws 350 watts on average.
6. 38 x 350 W = 13,300 W, which is 13.3 kilowatts for one rack.
7. An older datacentre was designed for 5 to 10 kilowatts per rack, so that rack would not fit its power budget. You would fill the rack only half full.
8. A modern rack of graphics processors for artificial intelligence work is a different animal. An NVIDIA GB200 NVL72 rack draws up to about 120 kilowatts, which is roughly what forty Indian homes use at peak.
9. Air cannot carry that much heat away. Those racks are water-cooled, with cold liquid piped directly onto the chips.

Rack type	Power per rack	Cooling
Legacy enterprise	5 to 10 kW	Air
Modern cloud	10 to 20 kW	Air, contained
AI training rack	Up to 120 kW	Direct liquid

43.2.4 PLAIN - what is really happening inside

1. Grid electricity arrives at high voltage and is stepped down by transformers outside the building.
2. It passes through switchgear that can move the whole load from one source to another in a fraction of a second.
3. It then reaches the uninterruptible power supply, the UPS. This is a bank of batteries or a spinning flywheel.
4. The UPS is not there to run the building for hours. It is there to hold the load for the 10 to 30 seconds a diesel generator needs to start and take over.
5. Generators keep enough fuel on site for a day or more, with contracts for refuelling after that.
6. From the UPS, power goes to rack-level power strips, called PDUs. Servers have two power supplies, plugged into two independent strips, fed from two independent paths.
7. Cooling runs in a loop. Computer room air handlers blow cold air under a raised floor or through overhead ducts into the cold aisle.
8. Server fans pull that air front to back, over the hot parts, and blow it into the hot aisle behind.
9. Physical barriers, doors and roofs, seal the aisles so cold and hot air never mix. Mixing is the single biggest waste in a machine room.
10. The hot air returns to the coolers, gives its heat to a water loop, and the water carries the heat to chillers or to outdoor cooling towers.

11. For very hot racks, air is skipped entirely. A cold plate sits directly on the chip and water takes the heat out of the rack.
12. Meanwhile the network: each server has one or two cables to the switch at the top of its own rack, and those switches connect upwards to a layer of larger switches, so that any server can reach any other in a small, fixed number of hops.

43.2.5 TECHNICAL - the engineer's version

1. Rack mechanics follow EIA-310-D. Rail spacing 19 inches, 1U equals 44.45 mm, common heights 42U, 45U, 48U and 52U, common depths 1000 to 1200 mm.
2. Rackmount servers are 1U, 2U or 4U. A 1U dual-socket server today typically holds 2 CPUs of 32 to 128 cores each, 12 to 32 memory slots and a handful of NVMe drives.
3. Blade servers put many thinner server modules into a shared chassis, sharing power supplies, fans and network fabric. They win on density and cabling and lose on vendor lock-in. Hyperscale operators mostly use custom rackmount designs instead, following the Open Compute Project, which Facebook founded in 2011.
4. Network topology is **spine-leaf**, also called a folded Clos network after Charles Clos's 1953 telephone switching paper. Every leaf connects to every spine, and no leaf connects to another leaf.

```
        [spine 1]   [spine 2]   [spine 3]   [spine 4]
           |  \   /   |   \  /    |   \  /     |
           |   \ /    |    \/     |    \/      |
           |   / \    |    /\     |    /\      |
        [leaf A]   [leaf B]   [leaf C]   [leaf D]
          | | |      | | |      | | |      | | |
        servers    servers    servers    servers

  Any server to any server: leaf -> spine -> leaf. Two hops.
  Add a spine to add bandwidth. Add a leaf to add racks.
```

5. Leaf switches are the top-of-rack switches, typically 48 ports of 25 or 100 gigabits per second down to servers, and 4 to 8 uplinks of 100 to 400 gigabits per second up to the spine.
6. The ratio of downlink to uplink bandwidth is the **oversubscription ratio**. 3:1 is common in general fleets. Storage and AI clusters are built at 1:1, meaning non-blocking.
7. Power redundancy is described as N, N+1 or 2N. The Uptime Institute Tier classification runs Tier I to Tier IV, where Tier III allows concurrent maintenance and Tier IV is fault tolerant.
8. ASHRAE Thermal Guidelines class A1 recommends an inlet temperature of 18 to 27 degrees Celsius. Operators run near the top of that band deliberately, because warmer air needs less chilling.
9. **Power usage effectiveness**, PUE, is total facility power divided by IT equipment power. A PUE of 2.0 means one watt of overhead for every watt of computing. The theoretical floor is 1.0.

Operator or group	PUE	Source year
Global average	1.54	Uptime 2025 survey
Hyperscale range	1.10 to 1.15	Uptime 2025 survey
Google fleet, TTM	1.09	Reported Q1 2026
Colocation and enterprise	1.58 to 1.80	Uptime 2025 survey

10. Google's published history shows fleet PUE falling from about 1.23 in 2008 to 1.09 today. The global average has been stuck near 1.54 since 2019, which is an honest and unflattering industry fact.
11. Physical security is audited, not asserted. The relevant reports are SOC 2 Type II, ISO/IEC 27001 and, for cardholder data, PCI DSS. Controls include perimeter fencing, vehicle barriers, two-factor mantrap entry, biometric readers, per-cage access lists and asset destruction on decommission.

12. Scale and cost, from 2026 construction cost surveys: roughly 11.3 million US dollars per megawatt of IT capacity for a standard facility, and 20 million or more per megawatt for an AI-optimized build.
13. On that basis a 500 megawatt campus costs about 5.5 to 6 billion US dollars standard, or over 10 billion for a dense AI build. Electrical systems alone are 40 to 50 per cent of construction cost.
14. Microsoft stated that it deployed about 2 gigawatts of new datacentre capacity during its fiscal year 2025. That single year of one company is comparable to the entire generating capacity of a large power station.

43.2.6 WORDS - remember these

1. Rack unit (U) — one shelf slot — 44.45 mm of vertical mounting height under EIA-310-D.
2. Top-of-rack switch — the small switch every server in a rack plugs into — the leaf layer of a Clos fabric, typically 48 x 25G with 100G uplinks.
3. Spine-leaf — a wiring plan where any machine is two hops from any other — a folded Clos topology with full leaf-to-spine meshing.
4. UPS — the battery that covers a power cut — an uninterruptible power supply sized to bridge to generator start, typically 10 to 30 seconds.
5. Hot and cold aisle — keeping cold air and hot exhaust apart — containment of supply and return airflow to prevent recirculation.
6. PUE — how much extra power the building wastes — total facility power divided by IT load, global average 1.54, hyperscale 1.10 to 1.15.
7. CRAC and CRAH — the room's air conditioners — computer room air conditioner with its own compressor, versus air handler fed by chilled water.

43.3 Regions, availability zones and edge locations

43.3.1 PLAIN - in simple words

1. A **region** is a place on the map where a provider has built capacity. Mumbai is a region. Frankfurt is a region. Ohio is a region.
2. Regions are far apart, and they are run as almost separate businesses. Prices differ, and some services exist in one region and not another.
3. Inside a region there are **availability zones**. A zone is one or more buildings with its own power, its own cooling and its own network links.
4. Two zones in one region are close enough that talking between them is fast, but far enough apart that one flood, fire or substation failure cannot take out both.
5. That is the whole point. A zone is a **failure domain**: a boundary that damage is not supposed to cross.
6. So you never put your only copy of anything in one zone. You run at least two copies, in two zones, and something in front that sends traffic to whichever is alive.
7. An **edge location** is a third thing and much smaller. It is a small set of machines in many cities that hold copies of files and terminate connections close to users.
8. Edge locations make things feel fast. They do not run your application. Do not confuse them with regions.
9. Rule of thumb: zones protect you from a building. Regions protect you from a city or a country. Edge locations protect you from distance.

43.3.2 PLAIN – a picture in your head

1. Think of a bank with branches.
2. A region is a city where the bank operates.
3. Availability zones are three separate branch buildings in that city, on different streets, on different electrical supplies.
4. If one branch burns down, your money is still in the other two, because the bank writes every transaction to all three before confirming it.
5. An edge location is an ATM. There are hundreds of them, in many towns, and they are fast to reach.
6. But an ATM cannot open an account or approve a loan. It holds cash and handles simple requests. For anything real it talks back to a branch.
7. Countries also have rules: some money must stay inside the country. That is data residency.

Where this comparison breaks: bank branches in a city are genuinely independent businesses that can run alone. Cloud availability zones share a regional control plane, and some regional services have components that live in only one zone. In the AWS event of 20 October 2025, one subsystem in one region failed and the effect was felt across the whole region and by customers worldwide, even though the individual zones had power and cooling the entire time.

43.3.3 PLAIN – a worked example

1. You run a shop website for Indian customers. You pick the AWS Mumbai region, whose code name is `ap-south-1`.
2. That region has three availability zones, named `ap-south-1a`, `ap-south-1b` and `ap-south-1c`.
3. You place two web servers in zone a and two in zone b, with a load balancer in front that spreads traffic across both.
4. You place your database as a primary in zone a and a standby copy in zone b, with automatic promotion if the primary is lost.
5. Now zone a loses power. The load balancer stops sending traffic to the two dead servers within about 30 seconds. The database standby is promoted in one to two minutes. Customers see a short blip.
6. Cost of that safety: you run four servers where two would do, and you pay for traffic that crosses between zones, at 0.01 US dollars per gigabyte in each direction.
7. Now suppose the whole Mumbai region fails. Nothing you have deployed helps. Surviving that needs a second region, which needs a second copy of your data kept up to date, which is a much larger project.

43.3.4 PLAIN – what is really happening inside

1. Zone names are shuffled per account on purpose. Your `ap-south-1a` and my `ap-south-1a` are usually different physical zones, so that customers do not all crowd into the alphabetically first one.
2. The stable identifier is a zone ID such as `aps1-az1`, which is the same for everyone. That is what you compare when you need to know two accounts are truly in the same building.
3. Between zones, the provider runs its own dedicated fibre, usually many redundant paths. Round trip time is typically well under two milliseconds.
4. Between regions, traffic crosses the provider's private long-distance backbone. Mumbai to Singapore is roughly 60 to 70 milliseconds round trip. Mumbai to North Virginia is roughly 190 to 220 milliseconds.
5. Those numbers come from the speed of light in glass and cannot be improved by any amount of money. Light travels about 200,000 kilometres per second in fibre, so 1,000 km of fibre costs about 5 milliseconds one way, before any switching delay.
6. A region failure, in practice, is rarely the building burning. It is far more often a software or configuration fault in a service that everything else in the region depends on, such as identity, DNS or the metadata store.
7. When that happens, machines that are already running usually keep running, but you cannot start new ones, cannot log in, and cannot change anything.

43.3.5 TECHNICAL - the engineer's version

1. Current published footprints, checked in August 2026. AWS states 123 availability zones across 39 geographic regions, with plans announced for the Kingdom of Saudi Arabia and Chile. Microsoft states over 70 Azure regions. Google states 43 regions and 130 zones.
2. AWS defines an availability zone as one or more discrete datacentres with redundant power, networking and connectivity, physically separated by a meaningful distance, up to 100 km apart, with single-digit millisecond latency between them.
3. Azure calls the equivalent an availability zone too, and pairs most regions into region pairs for platform-managed geo-replication. You cannot choose your pair.
4. Google Cloud calls them zones, and a zone is a deployment area within a region, isolated for failure purposes.
5. The India footprint, with names as the providers publish them:

Provider	Region name	Live from
AWS	Mumbai, ap-south-1	June 2016
AWS	Hyderabad, ap-south-2	21 Nov 2022
Azure	Central, South, West India	Sept 2015
Azure	India South Central	6 Aug 2026
Google	Mumbai, asia-south1	Nov 2017
Google	Delhi NCR, asia-south2	15 July 2021

6. Azure also offers Jio India West and Jio India Central, operated in partnership with Reliance Jio. Azure's three 2015 regions are in Pune, Chennai and Mumbai; the 2026 India South Central region is in Hyderabad and opened with three availability zones.
7. AWS announced an investment of 4.4 billion US dollars in India by 2030 at the Hyderabad launch. Microsoft's stated India commitment reached 20.5 billion US dollars by 2026, combining announcements from January 2025 and December 2025.
8. Data residency is a legal question, not a technical one. In India the relevant instruments are the Digital Personal Data Protection Act, 2023, and the Reserve Bank of India's April 2018 directive requiring payment system data to be stored in India.
9. In the European Union it is the General Data Protection Regulation of 2016, applied from May 2018, plus sector rules and sovereign cloud offerings.
10. Choosing a region therefore has three inputs: latency to your users, price, and law. They frequently disagree.
11. The honest note on multi-region. Marketing implies you tick a box. The reality is that you must solve write conflicts, replication lag, failover orchestration, DNS behaviour under partial failure, cost of duplicated capacity, and testing.
12. Almost nobody tests region failover under load. A failover path that has never been exercised is a hypothesis, not a capability.
13. Real example to anchor this. On 20 October 2025 AWS suffered a major disruption in `us-east-1`, caused by a latent race condition in the automated DNS management of DynamoDB. The trigger occurred around 06:49 UTC, DNS was restored by 09:25 UTC, and full recovery of dependent systems such as EC2 instance launches continued until about 20:50 UTC the same day, with some Redshift work finishing the next morning.
14. Many companies that believed they were multi-region discovered that their identity provider, their build system or their DNS was single-region.
15. Commands to inspect this: `aws ec2 describe-availability-zones --region ap-south-1` shows both the name and the stable zone ID.

43.3.6 WORDS - remember these

1. Region — a city-sized place where the provider has capacity — an independent deployment area with its own service catalogue, pricing and endpoints.
2. Availability zone — one building group with its own power — an isolated failure domain within a region, single-digit millisecond latency to peers.
3. Failure domain — the boundary damage should not cross — the blast radius of a single correlated fault.
4. Edge location — a small cache near users — a point of presence for content delivery and connection termination, not general compute.
5. Data residency — the rule that data must stay in a country — a legal obligation on storage location, distinct from encryption or access control.
6. Region pair — Azure's fixed partner region — a Microsoft-assigned pairing used for platform geo-replication and staged updates.

43.4 Virtualization, the technology that made cloud possible

43.4.1 PLAIN - in simple words

1. One physical computer used to run one operating system, and that was that.
2. Most of the time it was mostly idle, waiting for something to do.
3. Virtualization is a trick that lets one physical computer pretend to be many separate computers at once.
4. The program that does the pretending is called a **hypervisor**.
5. It sits between the real hardware and the pretend computers, and it hands out slices of processor, memory, disk and network.
6. Each pretend computer is called a **virtual machine**. It boots its own operating system and believes it is alone.
7. It is not alone. Ten or forty of its neighbours are on the same metal, completely unaware of each other.
8. This is what made renting computers by the hour possible. Without it, giving a customer a machine would mean physically plugging one in.
9. With it, giving a customer a machine means writing a few numbers into a table and starting a process.
10. It also lets the provider move a running customer off a server that needs repair, without switching that customer off. That is called live migration.

43.4.2 PLAIN - a picture in your head

1. Think of a large old house converted into flats.
2. The house is the physical server. The flats are the virtual machines.
3. Each family has its own front door, own kitchen, own furniture, and never sees the others.
4. There is one water main, one electricity supply and one roof, shared. A caretaker, the hypervisor, splits them fairly and stops one family from flooding another.
5. If one family plays loud music, the others may hear it a little. That is resource contention, the noisy neighbour problem.
6. If the caretaker needs to fix the roof over one flat, a very good caretaker can move the whole family, mid-dinner, to an identical flat next door, so fast they barely notice. That is live migration.

Where this comparison breaks: a family in a flat knows it is in a flat. A well built virtual machine genuinely cannot tell, by any ordinary means, that it is virtual. That indistinguishability is a formal design goal, stated by Gerald Popek and Robert Goldberg in their 1974 paper, and it is stronger than any housing analogy suggests. Also, the caretaker in this story has absolute power: the hypervisor can read every byte of every flat, which is why the trust model of the provider matters so much.

43.4.3 PLAIN - a worked example

1. Take a physical server with 2 processors of 64 cores each, so 128 physical cores, plus 1,024 gigabytes of memory.
2. With hyper-threading on, the operating system sees 256 logical processors.
3. You want to sell virtual machines with 4 virtual processors and 16 gigabytes of memory each.
4. On memory alone, 1,024 divided by 16 gives 64 virtual machines, minus about 5 per cent kept for the hypervisor itself. Call it 60.
5. On processors, 60 virtual machines x 4 virtual processors = 240 virtual processors against 256 logical ones. That is roughly 1:1, no oversubscription.
6. If instead you sell to customers whose machines are usually idle, you might place 120 virtual machines, giving 480 virtual processors on 256 logical ones, an oversubscription ratio of nearly 2:1.
7. That is a business decision, not a technical one, and it is exactly why cheap virtual servers feel inconsistent and expensive ones do not.

Setting	Ratio	Effect on tenant
No oversubscription	1:1	Steady, predictable
Light, 2:1	2:1	Usually fine, sometimes slow
Aggressive, 8:1	8:1	Frequent stealing of time

8. You can measure this from inside a Linux guest. Run `top` and look at the `st` column, short for steal time. It is the percentage of time your virtual processor was ready to run but the hypervisor gave the physical core to somebody else. Anything consistently above 5 is a crowded host.

43.4.4 PLAIN - what is really happening inside

1. A processor has at least two privilege levels: a privileged one for the operating system and an unprivileged one for applications.
2. Normally the operating system sits in the privileged level and controls the hardware directly.
3. Under virtualization, the hypervisor takes the privileged level, and the guest operating system is pushed down a level, even though it still thinks it is privileged.
4. When the guest executes an instruction that only a truly privileged program may run, the processor stops and hands control to the hypervisor. This is called a **trap**.
5. The hypervisor looks at what the guest was trying to do, does something safe and equivalent, and resumes the guest.
6. Memory is handled with two levels of translation. The guest maps its own pages as usual; the hardware then maps those again to real physical pages, using a second table the hypervisor controls.
7. Disks are usually just large files on the host, or slices of a network storage system. A write inside the guest becomes a write to that file.
8. Network cards are emulated or, better, partly real: modern cards can present many independent copies of themselves, and the hypervisor gives one copy directly to each guest.
9. Live migration works by copying memory pages to the destination host while the guest keeps running, repeatedly re-copying pages that changed, until the set of changed pages is small enough to copy in a brief pause.
10. That final pause is typically well under a second, and the guest's network connections survive because the address moves with it.

43.4.5 TECHNICAL - the engineer's version

1. Gerald Popek and Robert Goldberg, in Formal Requirements for Virtualizable Third Generation Architectures, Communications of the ACM, July 1974, gave three requirements: equivalence, resource control and efficiency.

2. Their theorem: an architecture is classically virtualizable if the set of sensitive instructions is a subset of the privileged instructions.
3. x86 failed that test. John Robin and Cynthia Irvine identified 17 problematic instructions in a 2000 USENIX Security paper. Instructions such as `POPF` silently did the wrong thing instead of trapping.
4. Two workarounds followed. VMware, founded in 1998, shipped binary translation: rewrite the guest's privileged instruction stream on the fly.
5. Xen, from the University of Cambridge, presented at SOSP in 2003, shipped **paravirtualization**: modify the guest kernel so it calls the hypervisor deliberately, through hypercalls, instead of executing offending instructions. Faster, but needs a modified guest.
6. Hardware assistance ended the argument. Intel VT-x shipped in November 2005; AMD-V, codenamed Pacifica, shipped in May 2006. Both add a new guest mode, so an unmodified guest kernel can run at its expected privilege level and trap cleanly to the hypervisor.
7. Memory virtualization was still expensive until second level address translation arrived: AMD Nested Page Tables in Barcelona in 2007, Intel Extended Page Tables in Nehalem in 2008. Before that, hypervisors maintained shadow page tables in software.
8. Device passthrough needs an IOMMU: Intel VT-d and AMD-Vi. PCI-SIG's Single Root I/O Virtualization specification, published 2007, lets one physical card present many virtual functions, each assignable to one guest.

Hypervisor type	Runs on	Examples
Type 1, bare metal	The hardware	ESXi, Xen, Hyper-V, KVM
Type 2, hosted	An operating system	VirtualBox, Workstation

9. The type 1 versus type 2 split is a useful teaching device and a slightly leaky one. KVM, merged into Linux 2.6.20 in February 2007, turns the Linux kernel itself into a type 1 hypervisor while Linux remains a normal operating system. Experts differ on which box it belongs in.
10. From the host's point of view a KVM virtual machine is an ordinary process. You can see it in `ps`, one thread per virtual processor, its guest memory is an anonymous mapping, and you can kill it with `kill`.
11. Live migration in QEMU/KVM is iterative pre-copy: transfer all pages, then repeatedly transfer the pages dirtied during the previous pass, then stop the guest, transfer the remainder and the device state, and resume on the target. Typical stop time is tens to a few hundred milliseconds.
12. The **hypervisor tax** is the fraction of a machine consumed by virtualization rather than by customers. Werner Vogels has written that on older AWS designs this was around 30 per cent of the resources.
13. AWS attacked it with hardware. The path: C3 in 2013 offloaded network processing, C4 in 2014 offloaded storage, Amazon acquired the Israeli chip company Annapurna Labs in early 2015, and the full **Nitro System** shipped with C5 instances in 2017.
14. Nitro has three parts: Nitro Cards, which are dedicated hardware running VPC networking, EBS storage and instance storage; the Nitro Security Chip, which verifies firmware; and the Nitro Hypervisor, a thin KVM-based component that now mainly assigns memory and CPU.
15. Because almost nothing is left in software, AWS can also sell `.metal` instances, where a customer gets the whole physical server with no hypervisor at all, yet still gets normal EBS volumes and VPC networking.
16. Microsoft's equivalent story runs through programmable hardware. The Catapult project put FPGAs in Azure servers from 2015, Accelerated Networking used those FPGA-based SmartNICs to move virtual switching off the CPU, and Azure Boost, announced in 2023, offloads storage and networking to a dedicated card in the same spirit as Nitro.
17. Observation commands: `lscpu` shows the hypervisor vendor line inside a guest; `systemd-detect-virt` names the hypervisor; `dmidecode -s system-manufacturer` often reveals it; `virsh list` and `virsh domstats` inspect KVM guests from the host.

43.4.6 WORDS - remember these

1. Hypervisor — the program that splits one computer into many — a virtual machine monitor providing equivalence, resource control and efficiency.
2. Type 1 versus type 2 — running on bare metal versus on top of an existing operating system — native versus hosted VMM.
3. Paravirtualization — the guest knows it is a guest and cooperates — replacing sensitive instructions with explicit hypercalls, as in Xen.
4. VT-x and AMD-V — processor features that make the trick cheap — hardware virtualization extensions adding a distinct guest execution mode, 2005 and

2006.

5. Oversubscription — selling more slices than you physically have — allocating virtual resources beyond physical capacity, relying on idle time.
6. Steal time — time your machine was ready but not scheduled — the `st` counter in Linux CPU accounting, the direct measure of host contention.
7. Live migration — moving a running machine without switching it off — iterative pre-copy of memory and device state to another host.
8. Nitro — AWS's offload hardware — dedicated cards and a minimal hypervisor that remove virtualization overhead and enable bare-metal instances, from 2017.

43.5 Containers

43.5.1 PLAIN - in simple words

1. A virtual machine is a whole pretend computer, with its own operating system inside. That costs memory and takes tens of seconds to start.
2. Most of the time you did not want a computer. You wanted to run one program, with the exact files and libraries it expects.
3. A **container** gives you that. It is one ordinary program, running on the normal operating system, but wrapped so that it can only see its own files, its own processes and its own network.
4. Say this clearly, because almost every beginner gets it wrong: a container is not a small virtual machine. There is no second operating system inside it.
5. From the host's point of view it is just a process, visible in the process list, killable, schedulable, like any other.
6. The wrapping is done by features built into the Linux kernel. The container asks for a private view of the world, and the kernel gives it one.
7. Because there is nothing to boot, a container starts in tens of milliseconds instead of tens of seconds.
8. Because there is no second operating system, it uses megabytes of extra memory instead of hundreds of megabytes.
9. The other half of the idea is the **image**: a frozen, shareable copy of all the files the program needs, built once and run anywhere.
10. That is why containers solved the oldest complaint in software: "it works on my machine". Now the machine travels with the program.
11. The trade is isolation. Virtual machines are separated by hardware enforcement. Containers are separated by kernel bookkeeping, which is thinner, and a kernel bug can cross it.

43.5.2 PLAIN – a picture in your head

1. Imagine a big office building with one security system, one power supply and one set of stairs. That is the operating system kernel.
2. A virtual machine is like building a whole separate house in the car park: its own foundations, its own wiring, its own everything.
3. A container is like giving one team a locked room inside the existing building, plus a rule at reception that they may not ask about any other team, plus a cap on how much electricity and how many desks they may use.
4. The team genuinely cannot see the other teams. Ask them who else is in the building and they will answer, honestly, "nobody".
5. Setting up a room takes minutes. Building a house in the car park takes months.
6. But the room is inside a building that somebody else controls. If the building's fire alarm has a flaw, everybody in it is affected.

Where this comparison breaks: the locked room suggests a physical wall. There is no wall. There is only the receptionist consistently refusing to answer certain questions. If you find a way to make the receptionist answer, you are out. That is exactly what a container escape is, and it is why you do not run untrusted code in a plain container without further sandboxing.

43.5.3 PLAIN – a worked example

1. Here is a real Dockerfile for a small Python web service, with every line explained afterwards.

```
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
RUN useradd -m appuser && chown -R appuser /app
USER appuser
EXPOSE 8000
ENV PORT=8000
CMD ["gunicorn", "-b", "0.0.0.0:8000", "app:application"]
```

2. `FROM python:3.12-slim` starts from an existing image that already contains a minimal Linux userland and Python 3.12. You are not building from nothing.
3. `WORKDIR /app` creates and enters a directory. Later commands run there.
4. `COPY requirements.txt .` copies just the list of dependencies in first.
5. `RUN pip install ...` installs those dependencies. Because this is a separate layer, and because only the requirements file was copied so far, this expensive step is reused from cache whenever your source changes but your dependency list does not. This ordering is the single most valuable trick in the file.
6. `COPY . .` now copies your actual source code.
7. `RUN useradd ...` creates a non-root user and gives it ownership.
8. `USER appuser` means every later instruction, and the running container, operate as that user. Running as root inside a container is a common and bad default.
9. `EXPOSE 8000` is documentation plus metadata. It does not open a port by itself.
10. `ENV PORT=8000` sets an environment variable inside the image.
11. `CMD [...]` is the command run when the container starts. The list form avoids a shell, so signals reach your process directly.
12. The commands to build and run it:

```
docker build -t festival-site:1.4 .
docker run -d --name web -p 8080:8000 \
  --memory 512m --cpus 1.5 festival-site:1.4
docker ps
```

```
docker logs -f web
docker exec -it web /bin/sh
docker stop web && docker rm web
```

13. `-p 8080:8000` maps port 8080 on the host to 8000 in the container.
14. `--memory 512m --cpus 1.5` are the limits. Without them, one container can consume the whole machine.

43.5.4 PLAIN – what is really happening inside

1. When you run a container, the runtime asks the Linux kernel to start a process with a private view of six or seven kinds of thing. Each private view is called a **namespace**.
2. The PID namespace gives the process its own process numbering. The first process inside sees itself as PID 1 and cannot see the host's processes.
3. The network namespace gives it its own network interfaces, its own routing table and its own firewall rules. Usually a virtual cable is created, one end inside, one end on a bridge in the host.
4. The mount namespace gives it its own filesystem tree, so `/` inside is not `/` outside.
5. The user namespace lets an identity that is root inside map to an ordinary unprivileged identity outside.
6. The IPC namespace separates shared memory and message queues. The UTS namespace lets the container have its own hostname.
7. Separately, **cgroups** put ceilings on what the process may consume: so much memory, so much processor time, so much disk bandwidth. If it exceeds the memory ceiling, the kernel kills it.
8. The filesystem is assembled from stacked read-only layers plus one writable layer on top. Reads fall through the stack to the first layer that has the file. Writes go to the top layer only, copying the file up first.
9. Because the read-only layers are shared, ten containers from the same image store one copy of the common files, not ten.
10. The image itself is just those layers plus a small JSON document describing the command, environment, user and layer order.
11. A **registry** is a web service that stores and serves those layers and documents, addressed by content hash.

43.5.5 TECHNICAL – the engineer's version

1. The idea is old. `chroot` appeared in Version 7 Unix in 1979: change the apparent root directory of a process. It isolates the filesystem view and nothing else, and it was never a security boundary.
2. FreeBSD jails, by Poul-Henning Kamp, arrived in FreeBSD 4.0 in March 2000, adding process, network and user isolation to the chroot idea.
3. Solaris Zones shipped in Solaris 10 in 2005, with resource controls and a full virtualized environment per zone.
4. On Linux the pieces arrived separately. The mount namespace landed in kernel 2.4.19 in 2002. Process containers, developed at Google by Paul Menage and Rohit Seth, were renamed control groups and merged in kernel 2.6.24 in January

2008. The user namespace was completed in kernel 3.8 in 2013.

5. LXC, released in 2008, was the first tool to assemble those pieces into something usable. It required real Linux skill.
6. Docker changed the packaging, not the kernel. Solomon Hykes demonstrated it at PyCon in Santa Clara and it was open sourced in March 2013, out of a company then called dotCloud, renamed Docker Inc. later that year.
7. Docker's contribution was the image format, the layer cache, the Dockerfile and a public registry: a developer workflow, not an isolation mechanism. Early versions used LXC underneath, replaced by its own libcontainer in version 0.9 in 2014.

8. Standardization followed. The **Open Container Initiative** was founded in June 2015 under the Linux Foundation, by Docker, CoreOS and others. Its runtime and image specifications reached version 1.0 in July 2017, and the distribution specification followed in 2021.
9. `runc` is the OCI reference runtime. containerd and CRI-O are the higher level runtimes that Kubernetes actually calls. Docker Engine today is a developer-facing layer over containerd.
10. Namespaces in the kernel, with their `clone` flags:

Namespace	Isolates	Flag
PID	Process numbering	CLONE_NEWPID
Network	Interfaces, routes	CLONE_NEWNET
Mount	Filesystem tree	CLONE_NEWNS
User	UID and GID mapping	CLONE_NEWUSER
IPC	Shared memory, queues	CLONE_NEWIPC
UTS	Hostname, domain	CLONE_NEWUTS

11. There are also cgroup and time namespaces, added later. cgroup v2, the unified hierarchy, is the current standard on modern distributions.
12. Overlay filesystems: `overlayfs` was merged in kernel 3.18 in 2014 and is the default storage driver. It presents a merged view of lower read-only directories and one upper writable directory, with copy-up on first write.
13. Image layers are content-addressed by SHA-256 digest. The image manifest lists layer digests; the config blob holds the command, environment and history. Pulling an image you already partly have transfers only the missing digests.
14. Additional hardening beyond namespaces and cgroups: seccomp-bpf to restrict system calls, AppArmor or SELinux for mandatory access control, dropping Linux capabilities, and read-only root filesystems.
15. For untrusted workloads, use a stronger boundary: gVisor, which intercepts system calls in userspace, or Kata Containers and AWS Firecracker, which put a genuine lightweight virtual machine around each container. Firecracker, open sourced by AWS in 2018, boots a micro virtual machine in about 125 milliseconds and is what runs AWS Lambda and Fargate.
16. Inspection commands: `lsns` lists namespaces, `nsenter` enters one, `systemd-cgls` and `systemd-cgtop` show cgroup trees, `docker image history` shows the layers and their sizes, `crictl ps` lists containers on a Kubernetes node.

43.5.6 WORDS - remember these

1. Container — an isolated process with its own files and limits — a process group running under restricting namespaces and cgroups on a shared kernel.
2. Namespace — a private view of one kind of system resource — a kernel mechanism partitioning PID, network, mount, user, IPC, UTS and cgroup views.
3. cgroup — a ceiling on how much a process may consume — control group hierarchy enforcing CPU, memory and I/O limits, cgroup v2 unified.
4. Image — a frozen copy of everything the program needs — an ordered set of content-addressed layers plus a config manifest, per the OCI image spec.
5. Layer — one saved change on top of the previous ones — a tar archive of filesystem differences, shared and cached by digest.
6. Registry — the shop where images live — an OCI distribution-spec service storing manifests and blobs, such as Docker Hub or ECR.
7. OCI — the agreed rules so tools interoperate — the Open Container Initiative runtime, image and distribution specifications, founded June 2015.

8. Container escape — breaking out into the host — exploitation of a kernel or configuration flaw to cross the namespace boundary.

43.6 Kubernetes and orchestration

43.6.1 PLAIN - in simple words

1. Containers are easy to run one at a time on your laptop.
2. Now imagine 400 of them, across 30 machines, each needing to find the others, restart when they die, and move when a machine is taken away.
3. Doing that by hand is impossible. You need a program that does it for you. That program is called an **orchestrator**.
4. **Kubernetes** is the one that won. The name is Greek for helmsman, and it is often shortened to K8s, because there are eight letters between K and s.
5. The idea at its heart is simple and worth memorizing: you describe what you want, and the system continuously works to make reality match.
6. You do not say "start three copies". You say "there should be three copies".
7. If one dies at 3am, nobody is paged, because nothing has been violated except temporarily. The system notices the gap and starts another.
8. If you edit the number to five, the system notices the gap and starts two more. The same machinery handles both cases.
9. This is called **declarative reconciliation**: state the desired end, let a control loop close the difference, forever.
10. Everything else in Kubernetes is detail on top of that one idea.

43.6.2 PLAIN - a picture in your head

1. Think of a thermostat on a wall.
2. You do not tell the heater to switch on for eleven minutes. You set 22 degrees.
3. The thermostat measures, compares with your setting, and acts. Then it does it again, forever.
4. Open a window and the temperature drops; the thermostat responds without being told about the window.
5. Kubernetes is a building full of thermostats, one for each kind of thing.
6. One watches "how many copies of this program are running" against "how many should be". Another watches "which machines are alive". Another watches "what traffic goes where".
7. Each loop is small, dull and independent. The intelligence is in the fact that they never stop running.

Where this comparison breaks: a thermostat has one input and one output. Kubernetes controllers act on each other's outputs, so an error can propagate. A wrong setting does not sit still and look wrong; it is enthusiastically and repeatedly enforced across your entire cluster in seconds. That is the same property that makes it powerful and makes mistakes fast.

43.6.3 PLAIN - a worked example

1. Here is a minimal but real pair of Kubernetes objects.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: festival-web
spec:
  replicas: 3
  selector:
    matchLabels:
      app: festival-web
```

```
  template:
    metadata:
      labels:
        app: festival-web
    spec:
      containers:
      - name: web
        image: registry.example.com/festival-site:1.4
        ports:
        - containerPort: 8000
        resources:
          requests:
            cpu: "250m"
            memory: "256Mi"
          limits:
            memory: "512Mi"
        readinessProbe:
          httpGet:
            path: /healthz
            port: 8000
---
apiVersion: v1
kind: Service
metadata:
  name: festival-web
spec:
  selector:
    app: festival-web
  ports:
  - port: 80
    targetPort: 8000
```

2. `kind: Deployment` says this is a managed set of identical copies.
3. `replicas: 3` is the desired state. This one number is the whole contract.
4. `selector.matchLabels` says which pods this deployment considers its own. The labels in the template must match, or nothing works.
5. `image:` names the exact image, including a version tag. Using `latest` here is the classic mistake, because two nodes can then run different code.
6. `requests` is what the scheduler uses to decide where the pod fits. `limits` is what the kernel enforces at runtime.
7. Note there is a memory limit but no CPU limit. That is a deliberate and widely recommended pattern: CPU throttling hurts latency, while unlimited memory risks killing the node.
8. `readinessProbe` tells Kubernetes when this copy may receive traffic. Without it, traffic arrives before the program is ready and users see errors during every deployment.
9. The `Service` gives the three copies one stable internal name and one stable internal address, and spreads connections across whichever copies are ready.
10. Apply and observe:

```
kubectl apply -f festival.yaml
kubectl get deploy,rs,pods -l app=festival-web
kubectl describe pod festival-web-6c9f7d8b4-x2ktr
kubectl logs -f deploy/festival-web
kubectl rollout status deploy/festival-web
kubectl rollout undo deploy/festival-web
```

43.6.4 PLAIN – what is really happening inside

1. `kubectl apply` sends your document to one program, the API server, which validates it and writes it into a database.
2. Nothing has run yet. All that exists is a written wish.
3. The deployment controller notices a Deployment with no matching ReplicaSet and creates one.
4. The ReplicaSet controller notices it should have three pods and has zero, so it creates three pod records. Still nothing is running.
5. The scheduler notices three pods with no node assigned. For each, it filters nodes that could fit, scores the survivors, picks the best, and writes the node name into the pod record.
6. On that node, an agent called the kubelet is watching for pods assigned to it. It sees one, pulls the image, and asks the container runtime to start it.
7. The kubelet then reports status back: starting, running, ready, or failed.
8. The endpoints controller notices a ready pod matching a Service's label selector, and adds its address to that Service's endpoint list.
9. On every node, a component programs the local packet rules so that traffic to the Service address is redirected to one of the current endpoint addresses.
10. If a pod dies, the kubelet reports it, the ReplicaSet controller sees two where three should be, and creates one more. The loop closes.
11. Nobody issued a command. Every step was a controller observing a difference and reducing it.

43.6.5 TECHNICAL – the engineer's version

1. Kubernetes descends from Borg, Google's internal cluster manager, described publicly in the 2015 EuroSys paper Large-scale cluster management at Google with Borg, and from its successor experiment Omega.
2. The first Kubernetes commit was pushed to GitHub on 6 June 2014: 250 files and 47,501 lines of Go, shell and Markdown. Joe Beda, Brendan Burns and Craig McLuckie are the usually named originators, with Borg engineers including Brian Grant and Tim Hockin joining early.
3. Version 1.0 was released in July 2015, and Google donated the project to the newly formed Cloud Native Computing Foundation under the Linux Foundation at the same time. That donation is the reason competitors adopted it.
4. Core objects you must know:

Object	What it is for
Pod	Co-located containers, one unit
ReplicaSet	Keeps N identical pods up
Deployment	Rolling updates of ReplicaSets
Service	Stable name and virtual IP
Ingress	HTTP routing from outside
ConfigMap	Non-secret configuration data
Secret	Base64-encoded sensitive data
Namespace	A naming and policy boundary
Node	One worker machine

5. A pod is the smallest schedulable unit. Containers in a pod share a network namespace, so they reach each other on `localhost`, and can share volumes.
6. Control plane components: the API server, the only component that talks to the datastore; the scheduler, which assigns pods to nodes; the controller manager, which runs the built-in control loops; and `etcd`, a distributed key-value store using the Raft consensus algorithm.
7. On each worker: the kubelet, which owns pod lifecycle on that node, a network proxy component, and a container runtime speaking the Container Runtime Interface, usually containerd or CRI-O.

8. Be honest about Secrets. By default they are base64-encoded, not encrypted, in etcd. Encryption at rest must be enabled explicitly, and anyone with read access to the namespace can read them.
9. Ingress is being superseded. The Gateway API reached general availability for its core resources in October 2023 and is the direction of travel; Ingress remains extremely common and is not going away soon.
10. Scale limits, from the official documentation: no more than 5,000 nodes, 150,000 pods and 300,000 containers in a single cluster, with at most 110 pods per node.
11. Managed offerings remove the control plane work but not the concepts: EKS on AWS from June 2018, AKS on Azure from June 2018, and GKE on Google Cloud from 2015, which was the first.
12. Now the honest section, which matters more than the object list.
13. Most teams do not need Kubernetes. This is not a fashionable opinion; it is what the cost structure says.
14. Kubernetes solves problems that appear at a certain size: many services, many teams, high change rate, bin-packing pressure across many machines, and a genuine need for self-healing across node failures.
15. Below that size it adds a permanent, non-trivial tax: cluster upgrades every few months, a networking plugin to understand, ingress controllers, certificate rotation, RBAC, admission controllers, storage classes, and a new class of failure modes nobody on the team has seen before.
16. A useful test with three questions. Do you run more than about ten distinct services? Do you have someone whose job includes the platform? Would a single machine restarting cause a real outage today?
17. If the answer to all three is no, a managed container service such as ECS, Cloud Run or App Service, or even two virtual machines behind a load balancer, will serve you better and cost less in time.
18. Where experts disagree: some argue the standard interface is worth the tax even for small teams, because hiring and tooling are common. Others argue that complexity you do not need is complexity that will page you at 3am. Both are stated in good faith and depend on team size and turnover.

43.6.6 WORDS - remember these

1. Orchestrator — the program that runs your containers for you — a cluster scheduler and lifecycle manager across many hosts.
2. Pod — the smallest thing Kubernetes runs — one or more containers sharing a network namespace and storage volumes, scheduled as a unit.
3. Deployment — the object that keeps N copies of your app alive — a controller managing ReplicaSets to provide declarative rolling updates.
4. Reconciliation — repeatedly closing the gap between wish and reality — a level-triggered control loop comparing desired and observed state.
5. etcd — the cluster's memory — a strongly consistent distributed key-value store using Raft, the only stateful control plane component.
6. kubelet — the agent on each machine — the node-level daemon that starts pods assigned to it and reports their status.
7. Service — a stable name in front of moving copies — a virtual IP and DNS name load-balancing to the current ready endpoints.

43.7 The Bezos API mandate

43.7.1 PLAIN - in simple words

1. Around 2002, Jeff Bezos sent an internal message to everyone building software at Amazon.
2. It said, in effect: every team must publish its work as a service that other teams call over the network, and no team may reach into another team's data any other way.
3. That sounds like a boring technical rule. It was one of the most consequential management decisions in the history of the industry.

4. Before it, teams shared databases and linked directly to each other's code. That is fast to write and impossible to change later.
5. After it, the only way to use another team's work was to call its published interface, over the network, like an outsider.
6. And the last rule was the sharp one: build every interface as if a stranger outside the company would use it.
7. Nobody was selling anything to outsiders yet. But once every internal service was already built for outsiders, selling it later was a small step rather than a rewrite.
8. That is the link between the memo and AWS. AWS is what you get when a company spends four years turning itself into a set of public-quality services and then notices it can sell them.
9. Amazon never published the memo. What everyone quotes is one former employee's recollection, written down in 2011.

43.7.2 PLAIN – a picture in your head

1. Think of a large hospital where every department shares one giant paper filing cabinet.
2. Radiology reaches into the pharmacy's drawer to check a prescription. Pharmacy reaches into billing's drawer. Everyone rummages everywhere.
3. It works, and it is fast, right up until one department reorganizes its drawer. Then four other departments break, and nobody knows who or why.
4. Now the director issues a rule: no department may open another department's drawer. If you need something, you fill in that department's request form and they answer it.
5. At first this is slower and everybody hates it. Forms take longer than rummaging.
6. But now pharmacy can completely redesign its drawer over a weekend, and nobody else notices, because the form did not change.
7. And one day the hospital realizes that since every department already answers formal requests, it can answer requests from other hospitals too, for a fee.

Where this comparison breaks: a paper form is slow but reliable. A network call can be slow, can fail halfway, can succeed while appearing to fail, and can arrive twice. The hospital analogy makes the mandate look like pure gain. The real version bought flexibility with a permanent bill in latency and in failure handling, and that bill never stops being paid.

43.7.3 PLAIN – a worked example

1. Here is the mandate as it is publicly reported. The source is a long post by Steve Yegge, an engineer who worked at Amazon for about six years and then at Google. He wrote it in October 2011 as an internal Google memo and posted it publicly by accident.
2. His introduction is "His Big Mandate went something along these lines". The rules, quoted:
3. One. "All teams will henceforth expose their data and functionality through service interfaces."
4. Two. "Teams must communicate with each other through these interfaces."
5. Three. "There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network."
6. Four. "It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols – doesn't matter."
7. Five. "All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions."
8. Then the closing line: "Anyone who doesn't do this will be fired."
9. The honest version: in Yegge's original post this is a list of seven items, not five plus a closing line. Item six is the firing line, and item seven is a one-line pleasantry, which he immediately identifies as his own joke. The five substantive rules plus the firing line is the accurate summary.

10. Yegge also wrote that item six "was quite real, so people went to work". He was recalling, not quoting a document. Amazon has never released the memo, so the wording is a recollection and should always be described that way.
11. Note what rule four is not doing. It does not pick a technology. That is deliberate and correct: the mandate is about boundaries, not protocols.

43.7.4 PLAIN - what is really happening inside

1. Follow what the rule forces, step by step. This is the part that matters.
2. Once teams may only call each other over the network, they must first be able to find each other. So somebody has to build a service registry, and the registry is itself a service.
3. Once hundreds of services call each other, any team can accidentally flood another. Yegge's words: "every single one of your peer teams suddenly becomes a potential DOS attacker". So quotas and throttling become mandatory everywhere, not optional.
4. Once a request crosses twenty services, an alarm has to reach the right team. Yegge again: a ticket "might bounce through 20 service calls before the real owner is identified", and at fifteen minutes per bounce that is hours. So you must build ownership metadata and automated pager routing.
5. Once you cannot see inside another team's code, "monitoring and QA are the same thing". A service that answers "I am fine" may have nothing working except the part that says "I am fine". So health checks must exercise real behaviour, and at that point monitoring has become automated testing.
6. Once debugging spans teams, it "is basically impossible unless there is a universal standard way to run every service in a debuggable sandbox". So you must build sandboxes and standardized local environments.
7. Look at that list again: discovery, quotas, paging, deep health checking, sandboxes. Every one of those is a component of a public cloud platform.
8. Amazon did not set out to build platform tooling. The mandate made it unavoidable, and four years later they had it.

43.7.5 TECHNICAL - the engineer's version

1. Organizational consequence one: the **two-pizza team**. Bezos's stated rule was that a team should be small enough to be fed by two pizzas, roughly six to ten people. The number is not the point; the bound on communication paths is.
2. Organizational consequence two: **you build it, you run it**. Werner Vogels, Amazon's Chief Technology Officer, stated it in an ACM Queue interview with Jim Gray in 2006. The team that writes a service also operates it and carries its pager.
3. That closes a loop. If the author of the 3am page is also the person woken by it, quality arguments stop needing management.
4. Organizational consequence three: **Conway's law**. Melvin Conway wrote in How Do Committees Invent?, published in Datamation in April 1968, that organizations produce designs which copy their own communication structures.
5. The mandate is Conway's law used deliberately. Change the communication structure, and the architecture follows. This is sometimes called the inverse Conway manoeuvre.
6. Now the costs, stated plainly, because the reader is owed them.
7. Latency. An in-process function call is on the order of nanoseconds. A call to another service in the same datacentre is on the order of 0.5 to 2 milliseconds. That is a factor of roughly a million.

Call type	Typical latency
Function call in process	1 to 100 ns
Same-host loopback call	50 to 200 us
Same-AZ service call	0.5 to 2 ms
Cross-region call	50 to 250 ms

Call type	Typical latency

8. Failure semantics. A local call either returns or throws. A network call can also time out with the work completed, be retried and applied twice, or succeed with the response lost. Every service boundary therefore needs idempotency keys, retries with backoff, timeouts and circuit breakers.
9. Data integrity. You can no longer wrap two teams' updates in one database transaction. You are pushed into eventual consistency and compensating actions.
10. Operational cost. Each service needs its own build, deployment, monitoring, on-call rota, dependency updates and security patching.
11. The most important honest point in this chapter: most companies that copied the microservice structure did not copy the mandate's discipline.
12. The mandate had four properties that copies usually lack. It was absolute, with no exceptions. It was enforced from the top. It required interfaces to be externalizable, which forces documentation, versioning and stability. And it was accompanied by real investment in the platform tooling that the consequences demanded.
13. Split a system into thirty services without discovery, quotas, tracing, ownership routing and sandboxes, and you have not built a service-oriented architecture. You have built a distributed monolith, which has all the latency and failure modes of distribution and none of the independence.
14. Where experts disagree: some argue the mandate is a timeless engineering principle, others that it was a response to a specific scale and coupling crisis at Amazon in 2001 and 2002, and that applying it to a twelve-person company is cargo cult. The second view is the majority among practitioners today.

43.7.6 WORDS - remember these

1. API mandate — the 2002 rule that teams talk only through published interfaces — an organization-wide prohibition on any interprocess communication other than service calls.
2. Externalizable — built as if a stranger will use it — designed with the documentation, versioning and stability required for third-party exposure.
3. Two-pizza team — a team small enough to feed with two pizzas — an organizational unit sized to bound internal communication paths.
4. You build it, you run it — the authors carry the pager — development teams own production operation of their own services.
5. Conway's law — you ship your org chart — systems mirror the communication structure of the organizations that design them, Conway 1968.
6. Distributed monolith — services that must all deploy together — a system with the costs of distribution and the coupling of a monolith.

43.8 How AWS actually started

43.8.1 PLAIN - in simple words

1. By 2000 Amazon was a large online shop with a serious internal problem.
2. Every new project spent months on the same groundwork: getting machines, setting up databases, storage, networking and monitoring.
3. Teams were reinventing the same plumbing over and over, and waiting on each other to do it.
4. The API mandate had already forced them to expose everything as a service. The next thought was obvious: make the infrastructure itself a service.
5. In 2003, engineers wrote a proposal describing computing capacity offered as a service, so that any team could ask for a server through an interface.
6. In parallel, a senior manager wrote a proposal arguing that this could be a business, sold to the outside world.

7. Amazon approved both. Development of the compute service was set up in Cape Town, South Africa, of all places, because the engineer leading it wanted to move home.
8. The public services arrived in 2006: storage in March, and rentable computers in August.
9. It was not a side project selling leftovers. It was a deliberate new business with its own staffing plan.

43.8.2 PLAIN - a picture in your head

1. Imagine a large restaurant chain that keeps building its own kitchens.
2. Every new branch designs plumbing, gas, extraction and cold storage from scratch, badly, and slowly.
3. Head office finally standardizes: one kitchen design, ordered as a kit, ready in a week.
4. Branches stop thinking about plumbing and start thinking about food. Opening a branch goes from a year to a month.
5. Then somebody points out that other restaurant chains have the same problem, and that the kit could be sold.
6. The kit business becomes larger and far more profitable than the original restaurants.

Where this comparison breaks: a kitchen kit is a product you build once and ship. AWS had to be operated forever, at growing scale, for customers who could leave. Also, the myth this analogy tempts you into is that head office sold its spare kitchens. It did not. It designed and built a product from scratch, then sold it, which is a completely different thing.

43.8.3 PLAIN - a worked example

1. The dates, in order, from public records.
2. July 2002: Amazon launches Amazon.com Web Services, a free interface letting affiliates pull product data. This is where the name comes from. It is not cloud computing.
3. 2003: Chris Pinkham and Benjamin Black write a short internal paper proposing standardized, automated infrastructure delivered as a service, and noting that virtual servers could be sold.
4. 2003: Andy Jassy, who had joined Amazon in 1997 and become Bezos's technical assistant in 2001, writes a separate proposal for the business, including a plan for the first 57 hires. It is approved by the senior team.
5. 2004: Pinkham moves to Cape Town and opens a development centre there to build the compute service.
6. 3 November 2004: Simple Queue Service launches in beta. Chronologically this is the first thing recognizably cloud.
7. 14 March 2006: Simple Storage Service, S3, launches generally available, at 15 US cents per gigabyte-month.
8. 25 August 2006: Elastic Compute Cloud, EC2, launches in limited beta. One instance size only, 1.7 GHz, 1.75 GB of memory, 160 GB of local disk, at 10 US cents per hour. It reached general availability in October 2008.
9. From 10 cents an hour in 2006 to 128.7 billion US dollars of revenue in 2025 is the whole arc.

43.8.4 PLAIN - what is really happening inside

1. Why Amazon and not IBM, Microsoft, Sun or a telephone company.
2. First, Amazon already had to run huge, spiky, low-margin infrastructure, and was already good at it out of necessity.
3. Second, the API mandate had already forced internal services to be built for external use. The hard organizational work was done before the product was conceived.
4. Third, retail is a low-margin business, so Amazon's culture was to treat infrastructure cost as an enemy. A company used to 3 per cent margins prices very differently from one used to 60 per cent margins.
5. Fourth, nobody else wanted the customers. In 2006 the buyers were startups with no money. Established vendors sold to enterprises with purchase orders and salespeople, and a self-service credit-card product looked like a toy.

6. Fifth, Amazon was willing to be unprofitable for years and did not have to protect an existing licence business. Microsoft and others did.
7. Now correct the myth. The popular story is that AWS was built to rent out servers left idle after the Christmas shopping peak.
8. It is false. Werner Vogels, Amazon's Chief Technology Officer, publicly rejected it in 2011, and Andy Jassy has repeated the correction many times.
9. The mechanics also do not work. Retail's peak is Q4, exactly when external customers would need capacity most, so you would have to evict them at the worst moment. And AWS ran on different hardware, in different facilities, with different software, from the start.
10. The kernel of truth is smaller and duller: Amazon's own experience of running infrastructure badly and expensively is what taught it what the product should be.

43.8.5 TECHNICAL - the engineer's version

1. The service launch timeline, with dates as published:

Year	Service	Date
2004	SQS beta	3 Nov 2004
2006	S3	14 Mar 2006
2006	EC2 beta	25 Aug 2006
2007	SimpleDB beta	14 Dec 2007
2008	EBS	20 Aug 2008
2008	CloudFront	18 Nov 2008
2009	ELB, CloudWatch	18 May 2009
2009	VPC	25 Aug 2009
2009	RDS	26 Oct 2009
2010	Route 53	5 Dec 2010
2012	DynamoDB	19 Jan 2012
2012	Redshift	28 Nov 2012
2014	Lambda, ECS	13 Nov 2014
2017	Fargate, Nitro C5	Nov 2017
2018	EKS general availability	June 2018

2. Read the order. Storage, then compute, then block storage, then content delivery, then load balancing and monitoring, then networking, then managed databases. Each launch is a response to what the previous one made painful.
3. VPC arriving three years after EC2 is the most instructive gap. The original EC2 put every instance on a flat shared network with a public address, which enterprises would not accept.
4. Financially, AWS was reported separately for the first time in Amazon's first quarter 2015 results, revealing a business with roughly 5 billion US dollars of annualized revenue that was already profitable. That disclosure repriced Amazon's stock and forced competitors to respond.
5. Current figures, from company filings. Full year 2025 AWS net sales were 128,725 million US dollars, with segment operating income of 45,606 million, an operating margin near 35 per cent, against total Amazon net sales of 716,924 million.
6. In the quarter ending June 2026, AWS reported 42.2 billion US dollars, growth of about 37 per cent, and operating income of 16.6 billion, which Amazon's chief executive described as the fastest growth in eighteen quarters.
7. Reference customer for scale: Netflix began moving to AWS in 2008 after a database corruption incident halted DVD shipping for three days, and completed the shutdown of its last own-datacentre systems in January 2016. It took seven years, and Netflix is a well-resourced engineering organization.

8. Where sources disagree. Some accounts credit the 2003 Pinkham and Black paper as the origin of AWS; others credit Jassy's business proposal; Black himself has written that theirs was one of several parallel efforts. The honest summary is that both existed in 2003 and both mattered.

43.8.6 WORDS - remember these

1. AWS — Amazon's cloud business — Amazon Web Services, launched publicly 2006, 128.7 billion US dollars of net sales in 2025.
2. EC2 — rentable computers by the hour — Elastic Compute Cloud, virtual machines billed per second with a 60 second minimum, beta August 2006.
3. S3 — a place to put files, reached over the web — Simple Storage Service, object storage launched 14 March 2006, the first generally available service.
4. SQS — a queue you do not have to run — Simple Queue Service, beta November 2004, chronologically the first AWS service.
5. Two-pizza origin — small teams owning services — the organizational precondition that let Amazon staff many independent services at once.
6. Spare capacity myth — the false story that AWS resold leftover retail servers — publicly denied by Amazon's CTO in 2011 and contradicted by the timeline.

43.9 The AWS services you actually need to know

43.9.1 PLAIN - in simple words

1. AWS advertises more than 200 services. You will use about eighteen.
2. Here is each one, in a sentence, as the problem it solves.
3. EC2: I need a computer. Rent a virtual machine by the second.
4. S3: I need somewhere to put files that never fills up. Store objects by name, reached over the web.
5. EBS: my rented computer needs a hard disk that survives a reboot. Attach a network-backed disk to one machine.
6. VPC: I need my machines on a private network of my own design, not on the open internet.
7. Route 53: I need names to turn into addresses, and I need to steer traffic. This is DNS, run for you.
8. RDS: I need a normal database, but I do not want to install, patch, back up or fail it over myself.
9. DynamoDB: I need a store that never slows down as it grows, and I can live with looking things up only by key.
10. Lambda: I need to run a small piece of code when something happens, and I do not want a server sitting there waiting.
11. IAM: I need to say exactly who may do what. This is the permission system, and it applies to every other service.
12. CloudFront: my users are far away and my files are large. Keep copies near them.
13. ELB and ALB: I have several machines and one address. Spread the traffic and skip the dead ones.
14. SQS: I need to hand work to something else without waiting for it, and without losing it if the other side is down.
15. SNS: I need to tell many interested parties that something happened.
16. ECS: I have containers and I want them run for me, simply.
17. EKS: I have containers and I want Kubernetes, without operating the control plane myself.
18. CloudWatch: I need to see what is happening. Logs, numbers and alarms.
19. CloudFormation: I want my whole setup written down as a file, so it can be rebuilt exactly.
20. Secrets Manager: I need to keep passwords somewhere that is not the source code.

43.9.2 PLAIN – a picture in your head

1. Think of building a shop in a market town.
2. EC2 is renting the shop unit. EBS is the storeroom attached to your unit. S3 is the town warehouse where anyone with a key can drop off boxes.
3. VPC is the private courtyard you rent, with your own gate and your own path layout. Route 53 is the street sign that tells people where your shop is.
4. ELB is the doorman who splits an arriving crowd between your three counters and stops sending people to the one that has closed.
5. RDS is a bookkeeper the market provides, who keeps your ledger, backs it up nightly and covers for himself when ill.
6. SQS is the in-tray by the back door. Deliveries go in whether you are ready or not, and you take them at your pace.
7. IAM is the master key system: who may open which door, when, from where.
8. CloudWatch is the set of meters and the alarm bell. CloudFormation is the architect's drawing from which the whole shop could be rebuilt.

Where this comparison breaks: in a market town you rent one unit and that is the bill. Here, almost every one of these has three or four separate meters running at once, and the doorman charges by the crowd as well as by the hour.

43.9.3 PLAIN – a worked example

1. A three-tier web application, in AWS parts, in the order you would create them.
2. VPC with six subnets across two availability zones: two public, two private for applications, two private for the database.
3. An Application Load Balancer in the public subnets, holding the TLS certificate.
4. An Auto Scaling group of EC2 instances in the private application subnets, each with a gp3 EBS root volume.
5. RDS for PostgreSQL in the database subnets, with a standby in the second availability zone.
6. S3 for user uploads, with CloudFront in front of it for delivery.
7. Route 53 holding the hosted zone and an alias record pointing at the load balancer.
8. IAM roles attached to the instances so that no access key is ever written to disk.
9. Secrets Manager holding the database password, fetched at startup.
10. CloudWatch collecting logs and metrics, with alarms on error rate and database CPU.
11. CloudFormation or Terraform describing all of the above in one repository.

43.9.4 PLAIN – what is really happening inside

1. Every one of these is reached the same way: an HTTPS request to a region-specific endpoint, signed with a key derived from your credentials.
2. That signature scheme is called Signature Version 4. It hashes the request, the date, the region and the service name, so a signature stolen for one region and service is useless elsewhere.
3. The AWS console, the `aws` command line tool and every language SDK all do exactly this. There is no private channel.
4. Services are separated into control plane operations, which create and modify resources, and data plane operations, which use them.
5. That split matters during outages. A control plane failure stops you creating new things while existing things keep serving.
6. Almost every service integrates with two others by default: IAM decides whether the call is allowed, and CloudTrail records that it happened.

43.9.5 TECHNICAL - the engineer's version

1. Equivalents across the three large providers. Names are current as of August 2026; providers rename services often, so check before quoting.

AWS	Azure	Google Cloud
EC2	Virtual Machines	Compute Engine
S3	Blob Storage	Cloud Storage
EBS	Managed Disks	Persistent Disk
VPC	Virtual Network	VPC
Route 53	Azure DNS	Cloud DNS
RDS	Azure SQL, Flexible	Cloud SQL
DynamoDB	Cosmos DB	Firestore, Bigtable
Lambda	Azure Functions	Cloud Run functions
IAM	Entra ID plus RBAC	Cloud IAM
CloudFront	Azure Front Door	Cloud CDN
ALB	Application Gateway	Cloud Load Balancing
SQS	Storage or Service Bus	Pub/Sub
SNS	Event Grid	Pub/Sub
ECS	Container Apps	Cloud Run
EKS	AKS	GKE
CloudWatch	Azure Monitor	Cloud Monitoring
CloudFormation	ARM, Bicep	Deployment Manager
Secrets Manager	Key Vault	Secret Manager

2. The mapping is approximate on purpose. DynamoDB and Cosmos DB have different consistency models. SQS is a queue with one consumer per message; Pub/Sub is a topic with fan-out, so it maps to both SQS and SNS.
3. Durability and availability commitments, as published in the service level agreements: S3 Standard is designed for 99.999999999 per cent durability, eleven nines, with a 99.9 per cent availability service level agreement.
4. Eleven nines means that for ten million objects you would statistically expect to lose one object every ten thousand years. It is a design target derived from redundancy, not a measured result.
5. EC2 offers a 99.99 per cent regional service level agreement when instances are spread across two or more availability zones, and 99.5 per cent for a single instance with certain storage configurations.
6. Storage duplication: S3 Standard stores objects redundantly across at least three availability zones. S3 One Zone-IA deliberately does not, which is why it is cheaper and why you must not use it for anything irreplaceable.
7. Service quotas are real and will surprise you: default limits such as 5 VPCs per region, 5 Elastic IP addresses per region, and per-instance-family vCPU limits are common causes of failed deployments. Check `aws service-quotas list-service-quotas`.
8. Useful commands: `aws sts get-caller-identity` answers "who am I", which is the first thing to run when something is denied; `aws configure list` shows which credentials and region are in effect.

43.9.6 WORDS - remember these

1. EC2 — a rented computer — Elastic Compute Cloud instances, billed per second with a 60 second minimum, in families tuned for compute, memory or storage.
2. S3 — files stored by name on the web — object storage with eleven nines design durability and a flat key namespace per bucket.
3. EBS — a disk for one machine — network-attached block storage, gp3 by default, snapshottable, tied to one availability zone.

4. VPC — your own private network in the cloud — a logically isolated virtual network with your own address ranges, subnets and route tables.
5. IAM — the rules about who may do what — Identity and Access Management, evaluating identity and resource policies on every API call.
6. Lambda — run code without a server — a function-as-a-service runtime billed per millisecond of execution and per request.
7. Signature Version 4 — how a request proves it is really you — the AWS request signing scheme scoped to date, region and service.

43.10 Azure, Google Cloud and the rest

43.10.1 PLAIN - in simple words

1. Microsoft announced its cloud, then called Windows Azure, in October 2008, and opened it for paying customers on 1 February 2010.
2. Its first version made a bet that turned out to be wrong. It offered a platform: you gave it your application, it ran it, and you never saw a machine.
3. That was elegant, but customers did not want elegant. They wanted to move the servers they already had, unchanged.
4. So Microsoft turned around and added plain virtual machines, and then Linux, and then everything else. It renamed itself Microsoft Azure in 2014 to signal that it was no longer a Windows product.
5. That late start cost Microsoft the developer-first market, which AWS held.
6. But Microsoft had something AWS did not: it was already inside almost every large company on earth, selling Windows, Office and, crucially, the identity system that decides who employees are.
7. So Azure won the enterprise from the inside. If your staff already log in with a Microsoft account, the cloud that understands that account has an enormous advantage.
8. Google Cloud started earlier in one narrow sense and later in every other. It is respected for containers, networking and data analysis, and is a clear third by revenue.
9. Below the big three there is a real second tier, and for many small projects it is cheaper and simpler.

43.10.2 PLAIN - a picture in your head

1. Imagine three shops selling building materials.
2. The first, AWS, opened years earlier, stocks absolutely everything, and its staff assume you are a professional builder. The catalogue is enormous and the labels are terse.
3. The second, Azure, is owned by the company that already supplied the doors, locks and keys for every office in town. It says: your existing keys work here, and we will send someone to your site.
4. The third, Google Cloud, is run by engineers who built the largest structures in the world for themselves, and now sell the tools. The tools are excellent. The shop assistants are fewer.
5. Around the corner are small shops selling four or five items at half the price, with a printed price list you can understand in a minute.

Where this comparison breaks: building materials are interchangeable between shops. Cloud services are not. Once your identity, your networking and your managed databases are with one provider, moving is a project measured in quarters, not an afternoon of carrying bricks.

43.10.3 PLAIN - a worked example

1. Azure's naming conventions are worth learning, because they are consistent and they differ from AWS.
2. Everything lives in a **resource group**, which is a folder for related resources with a shared lifecycle. AWS has no exact equivalent; the closest habits are tags and stacks.
3. Above resource groups sit subscriptions, which are the billing boundary, and above those, management groups, which apply policy.
4. Identity is not per subscription. It comes from a **tenant** in Entra ID, which can span many subscriptions.

5. So the Azure hierarchy reads: tenant, management group, subscription, resource group, resource.
6. In AWS the hierarchy reads: organization, organizational unit, account, resource. There is no resource group layer, which is why AWS practitioners use separate accounts for separation far more aggressively.
7. Creating a machine, side by side:

```
# Azure
az group create -n festival-rg -l centralindia
az vm create -g festival-rg -n web1 \
  --image Ubuntu2404 --size Standard_B2s \
  --admin-username azureuser --generate-ssh-keys

# AWS
aws ec2 run-instances --image-id ami-0abcd1234 \
  --instance-type t3.medium --key-name mykey \
  --subnet-id subnet-0123456789abcdef0
```

43.10.4 PLAIN – what is really happening inside

1. The reason Entra ID is described as Microsoft's real moat is worth understanding mechanically.
2. Most large organizations authenticate employees against Active Directory, a system Microsoft shipped in Windows 2000. It holds users, groups, machines and policies.
3. Entra ID, which was called Azure Active Directory until Microsoft renamed it on 11 July 2023, is the cloud version. It issues tokens for web applications using OAuth 2.0 and OpenID Connect.
4. Once employees sign in to Microsoft 365 with Entra ID, the same identity can authorize Azure resources, third-party software and internal applications.
5. That means the decision "which cloud" is partly already made by a decision taken years earlier about email and file sharing.
6. Switching cost is therefore not mainly about servers. It is about identity, group membership, conditional access rules and hundreds of application integrations.
7. Hybrid follows from the same fact. Azure Arc, Azure Stack and ExpressRoute exist so that machines in a company's own building appear in the same management and identity plane as machines in Azure.

43.10.5 TECHNICAL – the engineer's version

1. Azure timeline. Announced at the Professional Developers Conference on 27 October 2008 by Ray Ozzie, under the internal codename Red Dog, led by Dave Cutler, the architect of Windows NT.
2. Commercially available 1 February 2010, initially as a platform service with web roles and worker roles, Blob storage, SQL Azure and Service Bus.
3. Infrastructure services, meaning ordinary virtual machines, entered preview in June 2012 and reached general availability in April 2013. Renamed from Windows Azure to Microsoft Azure on 25 March 2014.
4. Position today: Azure passed 100 billion US dollars of annual revenue in Microsoft's fiscal year 2026, growing 41 per cent from 75 billion in fiscal

2025. Microsoft states over 70 Azure regions.

5. Google Cloud timeline. App Engine, a platform service, previewed 7 April

2008. Compute Engine was announced at Google I/O on 28 June 2012 in limited preview, opened broadly in May 2013 and reached general availability in December 2013. Google Kubernetes Engine launched in 2015, the first managed Kubernetes service.

6. Google Cloud's strengths are concentrated: Kubernetes, since it wrote it; networking, since it owns an unusually large private backbone; and data analytics, where BigQuery separates storage from compute and is genuinely ahead. Revenue was 17.664 billion US dollars in the December 2025 quarter, up 48 per cent.

Provider	Regions	Recent revenue
AWS	39, 123 AZs	128.7 bn, FY2025
Azure	Over 70	Over 100 bn, FY2026
Google Cloud	43, 130 zones	17.7 bn, Q4 2025

7. The rest of the field, briefly and honestly. Oracle Cloud Infrastructure competes on database licensing terms and, since 2024, on large AI compute contracts. IBM Cloud is strongest in regulated industries and mainframe adjacency. Alibaba Cloud leads in China and much of South East Asia.
8. DigitalOcean, Hetzner and OVHcloud sell simple virtual servers at a fraction of hyperscaler list prices, with flat pricing and generous or unmetered traffic. For a small production workload the difference can be five to ten times.
9. Cloudflare is a different shape: a network-first provider whose Workers run code at edge locations and whose R2 object storage charges no egress fee, which is a direct attack on the industry's most disliked charge.
10. The trade is real. The second tier gives you fewer managed services, fewer compliance certifications, smaller regional footprints and less depth of support. For a database-plus-web-server application, that often does not matter. For a regulated enterprise, it does.

43.10.6 WORDS - remember these

1. Azure — Microsoft's cloud — announced October 2008, commercially available 1 February 2010, over 100 billion US dollars in fiscal 2026.
2. Entra ID — the Microsoft login your company already uses — the cloud identity service formerly named Azure Active Directory, renamed 11 July 2023.
3. Resource group — an Azure folder for related things — a management and lifecycle boundary containing resources within one subscription.
4. Hybrid cloud — some machines yours, some rented, managed together — a deployment model joining on-premises infrastructure with public cloud under one control and identity plane.
5. Hyperscaler — one of the very large providers — an operator running global fleets at a scale where custom hardware and custom networks are economic.
6. Egress-free storage — object storage with no charge to read data out — Cloudflare R2's pricing model, contrasted with per-gigabyte egress charges.

43.11 The service models: IaaS, PaaS, SaaS and FaaS

43.11.1 PLAIN - in simple words

1. The models differ in one thing only: where the line is drawn between what you look after and what the provider looks after.
2. **IaaS**, infrastructure as a service. You rent machines, disks and networks. You install and patch the operating system, the runtime and your code. You have the most control and the most work.
3. **PaaS**, platform as a service. You give the provider your code. It provides the operating system, the runtime, scaling and deployment. You lose some control and most of the work.
4. **SaaS**, software as a service. You do not write anything. You use finished software over the web and configure it. Gmail, Salesforce, Zoom.
5. **FaaS**, function as a service, also called serverless. You give the provider one function. It runs it only when something triggers it, and charges only for the milliseconds it runs.
6. These are not four products. They are four points on one line, and the line is "how much of the stack do I own".
7. Nothing forces you to pick one. Real systems mix all four, usually without anybody deciding to.

43.11.2 PLAIN – a picture in your head

1. The standard analogy is pizza as a service, and it is genuinely useful.
2. Made at home: you buy flour, cheese, an oven, gas and a table. That is owning your own servers.
3. Take and bake: you buy a ready pizza and use your own oven, gas and table. That is infrastructure as a service.
4. Delivered: someone else makes and cooks it; you supply the table, the drinks and the dining room. That is platform as a service.
5. Dining out: you supply nothing but yourself and money. That is software as a service.

Where this comparison breaks, and it breaks in three specific places. First, it suggests you own less as you move along, when in fact you still own all the important things: your data, your access rules and your legal obligations. Second, pizza has one dimension of choice. Cloud stacks let you take the oven from one supplier and the dough from another, which the analogy cannot express. Third, and worst, the analogy implies dining out costs the most. In cloud pricing the fully managed option is often the cheapest at small scale and the most expensive at large scale, and the crossover is the whole argument.

43.11.3 PLAIN – a worked example

1. The same task, a small web application, at each level.
2. IaaS: launch an EC2 instance, install Ubuntu updates, install Python, install a web server, write a service unit file, configure log rotation, set up certificate renewal, and repeat all of it when you scale out.
3. PaaS: push your code to Azure App Service or Google Cloud Run. It builds a container, runs it, terminates TLS, scales copies with traffic, and rolls back if health checks fail.
4. FaaS: write one function that handles one request, register it against an HTTP trigger, and let the provider create and destroy execution environments as demand moves.
5. SaaS: realize you were building a form-and-spreadsheet tool and buy one.

Layer	IaaS	PaaS
Application code	You	You
Runtime, libraries	You	Provider
Operating system	You	Provider
Virtualization	Provider	Provider
Servers, network	Provider	Provider

43.11.4 PLAIN – what is really happening inside

1. The important consequence of the split is not convenience. It is who is accountable when something goes wrong.
2. This is formalized as the **shared responsibility model**, and every major provider publishes a version of it.
3. The provider is responsible for security **of** the cloud: the buildings, the hardware, the hypervisor, the physical network, and the managed service software itself.
4. The customer is responsible for security **in** the cloud: operating system patching where applicable, network rules, identity and permissions, data encryption choices, and application code.
5. The line moves with the model. On IaaS you patch the guest operating system. On PaaS the provider does. On SaaS almost everything technical is theirs.
6. One thing never moves. Access control and data classification are always yours, in every model, without exception.
7. This is why almost every publicized cloud breach is a customer-side misconfiguration rather than a provider failure: a storage bucket made public, a permission granted too widely, a key committed to a repository.

43.11.5 TECHNICAL - the engineer's version

1. The definitions are standardized in NIST SP 800-145, September 2011, which defines IaaS, PaaS and SaaS. FaaS post-dates it and is generally treated as a specialization of PaaS.
2. Responsibility split, stated as the providers state it:

Concern	IaaS	PaaS	SaaS
Data and access	You	You	You
Application code	You	You	Provider
Runtime, middleware	You	Provider	Provider
Guest OS patching	You	Provider	Provider
Host and hardware	Provider	Provider	Provider

3. Encryption responsibilities are subtler than the table suggests. Providers supply encryption at rest by default in most storage services, but key custody is a choice: provider-managed keys, customer-managed keys in a key management service, or customer-supplied keys held outside entirely.
4. Compliance is inherited, not transferred. A provider's ISO/IEC 27001 or SOC 2 report covers their controls. Your auditor still tests yours. The document that connects the two is the provider's customer responsibility matrix.
5. Service level agreements are financial instruments, not promises of uptime. A 99.99 per cent monthly agreement permits about 4 minutes 23 seconds of downtime per month, and the remedy for breach is typically a service credit of 10 to 100 per cent of that service's fee. It does not compensate your business loss.
6. Availability arithmetic that engineers get wrong: dependencies multiply. A system depending on four independent components each at 99.9 per cent has a theoretical availability of 0.999 to the fourth power, which is 99.6 per cent, or about three hours per month.

43.11.6 WORDS - remember these

1. IaaS — renting raw machines — infrastructure as a service, where the tenant manages the guest operating system and everything above it.
2. PaaS — hand over code, get a running service — platform as a service, provider-managed runtime, scaling and deployment.
3. SaaS — finished software you just use — software as a service, delivered multi-tenant over the network, configured not coded.
4. FaaS — one function, run on demand — function as a service, event-triggered, billed per invocation and per millisecond.
5. Shared responsibility — who is to blame for what — the provider secures the cloud, the customer secures what they put in it.
6. Service level agreement — the promised uptime and the refund if missed — a contractual availability target with service credits as the sole remedy.

43.12 Serverless, properly

43.12.1 PLAIN - in simple words

1. Serverless does not mean there are no servers. There are servers. You do not see them, name them, patch them or pay for them when idle.
2. A better name would have been "someone else's servers, allocated per request". That name lost.
3. You upload a function: a piece of code with one entry point.
4. You attach a **trigger**: an HTTP request, a file appearing in storage, a message in a queue, a database change, a timer.

5. When the trigger fires, the provider finds or creates an execution environment, runs your function, and returns the result.
6. You are billed for the milliseconds it ran and the memory it was given. Between events the cost is zero.
7. If a thousand events arrive at once, a thousand copies run at once. You did not configure that.
8. The catch is that each run starts with no memory of any previous run. Nothing you leave in a variable can be relied on next time.
9. The other catch is that the very first run after a pause is slower, because the environment has to be created. That is the **cold start**.

43.12.2 PLAIN - a picture in your head

1. Think of a hotel meeting room you can book by the minute.
2. Ordinary servers are like renting an office all year: it is yours, your things are on the desk, and you pay in August when nobody is there.
3. Serverless is like walking to reception, being handed an empty room in seconds, using it for ninety seconds, and leaving.
4. The room is always empty when you arrive. Anything you left last time is gone.
5. Most of the time a cleaned room is already waiting and you walk straight in. That is a warm start.
6. Occasionally none is ready, and staff must prepare one while you wait. That is a cold start.
7. If two hundred of your colleagues arrive at once, reception opens two hundred rooms, and you get one bill for two hundred short bookings.

Where this comparison breaks: a hotel has a fixed number of rooms and will eventually say no. Serverless platforms feel unlimited but have concurrency quotas, and hitting one produces throttling errors rather than a polite refusal. Also, in a hotel you pay per room. Here you pay per room multiplied by its size, so a function given four times the memory costs four times as much per second, even when it does not use the memory.

43.12.3 PLAIN - a worked example

1. A real AWS Lambda function in Python, triggered when an image is uploaded to S3, which records the upload in DynamoDB.

```
import os, json, boto3, urllib.parse

ddb = boto3.resource("dynamodb")
table = ddb.Table(os.environ["TABLE_NAME"])

def handler(event, context):
    for record in event["Records"]:
        bucket = record["s3"]["bucket"]["name"]
        key = urllib.parse.unquote_plus(
            record["s3"]["object"]["key"])
        size = record["s3"]["object"]["size"]
        table.put_item(Item={
            "pk": f"{bucket}/{key}",
            "size": size,
            "eventTime": record["eventTime"],
        })
    return {"statusCode": 200,
            "body": json.dumps({"ok": len(event["Records"])})}
```

2. The two lines creating `ddb` and `table` are outside the handler on purpose. Code outside the handler runs once per execution environment, not once per event, so the connection is reused across warm invocations.
3. `event` describes what happened. `context` carries the request id and the remaining time in milliseconds.

4. `unquote_plus` is not decoration. S3 event keys are URL-encoded, and spaces arrive as plus signs. Forgetting this is one of the most common real bugs.
5. The function returns; the environment is kept warm for a while in case another event arrives; eventually it is destroyed.
6. Now the cost. Suppose 2 million invocations a month, 200 milliseconds each, at 512 MB.
7. Compute: 2,000,000 x 0.2 s x 0.5 GB = 200,000 GB-seconds. At 0.0000166667 US dollars per GB-second that is 3.33 US dollars.
8. Requests: 2,000,000 at 0.20 US dollars per million is 0.40 US dollars.
9. Total about 3.73 US dollars per month, before the free allowance of one million requests and 400,000 GB-seconds, which would cover most of it.
10. A single small EC2 instance running all month costs about 30 US dollars. For this workload serverless is roughly ten times cheaper.

43.12.4 PLAIN - what is really happening inside

1. A cold start has four phases. The platform first downloads your code package.
2. It then creates an isolated execution environment. On AWS this is a Firecracker micro virtual machine, which boots in roughly 125 milliseconds.
3. It then starts the language runtime: interpreter, virtual machine or process.
4. Finally it runs your initialization code, the part outside the handler, and only then calls your handler.
5. After the handler returns, the environment is frozen rather than destroyed. A later event thaws it, and only the handler runs. That is a warm start, typically a few milliseconds of overhead.
6. Environments are frozen for some minutes and then reclaimed. The exact time is not documented and has changed over the years.
7. This is why state is unreliable. A warm environment keeps its variables, so caching sometimes appears to work, which is more dangerous than if it never worked.
8. Concurrency is not threads. Each concurrent event gets its own environment, so your function does not need to be thread-safe, but it also cannot share memory with its siblings.

43.12.5 TECHNICAL - the engineer's version

1. AWS Lambda launched 13 November 2014, the first mainstream function service. Google Cloud Functions followed in 2016, Azure Functions in 2016.
2. Current published limits for Lambda: memory from 128 MB to 10,240 MB in 1 MB steps; maximum execution time 15 minutes; deployment package 50 MB zipped direct upload, 250 MB unzipped, or up to 10 GB as a container image; request and response payload 6 MB synchronous, 256 KB asynchronous; `/tmp` from 512 MB to 10 GB.
3. CPU is not configured separately. It is allocated in proportion to memory, with roughly one full virtual CPU at about 1,769 MB. Raising memory to make a function faster is a legitimate and frequently profitable tactic.
4. Pricing, x86, at the time of writing: 0.20 US dollars per million requests and 0.0000166667 US dollars per GB-second, with an always-free tier of one million requests and 400,000 GB-seconds per month. Provisioned concurrency costs 0.0000041667 US dollars per GB-second on top.
5. Measured cold start durations at the median, from public benchmarking:

Runtime	Typical cold start
Rust	150 to 300 ms
Python 3.12	200 to 300 ms
Node.js 20	200 to 400 ms
C# .NET 8	500 to 1,500 ms
Java 17, no SnapStart	800 to 2,000 ms

6. Cold start frequency depends entirely on traffic shape. At sustained high traffic it is well under 1 per cent of invocations. At a handful of requests per hour, 30 to 50 per cent of invocations are cold.
7. Two mitigations exist. Provisioned concurrency keeps environments warm at a price. SnapStart, introduced for Java in 2022 and later extended, takes a snapshot of the initialized environment and restores from it, cutting multi-second Java starts to a few hundred milliseconds.
8. The VPC penalty is historical. Before 2019, attaching a Lambda to a VPC added 10 to 15 seconds to a cold start because an elastic network interface was created per environment. AWS re-engineered this with shared interfaces, and the penalty is now typically under 50 milliseconds. Beware of old advice.
9. When serverless is right: spiky or unpredictable traffic; event processing glue between services; scheduled jobs; low steady volume where a server would sit idle; teams with no operations capacity.
10. When serverless is a trap: sustained high throughput, where per-request pricing exceeds a reserved instance by a wide margin; workloads needing more than 15 minutes; anything needing large in-memory state or a warm cache; latency-critical paths where a cold start is unacceptable; heavy database use, because a thousand concurrent functions open a thousand connections and exhaust the database, which is what RDS Proxy exists to fix.
11. Note also the debugging cost. Local reproduction is imperfect, stack traces cross service boundaries, and understanding a failure requires distributed tracing rather than a debugger.
12. Vendor lock-in is strongest here. The function body is portable. The triggers, permissions, event shapes and surrounding services are not.

43.12.6 WORDS - remember these

1. Serverless — you never see or pay for an idle machine — an execution model with per-invocation billing, automatic scaling and no capacity management.
2. Cold start — the slow first run — the latency of creating an execution environment, starting the runtime and running initialization code.
3. Warm start — reusing a paused environment — invocation of a thawed environment, adding only milliseconds of overhead.
4. Trigger — the thing that causes your code to run — an event source binding such as HTTP, object creation, queue message or schedule.
5. Concurrency limit — how many copies may run at once — an account or function quota beyond which invocations are throttled.
6. Provisioned concurrency — paying to keep some copies warm — pre-initialized environments billed continuously to eliminate cold starts.
7. Firecracker — the tiny virtual machine underneath — AWS's open-source micro-VM monitor, booting in about 125 milliseconds, used by Lambda and Fargate.

43.13 Storage in the cloud

43.13.1 PLAIN - in simple words

1. There are three kinds of cloud storage, and choosing the wrong one is a common and expensive mistake.
2. **Block storage** is a virtual hard disk. You attach it to one machine, and the machine puts a filesystem on it. It is fast and it belongs to one owner.
3. **File storage** is a shared drive. Many machines mount it at once and see the same folders. Slower, but shared.
4. **Object storage** is different in kind. There are no folders and no disk. You store a whole item under a name, and you get the whole item back by name, over the web.
5. Object storage cannot be partly edited. To change one byte you upload the whole object again.
6. In exchange it is cheap, effectively unlimited, and enormously durable.
7. Choose block for databases and operating system disks. Choose file when several machines genuinely need the same files. Choose object for uploads, backups, logs, images, video and anything a browser will download.
8. There is one more thing everyone learns painfully. Putting data in is usually free. Taking data out to the internet is not.

43.13.2 PLAIN - a picture in your head

1. Block storage is a blank notebook on your desk. You can rub out one word on page 40 and write another. Only you have it.
2. File storage is a shared filing cabinet in the office. Several people open drawers at once, and there are rules about who may write.
3. Object storage is a left-luggage counter. You hand over a sealed bag and get a ticket. Later you show the ticket and get the whole bag back.
4. You cannot ask the counter to change one shirt inside the bag. You take the bag, change the shirt, and hand back a new bag.
5. The counter is astonishingly cheap and never runs out of shelves.
6. But it charges you a small fee every time you carry a bag out of the building.

Where this comparison breaks: a left-luggage counter has one shelf per bag. Object storage keeps several copies of your bag in several buildings, which is why it survives a fire and a notebook does not. Also, the counter here is reachable by anyone in the world with the right ticket, which is a security property the analogy hides and which has caused a great many public data leaks.

43.13.3 PLAIN - a worked example

1. Real S3 list prices for the North Virginia region, August 2026, per gigabyte per month.

Storage class	Price per GB	Use for
Standard, first 50 TB	0.0230	Active data
Standard-IA	0.0125	Monthly access
One Zone-IA	0.0100	Recreatable data
Glacier Instant	0.0040	Quarterly access
Glacier Flexible	0.0036	Backups
Glacier Deep Archive	0.00099	Legal retention

2. Deep Archive is about 23 times cheaper than Standard. That is why lifecycle rules exist.
3. Requests cost separately: 0.005 US dollars per 1,000 PUT requests and 0.0004 per 1,000 GET requests in Standard. In Glacier Deep Archive a PUT costs 0.05 per 1,000, ten times more, because the class is designed for few, large objects.

4. Store 10 terabytes of logs. In Standard: 10,240 GB x 0.023 = 235.52 US dollars per month. In Deep Archive: 10,240 x 0.00099 = 10.14 US dollars.
5. Now the trap. You want to download those 10 terabytes to your own datacentre.
6. Data transfer out to the internet is billed at 0.09 US dollars per gigabyte for the first 10 TB each month, after 100 GB free.
7. 10,240 GB x 0.09 = 921.60 US dollars, minus the free 100 GB, so about 912 US dollars.
8. Read that again. Storing the data for a month cost 10 dollars. Reading it once cost 912. The exit costs 90 times the rent.
9. That asymmetry is the single most important commercial fact about cloud storage, and it is not an accident.

43.13.4 PLAIN - what is really happening inside

1. When you PUT an object, the service splits it, computes checksums, and writes copies to storage nodes in at least three separate availability zones before it answers "done".
2. That is why the write is slower than a local disk write and why the durability claim is so high.
3. Object names look like paths, but there are no directories. `2026/08/log.txt` is one flat key that happens to contain slashes, and the console draws fake folders from them.
4. A lifecycle policy is a background process the service runs on your bucket daily. It looks at object age and moves or deletes objects by rule.
5. Transitions are not free: each object moved costs a lifecycle transition request. Moving ten million tiny objects to Glacier can cost more than leaving them where they were.
6. Egress is metered at the edge of the provider's network. Traffic to the internet is charged; traffic to another service inside the same region is usually free; traffic between availability zones or regions has its own smaller rate.

43.13.5 TECHNICAL - the engineer's version

1. Consistency. S3 offered eventual consistency for overwrites and deletes until 1 December 2020, when AWS made it strongly read-after-write consistent for all operations, at no extra cost and with no performance penalty. A great deal of published advice about "eventual consistency workarounds" predates that date and is now wrong.
2. Strong consistency here means a successful PUT is immediately visible to a subsequent GET or LIST from any client. It does not provide transactions across objects, and there is still no locking beyond conditional writes.
3. Performance figures: at least 3,500 PUT, COPY, POST or DELETE and 5,500 GET or HEAD requests per second per partitioned prefix, scaling by adding prefixes. Maximum object size 5 TB; maximum single PUT 5 GB; multipart upload required above that, with up to 10,000 parts.
4. Block storage on AWS is EBS: gp3 at 0.08 US dollars per GB-month with 3,000 IOPS and 125 MB/s included, extra IOPS at 0.005 each per month; io2 at 0.125 per GB-month plus 0.065 per provisioned IOPS. Snapshots cost 0.05 per GB-month standard, 0.0125 archived, and are incremental.
5. gp2, the older class, costs 0.10 per GB-month and ties performance to size at 3 IOPS per GB. gp3 is cheaper and decouples the two. Migrating gp2 volumes to gp3 is one of the easiest genuine savings in AWS.
6. File storage is EFS on AWS, Azure Files, or Filestore on Google Cloud. EFS speaks NFSv4.1 and costs several times more per gigabyte than EBS, so use it only when sharing is a real requirement.
7. Lifecycle policy, expressed in the JSON that the API actually takes:

```
{"Rules": [{
  "ID": "logs-tiering",
  "Filter": {"Prefix": "logs/"},
  "Status": "Enabled",
  "Transitions": [
    {"Days": 30, "StorageClass": "STANDARD_IA"},
```

```
    {"Days": 90, "StorageClass": "GLACIER"},
    {"Days": 365, "StorageClass": "DEEP_ARCHIVE"}],
  "Expiration": {"Days": 2555}
}]}
```

8. Minimum storage durations are billed even if you delete early: 30 days for Standard-IA and One Zone-IA, 90 days for Glacier Flexible, 180 days for Deep Archive. Standard-IA and One Zone-IA also bill a minimum object size of 128 KB, so tiering many small files can increase the bill.
9. Retrieval from Deep Archive takes up to 12 hours in standard mode and costs per gigabyte on top. It is an archive, not cold storage you can browse.
10. Egress pricing tiers for data out to the internet from US and EU regions: first 100 GB per month free across all services, then 0.09 US dollars per GB to 10 TB, 0.085 to 50 TB, 0.07 to 150 TB, and 0.05 above that. Cross- availability-zone traffic is 0.01 per GB in each direction, and NAT gateway processing adds 0.045 per GB on top of everything else.
11. Two changes have eroded the egress lock. From 5 March 2024 AWS offers free data transfer out for customers who are leaving AWS entirely, with 90 days to complete the move and a support request required; other major providers made similar announcements. And the European Union's Data Act, Regulation 2023/2854, requires switching charges including egress fees to be fully withdrawn from 12 January 2027.
12. Cloudflare R2 and several second-tier providers charge no egress at all, which is a deliberate competitive attack on this pricing model.

43.13.6 WORDS - remember these

1. Object storage — whole items stored and fetched by name — a flat key-value store over HTTP, immutable per object, replicated across zones.
2. Block storage — a virtual hard disk for one machine — network-attached block device presented as raw sectors, formatted by the guest.
3. File storage — a shared network drive — a managed NFS or SMB filesystem mountable by many clients at once.
4. Storage class — how cheap and how slow — a tier trading price per gigabyte against retrieval time, minimum duration and request cost.
5. Lifecycle policy — automatic tiering and deletion by age — bucket rules that transition or expire objects on a daily background pass.
6. Egress — the charge for data leaving — per-gigabyte data transfer out to the internet, typically 0.09 US dollars per GB at the first tier.

43.14 Cloud networking

43.14.1 PLAIN - in simple words

1. A **VPC**, virtual private cloud, is your own private network inside the provider's network. You choose the address range and the layout.
2. Inside it you create **subnets**, which are slices of that address range, each pinned to one availability zone.
3. A subnet is called **public** if its route table sends unknown traffic to an internet gateway, and **private** if it does not. That is the only difference.
4. An **internet gateway** lets traffic in and out. A **NAT gateway** lets private machines reach out to the internet while nothing can reach in.
5. **Security groups** are firewalls attached to individual machines. They allow traffic; they cannot deny it, and they remember connections.
6. **Network ACLs** are firewalls attached to whole subnets. They can allow and deny, and they remember nothing.

7. **Load balancers** sit in front of your machines and spread traffic. Layer 4 forwards connections without reading them. Layer 7 reads the request and can route by hostname or path.
8. **Private endpoints** let you reach a provider service without your traffic going over the internet at all.
9. **Peering** joins two networks. A **transit gateway** joins many, in a hub pattern, instead of one link per pair.

43.14.2 PLAIN – a picture in your head

1. Think of a gated business park that you have rented and laid out.
2. The park boundary is the VPC. The address range is the numbering scheme for all the plots.
3. Subnets are the blocks of plots. Two blocks are in one part of the park and two in another, on separate power and water.
4. The main gate is the internet gateway.
5. A NAT gateway is a courier desk. Staff in the inner blocks can send parcels out through it and get replies, but nobody outside can address a parcel to them directly.
6. Security groups are the locks on each individual building's doors.
7. Network ACLs are the barriers at the entrance to each block, checking every vehicle in both directions with no memory of who came in.
8. A load balancer is the receptionist at the front, directing arrivals to whichever office is open.

Where this comparison breaks: in a real park a delivery van drives itself. Here, every packet's path is decided by a route table you wrote, and if no route matches, the packet is silently dropped. There is no lost-property office and no error message. Silence is the normal failure mode, which is why the first thing to check in any cloud networking problem is the route table.

43.14.3 PLAIN – a worked example

1. A standard three-tier VPC, spread over two availability zones.

```
VPC 10.0.0.0/16   Region: ap-south-1
+-------------------------------------------------------+
|   AZ ap-south-1a          |   AZ ap-south-1b          |
|                           |                           |
| PUBLIC 10.0.0.0/24        | PUBLIC 10.0.1.0/24        |
|  [ALB node]  [NAT gw]     |  [ALB node]  [NAT gw]     |
|      |            ^       |      |           ^        |
|      v            |       |      v           |        |
| APP 10.0.10.0/24          | APP 10.0.11.0/24          |
|  [web ec2] [web ec2] -----+  [web ec2] [web ec2]      |
|      |                    |      |                    |
|      v                    |      v                    |
| DATA 10.0.20.0/24         | DATA 10.0.21.0/24         |
|  [RDS primary]            |  [RDS standby]            |
+-------------------------------------------------------+
       ^
       |  Internet gateway  <-> the internet
```

2. Address maths. `10.0.0.0/16` gives 65,536 addresses. Each `/24` gives 256, of which AWS reserves 5, leaving 251 usable.
3. Route table for the public subnets: `10.0.0.0/16` local, `0.0.0.0/0` to the internet gateway.
4. Route table for the application subnets: `10.0.0.0/16` local, `0.0.0.0/0` to the NAT gateway in the same availability zone.
5. Route table for the data subnets: `10.0.0.0/16` local, and nothing else. The database cannot reach the internet at all, which is deliberate.
6. Security group chain: the load balancer group allows 443 from anywhere; the web group allows 8000 only from the load balancer's group; the database group allows 5432 only from the web group.

7. Referencing a group rather than an address range is the important habit. It keeps working when instances are replaced.
8. One NAT gateway per availability zone, not one shared, because a shared NAT gateway makes the other zone depend on the first, defeating the whole design.

43.14.4 PLAIN - what is really happening inside

1. None of this is physical. There is no cable and no real router.
2. Every packet leaving an instance is intercepted by the host's virtualization hardware, which looks up your VPC's rules and rewrites the packet into an encapsulated form.
3. That encapsulated packet crosses the provider's physical network as ordinary traffic, is decapsulated at the destination host, and delivered.
4. Your route tables and security groups are entries in a distributed database that is pushed to every host that needs them.
5. On AWS this mapping work happens on the Nitro card, not on the CPU, which is why a security group costs almost nothing in performance.
6. Because rules are enforced at the source host, a denied packet never reaches the network. There is no device to log it, which is why denied traffic is invisible unless you enable flow logs.

43.14.5 TECHNICAL - the engineer's version

1. Address planning uses RFC 1918 private ranges: `10.0.0.0/8`, `172.16.0.0/12` and `192.168.0.0/16`. AWS VPC CIDR blocks may be `/16` to `/28`. Overlapping ranges between VPCs make peering impossible later, so allocate deliberately from day one.
2. Security groups versus network ACLs, the exam question and the real one:

Property	Security group	Network ACL
Attaches to	Instance interface	Subnet
Rules	Allow only	Allow and deny
State	Stateful	Stateless
Evaluation	All rules, any match	Numbered, first match

3. **Stateful** means the return traffic of an allowed outbound connection is automatically permitted. **Stateless** means you must write both directions, including the ephemeral port range 1024 to 65535 for replies. This is the same distinction drawn in Chapter 34 for ordinary firewalls, and it is the most common source of network ACL mistakes.
4. Load balancers by layer. A network load balancer operates at layer 4, forwards TCP or UDP flows, preserves the client address, handles millions of connections per second and adds tens of microseconds. An application load balancer operates at layer 7, terminates TLS, understands HTTP, and can route by host header, path, method or header value, at the cost of about one millisecond and a more complex failure surface.
5. Private connectivity to services: gateway endpoints, which are route table entries for S3 and DynamoDB and are free; and interface endpoints, which are PrivateLink elastic network interfaces in your subnets, billed per hour and per gigabyte. Using a gateway endpoint for S3 removes both NAT gateway processing charges and the internet path.
6. VPC peering is one-to-one, non-transitive, and requires non-overlapping CIDR blocks. If A peers with B and B peers with C, A cannot reach C. That limitation is exactly why AWS Transit Gateway exists: a hub that all VPCs attach to, giving transitive routing, at a per-attachment hourly charge plus per-gigabyte processing.
7. Hybrid connectivity: site-to-site VPN over the internet using IPsec, typically 1.25 Gbps per tunnel and subject to internet variability; or Direct Connect, a dedicated physical circuit from 50 Mbps to 100 Gbps

with predictable latency and lower egress rates. Azure calls the equivalent ExpressRoute; Google calls it Cloud Interconnect.

8. Observation: VPC Flow Logs record accepted and rejected flows to CloudWatch or S3. Reachability Analyzer traces a path through your configuration and tells you which rule blocked it, without sending a packet. `aws ec2 describe-route-tables` and `aws ec2 describe-security-groups` are the two commands most often needed at 3am.

43.14.6 WORDS - remember these

1. VPC — your own private network in the provider — a logically isolated virtual network with tenant-defined CIDR, subnets and routing.
2. Subnet — a slice of that network in one zone — a CIDR range bound to a single availability zone with one associated route table.
3. Internet gateway — the way in and out — a horizontally scaled VPC component providing a route to and from the public internet.
4. NAT gateway — outbound only, for private machines — a managed network address translation service, billed hourly and per gigabyte.
5. Security group — a stateful allow-list on one machine — instance-level rules permitting traffic, with automatic return-path allowance.
6. Network ACL — a stateless filter on a whole subnet — numbered allow and deny rules evaluated in order, requiring explicit return rules.
7. Transit gateway — one hub instead of many pairwise links — a regional router providing transitive connectivity between many VPCs and on-premises networks.

43.15 Identity and access

43.15.1 PLAIN - in simple words

1. Every action in a cloud account is an API call, and every API call is checked against rules before it runs.
2. The system holding those rules is identity and access management, IAM.
3. There are two kinds of actor. A **user** is a person with a long-lived password or key. A **role** is a set of permissions that anything can borrow temporarily.
4. A **policy** is a document listing which actions are allowed on which resources, and under which conditions.
5. The single most important habit is **least privilege**: grant exactly what is needed and nothing more, then review it.
6. The second most important habit is to stop using long-lived keys.
7. A long-lived key is a password that never expires, is often copied into a config file, and frequently ends up in a code repository.
8. A role gives out temporary credentials that expire in an hour. If they leak, the window is small.
9. Machines should never hold permanent keys. A server gets a role attached to it, and the credentials appear automatically and rotate themselves.

43.15.2 PLAIN - a picture in your head

1. Think of a hospital again.
2. A user is a staff member with a permanent badge that opens certain doors.
3. A role is a coat hanging by the door of the operating theatre. Anyone authorized may put it on, and while wearing it they can do surgery.
4. The coat is returned. It cannot be taken home, and it stops working at the end of the shift.
5. A policy is the written list of which doors that coat opens.
6. Least privilege means the surgery coat does not also open the pharmacy safe, even though it would occasionally be convenient.

7. The commonest real-world failure is a coat that opens everything, made once during an emergency, and never taken out of circulation.

Where this comparison breaks: a coat is physical, so only one person wears it at a time. Temporary credentials can be copied and used from anywhere in the world until they expire. That is why conditions such as source address, multi-factor authentication and session duration matter as much as the permission list.

43.15.3 PLAIN – a worked example

1. A policy allowing an application to read one bucket and nothing else.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "ReadUploadsOnly",
    "Effect": "Allow",
    "Action": ["s3:GetObject", "s3:ListBucket"],
    "Resource": [
      "arn:aws:s3:::festival-uploads",
      "arn:aws:s3:::festival-uploads/*"],
    "Condition": {
      "Bool": {"aws:SecureTransport": "true"}
    }
  }]
}
```

2. Two resource lines are needed because `ListBucket` acts on the bucket and `GetObject` acts on objects inside it. Forgetting this is a daily support question.
3. The condition rejects any request that did not arrive over TLS.
4. `Version` is a fixed policy language date, not your version number. It must be `2012-10-17`.
5. Attach this policy to a role, attach the role to the instance, and the application never sees a key.
6. Compare with the wildcard policy that people write when they are tired: `"Action": "s3:*", "Resource": "*"`. That grants delete on every bucket in the account, including the backups.

43.15.4 PLAIN – what is really happening inside

1. Policy evaluation follows a fixed order, and knowing it saves hours.
2. Start from an implicit deny. Nothing is allowed by default.
3. Evaluate all applicable policies: identity policies on the caller, resource policies on the target, permission boundaries, service control policies from the organization, and session policies.
4. If any of them contains an explicit deny that matches, the answer is deny. Nothing can override an explicit deny.
5. Otherwise, if at least one allow matches and no boundary excludes it, the answer is allow.
6. Otherwise the answer is deny by default.
7. In one line: explicit deny beats allow, and allow beats the default deny.
8. Temporary credentials work through a token service. An entity calls “assume this role”, proving its own identity; the service checks the role’s trust policy; if satisfied it returns an access key, a secret key and a session token with an expiry.
9. On an EC2 instance a metadata service delivers those credentials to software on the machine and refreshes them before expiry. On Kubernetes, a service account token is exchanged for the same thing.

43.15.5 TECHNICAL – the engineer’s version

1. Policy types and their interaction: identity-based policies attach to users, groups and roles; resource-based policies attach to buckets, queues and keys and can grant cross-account access without a role; permission boundaries cap the maximum a principal can be granted; service control policies cap an entire account within an organization; session policies narrow one session.

2. A principal's effective permission is the intersection of all caps and the union of all grants, minus every explicit deny.
3. Instance metadata: IMDSv1 was a plain unauthenticated HTTP GET to `169.254.169.254`. IMDSv2 requires a PUT to obtain a session token first and sets a low IP hop limit, which defeats the classic server-side request forgery path. AWS began defaulting new instances to IMDSv2 in 2024. Enforce it explicitly on older infrastructure.
4. That specific weakness produced the best-known cloud breach. In March 2019 an attacker exploited a server-side request forgery flaw in a misconfigured web application firewall at Capital One, reached the metadata service, obtained the credentials of an over-permissive role, and used them to list and copy S3 buckets containing roughly 106 million customer records. Regulators fined the bank 80 million US dollars in 2020 and a class action settled at 190 million.
5. Note that no AWS component failed. Every step used a documented feature. The failure was a firewall misconfiguration plus a role with far more S3 access than the application needed.
6. The two dominant causes of cloud compromise remain over-permissive policies and leaked long-lived keys. Keys leak through public repositories, container images, CI logs, browser bundles and screenshots. Automated scanners find a newly committed AWS key in public GitHub within minutes.
7. Mitigations in order of value: use roles and short-lived credentials everywhere; federate human access through single sign-on so no human has an access key at all; enable multi-factor authentication on every account with console access; turn on GitHub push protection and secret scanning; set an organization-wide service control policy denying dangerous actions outright.
8. Right-sizing tools: IAM Access Analyzer generates a least-privilege policy from CloudTrail history; last-accessed data shows which services a principal has actually used; Access Advisor exposes the same for roles.
9. Naming, for cross-provider fluency. AWS uses roles and JSON policies. Azure uses Entra ID principals with role-based access control, where a role assignment binds a role definition to a scope. Google Cloud binds members to predefined or custom roles at a resource in the hierarchy, and its default Compute Engine service account is famously over-permissive and should be replaced.
10. Diagnostic commands: `aws sts get-caller-identity` to learn who you are; `aws iam simulate-principal-policy` to test a decision before deploying it; CloudTrail event history to find the exact denied call and its reason.

43.15.6 WORDS - remember these

1. IAM — the rules about who may do what — identity and access management, evaluated on every API call against identity and resource policies.
2. Role — a borrowable set of permissions — an identity with no long-term credentials, assumed to obtain time-limited session credentials.
3. Policy — the document listing allowed actions — a JSON statement set of effect, action, resource and condition.
4. Least privilege — grant only what is needed — the principle of minimizing granted permissions to those actually exercised.
5. Explicit deny — a refusal nothing can override — the highest-precedence outcome in policy evaluation, above any allow.
6. Instance metadata service — where a machine gets its own credentials — the link-local endpoint at 169.254.169.254, hardened as IMDSv2.
7. Service control policy — an account-wide ceiling — an organization policy limiting the maximum permissions available in member accounts.

43.16 What cloud costs and how bills explode

43.16.1 PLAIN – in simple words

1. A cloud bill is not one price. It is dozens of meters running at once.
2. The usual dimensions are: time a machine existed, storage held per month, number of requests, gigabytes moved, and per-hour charges for things that feel free, such as addresses and gateways.
3. Three ways to pay for compute. **On demand** is the list price, no commitment. **Reserved**, or savings plans, means promising one or three years of spending for a discount of roughly 30 to 72 per cent. **Spot** means using spare capacity at up to 90 per cent off, on the condition it can be taken away with two minutes' notice.
4. Bills explode for boring reasons, almost never exciting ones.
5. Something was left running. Somebody chose a machine ten times bigger than needed. Logs were kept forever. Data crossed a boundary that charges.
6. The free tier is genuinely useful and genuinely a trap: it hides which resources cost money until the day it expires.
7. The honest headline: at steady, predictable load, owning hardware is usually cheaper than renting it. Cloud wins on speed, elasticity and not needing a team, not on unit price.

43.16.2 PLAIN – a picture in your head

1. Think of a mobile phone bill from before unlimited plans.
2. The handset rental is the machine. The data pack is the storage. Calls are requests. Roaming is egress.
3. Everyone understood the handset charge. Everyone was shocked by roaming.
4. Roaming had three properties that made it dangerous: it was per unit, it was invisible while it happened, and it was triggered by crossing a boundary nobody was thinking about.
5. Cloud egress and cross-zone traffic are exactly that. The boundary is invisible in your code and expensive in your bill.
6. And there is one extra hazard the phone bill did not have: your program can generate charges by itself, all night, with nobody watching.

Where this comparison breaks: a phone bill has a ceiling because a human can only talk so much. A misconfigured job can generate a five-figure bill in hours, and there is no hard spending limit you can set on the major clouds. Budgets alert you; they do not stop you.

43.16.3 PLAIN – a worked example

1. A real monthly bill for a modest production application in North Virginia, using the list prices quoted earlier in this chapter. Two web servers across two zones, a managed database with a standby, a load balancer, storage and 500 GB of traffic to users.

Item	Calculation	USD
2 x t3.medium, 730 h	2x730x0.0416	60.74
2 x 30 GB gp3 root	60x0.08	4.80
RDS db.t4g.medium, MAZ	730x0.129	94.17
RDS storage 100 GB gp3	100x0.115	11.50
ALB, base + ~8 LCU	730x(0.0225+0.064)	63.15
NAT gateway, 1 zone	730x0.045	32.85
NAT data, 200 GB	200x0.045	9.00
S3 200 GB Standard	200x0.023	4.60
Egress 500 GB, 100 free	400x0.09	36.00
Route 53 zone + queries	0.50 + ~0.40	0.90
Secrets Manager, 4	4x0.40	1.60
CloudWatch logs 20 GB	20x0.50	10.00
Total		329.31

Item	Calculation	USD

2. Now read the bill rather than the total. The two web servers, the thing everybody thinks of as "the cost", are 18 per cent of it.
3. The database costs more than the servers. The load balancer costs as much as the servers. The NAT gateway costs 42 dollars to forward packets.
4. Egress of half a terabyte, which is a small website, costs more than the object storage holding everything.
5. Savings available without changing the architecture: a one-year compute savings plan takes about 30 per cent off the EC2 and RDS lines, roughly 47 dollars. A gateway endpoint for S3 removes much of the NAT data charge. Log retention set to 14 days cuts CloudWatch.
6. For comparison, a dedicated server of similar capability from a low-cost European host costs about 40 to 70 US dollars per month including generous traffic, but with no managed database, no standby, no load balancer and no support.

43.16.4 PLAIN - what is really happening inside

1. Every service writes usage records continuously, usually per hour, sometimes per second, into a billing pipeline.
2. Those records land in a detailed usage report with one row per resource, per usage type, per hour. The monthly invoice is a summary of tens of millions of rows.
3. This is why bills are hard to read: the invoice groups by service, but costs are caused by resources and by teams.
4. Tags are the only mechanism that connects the two. An untagged resource is effectively anonymous spending.
5. Charges accrue whether or not anything useful happens. A stopped EC2 instance still bills its EBS volume, its snapshots and its allocated public address.
6. Since February 2024 AWS charges for every public IPv4 address, in use or not, at 0.005 US dollars per hour, about 3.60 per month each. Many organizations discovered hundreds of forgotten addresses.

43.16.5 TECHNICAL - the engineer's version

1. Commitment mechanisms: Reserved Instances lock instance family and region; Savings Plans commit to a dollar per hour of spend and are more flexible. Discounts run roughly 30 to 40 per cent for one year with no upfront payment, and up to 72 per cent for three years paid fully in advance.
2. Spot instances draw on spare capacity at typically 70 to 90 per cent off, with a two-minute interruption notice. Correct uses are stateless web tiers behind a load balancer, batch processing, continuous integration runners and fault-tolerant analytics. Incorrect uses are databases and anything holding state a customer cares about.
3. Free tier changed materially on 15 July 2025. Accounts created from that date receive 100 US dollars of credit at sign-up and up to 100 more for completing activities, on a plan that ends after six months or when credits run out. The old twelve-month free tier applies only to accounts created before that date. Always-free allowances, such as one million Lambda requests per month, continue under both schemes.
4. The classic explosion patterns, in the order they actually occur: a development environment left running over a holiday; log retention set to never expire; a NAT gateway processing data that should have used an endpoint; cross-availability-zone chatter between services that could have been zone-aware; oversized instances chosen from a template; unattached EBS volumes and old snapshots; and a recursive trigger, where a function writes to the bucket that triggers it.
5. **FinOps** is the discipline that grew up around this. The FinOps Foundation was formed in 2019 and moved under the Linux Foundation in 2020. Its model is three iterative phases: inform, meaning visibility and allocation; optimize, meaning rightsizing and commitments; and operate, meaning continuous governance.

6. Practical controls, in order of return on effort: mandatory cost allocation tags enforced by policy; AWS Budgets with alerts at 50, 80 and 100 per cent; automatic shutdown schedules for non-production; Compute Optimizer or equivalent for rightsizing; S3 lifecycle rules and log retention limits; gateway endpoints for S3 and DynamoDB; and one named person who reviews the top ten line items every month.
7. Now the honest comparison, which the industry avoided for a decade.
8. A single dedicated server bought outright, amortized over five years, is typically two to five times cheaper per unit of compute than the equivalent on-demand cloud instance, before staff costs.
9. The best-documented case is 37signals, the company behind Basecamp. It spent 3,201,564 US dollars on cloud in 2022, moved seven applications off AWS during 2023 onto about 700,000 US dollars of Dell hardware, and reported savings of nearly 2 million US dollars in the first full year, with total projected savings above 10 million over five years. It then moved roughly 10 petabytes off S3 as well.
10. Dropbox is the other well-known case. It moved the bulk of its storage off AWS onto its own infrastructure, a project called Magic Pocket, and its 2018 filing reported 74.6 million US dollars of infrastructure savings across 2016 and 2017.
11. Both are the same shape: large, steady, predictable workloads, with strong in-house engineering, and a bill dominated by storage and bandwidth.
12. When cloud is genuinely the better economic choice: unpredictable or seasonal load; early-stage products where the requirement is unknown; small teams with nobody to run hardware; workloads needing global reach quickly; and anything where time to market dominates unit cost.
13. When owning is better: steady load at scale; heavy egress; large storage volumes; regulatory requirements that push to dedicated hardware anyway; and organizations that already employ infrastructure staff.
14. The reasonable middle position, and the one most large companies have reached by 2026, is a split: steady baseline on owned or long-committed capacity, burst and experimentation in the cloud.

43.16.6 WORDS - remember these

1. On-demand pricing — full price, no promises — per-second or per-hour billing with no commitment and no discount.
2. Reserved and savings plans — a discount for a promise — one or three year commitments giving roughly 30 to 72 per cent off.
3. Spot — very cheap, can be taken away — interruptible spare capacity at 70 to 90 per cent off with a two-minute notice.
4. Egress — the charge for data leaving — per-gigabyte data transfer out, the most common source of bill surprise.
5. Idle spend — paying for what nobody uses — unattached volumes, stopped instances still holding storage, unused addresses and forgotten environments.
6. FinOps — treating cloud spend as an engineering problem — the inform, optimize and operate practice defined by the FinOps Foundation from 2019.
7. Repatriation — moving workloads back off cloud — relocating steady, high volume workloads to owned hardware for cost reasons.

43.17 DevOps, infrastructure as code and the modern workflow

43.17.1 PLAIN - in simple words

1. For decades, two groups fought. Developers were paid to ship changes. Operators were paid to keep things stable. Change is the enemy of stability, so they were structurally opposed.
2. **DevOps** is the idea of removing that opposition: one team, shared goals, shared responsibility for both shipping and running.
3. It is a way of organizing people. It is not a tool, not a job title and not a department, although the industry turned it into all three.
4. The technical half is **infrastructure as code**: instead of clicking in a web console, you write a file describing what should exist, and a tool makes reality match it.
5. That file goes into version control, gets reviewed, and can rebuild the entire environment from nothing.
6. **Immutable infrastructure** takes it further: never patch a running machine. Build a new one from a new image and throw the old one away.
7. **GitOps** takes it further still: the repository is the only source of truth, and an agent inside the cluster continuously pulls from it and corrects drift.
8. **Observability** is being able to answer questions about a running system that you did not think to ask in advance. It needs logs, metrics and traces together, because each answers a different question.
9. **Site reliability engineering** is the discipline of deciding, numerically, how reliable a service must be, and spending the remaining unreliability deliberately on shipping features.

43.17.2 PLAIN - a picture in your head

1. Think of two ways to furnish a flat.
2. Way one: over three years you buy things, move them, and drill holes. It looks fine. Now furnish an identical flat next door exactly the same way. You cannot. Nobody wrote anything down.
3. Way two: you keep a plan on one sheet. Every change is drawn on the plan first, then done. Furnishing the identical flat takes an afternoon.
4. Infrastructure as code is the second way.
5. Immutable infrastructure is the rule that you never repaint a wall. You build a new wall to the new plan and remove the old one.
6. GitOps adds a caretaker who compares the flat to the plan every few minutes and undoes anything that does not match, including your own quick fix.

Where this comparison breaks: a flat has no state. A database has state, and you cannot throw it away and build a new one. The immutable rule applies cleanly to servers that hold no data, and needs careful exceptions for anything that does. That distinction, stateless versus stateful, is the whole reason people speak of cattle and pets.

43.17.3 PLAIN - a worked example

1. The same load balancer and instance, written in Terraform.

```
resource "aws_security_group" "web" {
  name   = "festival-web"
  vpc_id = aws_vpc.main.id

  ingress {
    from_port       = 8000
    to_port         = 8000
    protocol        = "tcp"
    security_groups = [aws_security_group.alb.id]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
```

```
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_instance" "web" {
  count                  = 2
  ami                    = var.app_ami_id
  instance_type          = "t3.medium"
  subnet_id              = aws_subnet.app[count.index].id
  vpc_security_group_ids = [aws_security_group.web.id]
  iam_instance_profile   = aws_iam_instance_profile.web.name

  tags = {
    Name = "festival-web-${count.index}"
    Env  = "prod"
    Cost = "festival"
  }
}
```

2. `aws_security_group.alb.id` is a reference, not a copied value. Terraform builds a dependency graph from those references and orders creation itself.
3. `count = 2` creates two instances. Changing it to four is a one-character edit and a reviewable change.
4. `var.app_ami_id` points at a machine image built in advance. Changing that variable replaces both instances, which is immutable infrastructure in practice.
5. The tags are not decoration. They are how the monthly bill gets attributed.
6. The workflow:

```
terraform init
terraform plan -out=tf.plan
terraform apply tf.plan
terraform state list
terraform destroy
```

7. `plan` is the important command. It prints exactly what will be created, changed or destroyed, before anything happens. Reviewing a plan is the equivalent of reviewing a diff.

43.17.4 PLAIN - what is really happening inside

1. Terraform keeps a **state file**: a record of which real resources correspond to which blocks in your code.
2. On `plan` it does three reads: your configuration, the state file, and the actual cloud API. It then computes the difference.
3. That is why the state file matters so much. Lose it and Terraform no longer knows it owns anything, and will try to create everything again.
4. In any team the state lives remotely, in S3 or an equivalent, with locking so two engineers cannot apply at once.
5. GitOps inverts the direction of control. Instead of a pipeline pushing changes into the cluster, an agent inside the cluster pulls the repository and applies it continuously.
6. That means the cluster needs no inbound credentials from the pipeline, and any manual change made by hand is reverted within minutes.
7. Observability data flows separately. Applications emit structured logs, numeric metrics and trace spans; agents collect them; backends store and index them; dashboards and alerts read them.

43.17.5 TECHNICAL - the engineer's version

1. Origins. The word DevOps comes from DevOpsDays, organized by Patrick Debois in Ghent, Belgium, in October 2009. The cultural model is often summarized as CALMS: culture, automation, lean, measurement, sharing.

2. Tooling lineage for infrastructure as code: CFEngine 1993, Puppet 2005, Chef 2009, SaltStack 2011, Ansible 2012. Those are configuration management, which converges the inside of an existing machine.
3. Provisioning tools create the machines and the surrounding cloud resources: AWS CloudFormation from February 2011, Terraform from HashiCorp in 2014, Azure Resource Manager templates and the friendlier Bicep, and Pulumi from 2018, which uses ordinary programming languages instead of a domain-specific one.
4. A licensing event worth knowing: HashiCorp changed Terraform's licence from Mozilla Public License to the Business Source License in August 2023. The community forked it as OpenTofu, adopted by the Linux Foundation in September 2023, with version 1.0 in January 2024. If you are choosing today, this is a live consideration.

Tool	Language	Scope
CloudFormation	YAML or JSON	AWS only
Terraform, OpenTofu	HCL	Multi-provider
Pulumi	Python, TS, Go	Multi-provider
Ansible	YAML	Config management

5. Immutable infrastructure practice: bake an image with Packer or an equivalent, version it, deploy by replacing instances, and forbid interactive login to production. The phrase "cattle, not pets" comes from a 2012 talk by Bill Baker of Microsoft, popularized by Randy Bias.
6. GitOps was named by Alexis Richardson of Weaveworks in 2017. The CNCF OpenGitOps project states four principles: the desired state is declarative; it is versioned and immutable; agents pull it automatically; and agents continuously reconcile actual to desired. Argo CD and Flux are the two main implementations.
7. Observability, the three signals, and why you need all three. Metrics are cheap numbers over time and tell you that something is wrong. Logs are detailed records of individual events and tell you what happened in one case. Traces follow a single request across service boundaries and tell you where the time went and which service failed.
8. Metrics alone cannot tell you which user. Logs alone cannot show a system-wide pattern or a cross-service path. Traces alone are usually sampled and cannot be the record of truth. The three are complementary, not alternatives.
9. OpenTelemetry, formed in 2019 from the merger of OpenTracing and OpenCensus under the CNCF, is now the vendor-neutral standard for emitting all three. The wire format is OTLP.
10. Site reliability engineering began at Google in 2003 under Ben Treynor Sloss and was published as the book Site Reliability Engineering in 2016. Its core vocabulary:
11. An **SLI**, service level indicator, is a measured number, such as the proportion of requests served successfully in under 300 milliseconds.
12. An **SLO**, service level objective, is your internal target for that indicator, such as 99.9 per cent over 28 days.
13. An **SLA** is the external contract, always weaker than the objective, with financial consequences.
14. An **error budget** is the allowed failure: 100 per cent minus the objective. At 99.9 per cent over 30 days it is about 43 minutes of failure.
15. The budget is the point. If it is unspent, you are being too cautious and should ship faster. If it is exhausted, feature work stops until reliability is restored. It converts an argument about risk into arithmetic.
16. Blameless postmortems, formalized at Etsy by John Allspaw and colleagues from around 2012, assume that people act reasonably given the information they had, and look for the systemic conditions that made the error possible. Naming an individual ends learning, because the next person hides the near miss.

17. On-call practice that is defensible: a rota no thinner than one week in six, documented runbooks for every alert, alerts that are actionable or deleted, compensation or time off for nights worked, and a rule that a page nobody acted on gets removed rather than ignored.

43.17.6 WORDS - remember these

1. DevOps — one team owning both shipping and running — a cultural model removing the structural conflict between change and stability, from 2009.
2. Infrastructure as code — the environment written down as a file — declarative definitions applied by a tool, versioned and reviewed like source code.
3. State file — the tool's record of what it owns — Terraform's mapping between configuration blocks and real resource identifiers, stored remotely with locking.
4. Immutable infrastructure — replace, never patch — deploying new pre-built images instead of modifying running machines.
5. GitOps — the repository is the truth and an agent enforces it — declarative state pulled and continuously reconciled from version control.
6. SLI, SLO, SLA — the measurement, the internal target, the external contract — the three levels of reliability commitment in site reliability engineering.
7. Error budget — the failure you are allowed to spend — 100 per cent minus the objective, used to arbitrate between shipping and stabilizing.
8. Observability — being able to ask new questions of a live system — the combination of metrics, logs and traces, standardized by OpenTelemetry.

43.18 Microservices versus monolith, decided honestly

43.18.1 PLAIN - in simple words

1. A **monolith** is one program containing all your features, built and deployed as one unit.
2. **Microservices** means splitting those features into many small programs that talk over the network and deploy independently.
3. The genuine benefit is independence. Teams ship without coordinating, and one service can be rewritten or scaled without touching the rest.
4. The genuine cost is that every internal function call you replaced with a network call can now be slow, fail, arrive twice or vanish.
5. A method call cannot half-succeed. A network call can.
6. So the question is never "which is better". It is "do I have enough teams for the independence to be worth the failure handling".
7. For a team of five, the answer is almost always no.
8. The recommendation, which is now mainstream rather than contrarian: start with a **modular monolith**. One deployable unit, with strict internal boundaries and no reaching across them.
9. Then, if a module genuinely needs to scale differently, be written in another language, or be owned by another team, extract that one module.
10. You get the mandate's discipline about boundaries without paying the network bill until you must.

43.18.2 PLAIN - a picture in your head

1. Picture a restaurant kitchen with one large team in one room.
2. Anyone can shout to anyone. Coordination is instant and free. The room does not scale past a certain number of cooks.
3. Now picture five separate kitchens in five buildings, each producing one part of the meal, connected by couriers.
4. Each kitchen can be rebuilt, restaffed or expanded without closing the others.

5. But every request is now a courier trip. Couriers get stuck in traffic. Sometimes a courier arrives and the dish has already been sent again.
6. A modular monolith is one building with five clearly separated stations, fixed hatches between them, and a rule that nobody walks into another station's area.
7. If the dessert station later needs its own building, it already has a defined hatch to replace with a courier route.

Where this comparison breaks: kitchens are physically fixed, so the analogy suggests the split is permanent and expensive. In software the split is reversible in principle, and several well-known organizations have merged services back. It also understates the failure modes. A courier who never arrives is obvious. A network call that succeeded but whose reply was lost looks exactly like one that failed, and that ambiguity is the hardest problem in the whole subject.

43.18.3 PLAIN - a worked example

1. One user action, "place order", in both designs.
2. Monolith: one process calls `reserve_stock()`, `charge_card()` and `create_shipment()` inside a single database transaction. If the card fails, the transaction rolls back and no stock was reserved. Total time about 40 milliseconds. The failure model is one exception.
3. Microservices: the order service calls the inventory service, then the payment service, then the shipping service, each over the network.
4. There is no shared transaction. If payment fails after stock was reserved, the reservation must be actively undone by a **compensating action**.
5. That pattern is called a **saga**: a sequence of local transactions where each step has a defined undo.
6. Now the hard case. The order service calls payment and the reply is lost. Was the card charged? Unknown.
7. If it retries blindly, it may charge twice. So payment must accept an **idempotency key** and treat a repeat of the same key as the same charge.
8. Timing: three network calls at roughly 2 milliseconds each, plus serialization at both ends, plus retries, gives something like 15 to 40 milliseconds instead of 40 microseconds of function calls.

Aspect	Monolith	Microservices
Call cost	Nanoseconds	Milliseconds
Failure model	Exception	Timeout, dupe, loss
Consistency	Transaction	Saga, eventual
Deploy unit	One	Many, independent

43.18.4 PLAIN - what is really happening inside

1. Once a request crosses services, no single log file contains the story.
2. **Distributed tracing** fixes this. The first service creates a trace identifier and passes it to everything it calls, in an HTTP header.
3. Each service records a **span**: its own start time, duration, name, and the identifier of the span that caused it.
4. A backend collects the spans and reassembles them into one tree by trace identifier, showing exactly where the time went.
5. Without tracing, debugging a slow request across eight services means guessing and correlating timestamps by hand.
6. Eventual consistency means different services briefly hold different answers. The order exists but the shipment does not yet. Your user interface must be designed for that, not surprised by it.

43.18.5 TECHNICAL - the engineer's version

1. The **fallacies of distributed computing** are the checklist. L. Peter Deutsch wrote seven at Sun Microsystems around 1994; James Gosling added the eighth. Each is something engineers assume without noticing.
2. The network is reliable. Latency is zero. Bandwidth is infinite. The network is secure. Topology does not change. There is one administrator. Transport cost is zero. The network is homogeneous.
3. Every one is false, and every microservice architecture that ignores one will fail in exactly the way that fallacy predicts.
4. Distributed tracing derives from Google's Dapper paper, published in 2010. The modern interoperability standard is W3C Trace Context, a Recommendation since 2021, which defines the `traceparent` and `tracestate` HTTP headers.
5. Sagas come from a 1987 paper by Hector Garcia-Molina and Kenneth Salem at Princeton, originally about long-lived database transactions. Two coordination styles exist: choreography, where services react to each other's events, and orchestration, where one coordinator drives the sequence and knows the undo steps. Choreography is looser and harder to debug; orchestration is clearer and centralizes the logic.
6. The consistency background is the CAP theorem: conjectured by Eric Brewer in 2000 and proved by Seth Gilbert and Nancy Lynch in 2002. Under a network partition you must choose between consistency and availability. Most business systems choose availability and reconcile afterwards.
7. Required infrastructure before splitting, and this list is the real cost: service discovery, centralized configuration, distributed tracing, correlated structured logging, per-service alerting and ownership routing, contract testing between services, an API gateway, retries with exponential backoff and jitter, circuit breakers, idempotency keys, and a deployment pipeline per service.
8. If your organization cannot build or buy that list, splitting will make reliability worse, not better. This is the Bezos mandate lesson restated: Amazon built all of it, and that is why the structure worked.
9. Published reversals are instructive. In March 2023 an Amazon Prime Video team wrote publicly that moving their audio and video monitoring service from a distributed serverless design to a single scaled process reduced infrastructure cost by over 90 per cent. Shopify and Stack Overflow are long-standing examples of very large systems run as well-structured monoliths.
10. Martin Fowler's 2015 article MonolithFirst argues that almost every successful microservice system he had seen started as a monolith that was later split, and almost every system built as microservices from scratch ran into serious trouble. That remains the best short statement of the position.
11. Where experts disagree: proponents argue that organizational independence is worth the cost from a surprisingly small size, especially with many teams and high turnover. Critics argue the boundaries chosen before you understand the domain are almost always wrong, and wrong boundaries are far more expensive across a network than inside one process. The second view has gained ground since about 2022.
12. The defensible default for a small team: one deployable unit; modules with explicit interfaces; no module reading another module's tables; a single database with clear schema ownership; and extraction only when a specific, named pressure demands it.

43.18.6 WORDS - remember these

1. Monolith — one program with everything in it — a single deployable unit containing all business capabilities.
2. Modular monolith — one deployable unit with strict internal walls — enforced module boundaries and no cross-module data access, without network calls.
3. Microservices — many small independently deployed programs — services owned by separate teams communicating only over the network.
4. Distributed monolith — the worst of both — services that must be deployed together, paying network costs without gaining independence.

5. Saga — a sequence of steps with defined undos — a long-running transaction composed of local transactions plus compensating actions, Garcia-Molina and Salem 1987.
6. Idempotency key — a token making a repeat safe — a client-supplied identifier letting a server recognize and collapse duplicate requests.
7. Distributed tracing — following one request across services — correlated spans under one trace identifier, propagated via W3C Trace Context headers.
8. Fallacies of distributed computing — the eight false assumptions — Deutsch's seven from about 1994 plus Gosling's eighth.

43.98 Common wrong ideas

1. Wrong: the cloud is just someone else's computer, so it is the same thing as renting a server. Right: the rented computer is the smallest part. What you are buying is self-service provisioning in seconds, elasticity, a metered bill, and roughly 200 managed services you did not have to build. A rented server with a two-week ticket queue fails the NIST definition on two of its five required characteristics.
2. Wrong: serverless means there are no servers. Right: there are many servers. You do not choose, name, patch or pay for them while idle. You still hit memory limits, execution timeouts, concurrency quotas and cold starts, all of which are properties of real machines.
3. Wrong: cloud is always cheaper. Right: cloud is cheaper for spiky, unpredictable or small workloads, and usually more expensive per unit for steady large ones. 37signals reported saving nearly 2 million US dollars a year by leaving, and Dropbox reported 74.6 million over two years. Both had large, steady, predictable workloads and their own engineers.
4. Wrong: multi-cloud avoids lock-in. Right: it usually multiplies it. You now depend on the intersection of features from two providers, need staff expert in both, pay egress between them, and lose the deep managed services that made the cloud worth using. Genuine portability comes from boundaries in your own code, not from a second account.
5. Wrong: a container is a lightweight virtual machine. Right: a container is a process on the host kernel, isolated by namespaces and limited by cgroups. There is no second operating system and no hardware boundary. That is why it starts in milliseconds and why a kernel flaw crosses it.
6. Wrong: deploying across availability zones protects me from an outage. Right: it protects you from one building. It does not protect you from a regional control plane failure, which is what actually happens most often. The AWS event of 20 October 2025 affected an entire region for more than fifteen hours while the individual zones kept their power.
7. Wrong: Kubernetes is the modern way to run applications, so we should use it. Right: Kubernetes solves problems that appear with many services, many teams and real bin-packing pressure. Below that, it is a permanent tax in upgrades, networking, certificates and new failure modes. Most teams are better served by a managed container service.
8. Wrong: the provider is responsible for security. Right: the provider secures the cloud; you secure what you put in it. Access control and data classification never transfer, in any service model. Nearly every publicized cloud breach has been a customer misconfiguration, not a provider failure.
9. Wrong: AWS was built to sell Amazon's spare Christmas capacity. Right: AWS was a deliberate new business, proposed in writing in 2003 with its own staffing plan, built on separate hardware. Amazon's Chief Technology Officer publicly denied the spare capacity story in 2011, and retail's peak coincides with customers' peak, so the arrangement could never have worked.
10. Wrong: microservices make systems more reliable. Right: they make failure partial rather than total, which is only an improvement if you have built tracing, retries, idempotency, circuit breakers and ownership routing. Split without those and you get a distributed monolith: all the network failure modes with none of the independence.

43.99 Chapter summary in 20 lines

1. The cloud is other people's computers in other people's buildings, rented by the second and reached over the internet; the mystique is marketing.
2. What genuinely changed is self-service in seconds, elasticity, metered billing and somebody else owning the hardware; what did not change is that physical machines still fail.
3. NIST SP 800-145, from September 2011, gives the five-part test: self-service, network access, pooling, elasticity and measurement.
4. A datacentre is racks measured in 44.45 millimetre units, top-of-rack switches in a spine-leaf fabric, dual power through UPS and generators, and hot and cold aisle cooling; efficiency is PUE, globally stuck near 1.54 while hyperscalers reach 1.09 to 1.15.
5. A region is a place, an availability zone is an independent failure domain inside it, and an edge location is a cache; deploy across zones always, and treat multi-region as a large project rather than a checkbox.
6. India has AWS Mumbai from 2016 and Hyderabad from 2022, Azure's three regions from 2015 plus India South Central from August 2026, and Google Mumbai from 2017 and Delhi NCR from 2021.
7. Virtualization made renting possible: a hypervisor slices one machine into many, made cheap by Intel VT-x in 2005 and AMD-V in 2006, and made nearly free by offload hardware such as AWS Nitro from 2017.
8. A container is a process with namespaces and cgroups on a shared kernel, not a small virtual machine; the idea runs back through LXC, Solaris Zones, FreeBSD jails and chroot in 1979, and Docker in 2013 supplied the packaging.
9. Kubernetes, first committed on 6 June 2014 and descended from Google's Borg, works by declarative reconciliation: state the desired end and let control loops close the gap forever.
10. Most teams do not need Kubernetes, and saying so is not contrarian; it is what the cost of upgrades, networking and new failure modes implies.
11. Around 2002 Jeff Bezos mandated that all teams expose data and functionality only through service interfaces, with no back doors, and that every interface be designed to be externalizable, on pain of dismissal.
12. That mandate forced Amazon to build service discovery, quotas and throttling, pager routing, monitoring merged with quality assurance and standard sandboxes, which is precisely the toolkit of a cloud platform.
13. It also bought latency, partial failure and operational cost; most companies copied the structure without the discipline and got a distributed monolith.
14. AWS itself came from a 2003 paper by Chris Pinkham and Benjamin Black and a parallel business proposal by Andy Jassy; SQS launched in beta in November 2004, S3 on 14 March 2006 and EC2 on 25 August 2006 at ten cents an hour.
15. It was never built from spare retail capacity, and AWS reported 128.7 billion US dollars of net sales for 2025 against Azure's 100 billion for fiscal 2026.
16. The service models differ only in where the line falls between what you manage and what the provider manages, and the security version of that line is the shared responsibility model, in which access control is always yours.
17. Serverless means per-invocation billing and no idle cost, at the price of cold starts of 200 milliseconds to two seconds, statelessness, a fifteen minute ceiling and heavy lock-in.
18. Storage splits into object, block and file; the decisive commercial fact is that storing ten terabytes costs about ten dollars a month in deep archive while reading it out once costs about nine hundred.
19. Cloud networking is a VPC of subnets and route tables, with stateful security groups on machines and stateless network ACLs on subnets, and identity is governed by policies where an explicit deny always wins.
20. Bills explode from idle resources, oversized instances, egress and cross-zone chatter rather than from compute; own the steady baseline, rent the burst, and start a small system as a modular monolith.

44

CHAPTER

KB

Security

44.0 What this chapter gives you

1. You will be able to say what security actually means, using the five properties professionals name, and build a threat model in plain words.
2. You will be able to explain what a hash is, why MD5 and SHA-1 are dead, and read real hash values computed in front of you.
3. You will be able to store passwords correctly, choose Argon2id parameters that a working engineer would accept, and explain salt versus pepper.
4. You will be able to describe symmetric and asymmetric encryption, name the modes that are safe in 2026, and say why nonce reuse destroys everything.
5. You will be able to explain post-quantum cryptography, the standards NIST approved on 13 August 2024, and why "harvest now, decrypt later" matters.
6. You will be able to describe every common web vulnerability class, the flawed pattern behind it, and the fix that removes it by construction.
7. You will be able to explain memory safety, why C allows the bugs it does, and what the 2023 to 2025 government guidance actually asks for.
8. You will be able to name the realistic ways machines get infected, recognize social engineering, and run a first hour of incident response calmly.
9. You will read this as a defender. The chapter explains how attacks work so that you can stop them, and contains no working attack code by design.
10. You will build on two earlier chapters rather than repeat them: Chapter 32 covered TLS and certificates, and Chapter 42 covered SSH keys and tokens.

44.1 What security actually means

44.1.1 PLAIN – in simple words

1. Security is not one thing. It is a small set of promises about your data.
2. Promise one: only the people who should see it can see it. That is **confidentiality**.
3. Promise two: nobody can change it without you noticing. That is **integrity**.
4. Promise three: when you need it, it is there. That is **availability**.
5. Those three together have a nickname in the trade: the CIA triad. It has nothing to do with any spy agency. The letters are just the three words.
6. Two more promises matter and are often left out.
7. Promise four: the thing claiming to be Ravi really is Ravi. That is **authenticity**.
8. Promise five: once Ravi signed something, he cannot later say he did not. That is **non-repudiation**.
9. Notice that these promises can fight each other.
10. Locking a file with a key nobody has protects confidentiality perfectly and destroys availability completely.

11. So security is always a set of trade-offs, never a single maximum.
12. And this is the sentence to carry with you: security is a property of a whole system in a particular situation. It is not a product you buy.
13. A very strong lock on a door with a glass panel next to it is not security. It is a strong lock.

44.1.2 PLAIN - a picture in your head

1. Think of a small jewellery shop on a busy street.
2. Confidentiality is the curtain in the back room, so passers-by cannot see the safe being opened.
3. Integrity is the sealed ledger, so nobody can quietly add a line saying they already paid.
4. Availability is the shop being open at ten in the morning, as promised.
5. Authenticity is checking the ID of the man who says he is from the insurance company.
6. Non-repudiation is the signed receipt the customer cannot later deny.
7. Now do the thing a security professional does. Ask three questions.
8. What am I protecting? The gold, the ledger, the customer list, the reputation.
9. From whom? A casual thief, a professional gang, a dishonest employee, a competitor, a government.
10. What happens if I fail? Money lost, customers gone, licence revoked, somebody physically harmed.
11. Those three questions, written down, are a **threat model**.

Where this comparison breaks: a shop thief must physically be on that street. A network attacker can be in another country, can try a million shops per hour, and can automate everything. Scale is the difference that changes everything. A lock that keeps out ninety-nine of a hundred thieves is a good lock in a street and a broken lock on the internet, because the internet will send you ten million thieves per day and only needs one to succeed.

44.1.3 PLAIN - a worked example

1. Let us threat model something real: the reader's own laptop and its GitHub account, from the running example used in this book.
2. Assets, written plainly.

Asset	Why it matters	If lost
Source code	The work itself	Leak, theft
GitHub token	Can push code	Malicious commits
SSH private key	Identity to servers	Impersonation
Laptop disk	Everything at once	Total compromise

3. Adversaries, from cheapest to most expensive.
4. An automated scanner that tries known passwords on anything it finds. Cost to the attacker: near zero. Likelihood: certain, continuously.
5. A phishing email pretending to be GitHub. Cost: low. Likelihood: high.
6. A thief who steals the laptop from a cafe. Cost: moderate. Likelihood: low but real.
7. A targeted attacker who wants this specific person. Cost: high. Likelihood: low unless the person is unusually interesting.
8. Now score the risk. **Risk = likelihood x impact**. Both parts matter.
9. Stolen laptop: likelihood low, impact very high. Product: medium-high. So full disk encryption is worth it.
10. Automated password guessing: likelihood certain, impact high. Product: very high. So a hardware key or passkey on GitHub is worth it.
11. A nation-state adversary: likelihood very low for most readers. Do not spend your budget there first.
12. That last line is the whole discipline. You cannot defend everything, so you rank by likelihood times impact and start at the top.

44.1.4 PLAIN - what is really happening inside

1. Every system has an **attack surface**: the complete list of places where untrusted input can reach your code.
2. Open network ports are attack surface. So are file uploads, query parameters, environment variables, image parsers and USB sockets.
3. Attackers do not "hack in" the way films show. They find one place where your assumption about input was wrong.
4. The pattern is nearly always the same. A program expected data and received instructions, or expected a small thing and received a big one.
5. There is a second, equally common pattern: the program did the right check on the wrong thing, or forgot the check on one path out of twenty.
6. Attacks chain. A low-value flaw plus another low-value flaw often equals a total compromise.
7. Example of a chain: a public page leaks internal server names, one of those servers has an old unpatched service, that service runs as administrator, and administrator can read the database credentials.
8. No single link in that chain was catastrophic. The chain was.
9. Defenders therefore think in layers, not in walls. If one control fails, the next one should still be standing. That is called **defence in depth**.
10. And defenders think about **blast radius**: when this component is fully compromised, what exactly can the attacker now reach?
11. Reducing blast radius is usually cheaper and more reliable than trying to make a component unbreakable.

The honest version: "attackers exploit vulnerabilities" is only half true. In real incident data, most intrusions begin with valid credentials or a person being tricked, not with a clever technical exploit. The technical exploit usually comes later, to move sideways once inside.

44.1.5 TECHNICAL - the engineer's version

1. The five properties are formalized in ISO/IEC 27000 and in NIST SP 800-53. Confidentiality, integrity and availability are the statutory triad in United States federal law under FISMA 2002.
2. Authenticity and non-repudiation are usually added as separate objectives; the combined five are sometimes called the Parkerian hexad when utility and possession are added as well, following Donn Parker's 1998 formulation.
3. Threat modelling has named methodologies. STRIDE, created at Microsoft by Loren Kohnfelder and Praerit Garg in 1999, enumerates Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege.
4. PASTA (Process for Attack Simulation and Threat Analysis, 2012) and LINDDUN for privacy threats are the other two commonly taught.
5. Attacks are catalogued by MITRE ATT&CK, first published in 2013 and updated continuously. It maps real observed adversary behaviour into tactics and techniques with identifiers such as T1566 for phishing.
6. Vulnerabilities get CVE identifiers, run by MITRE since 1999. Severity is scored by CVSS, currently version 4.0, published by FIRST in November 2023.
7. Weakness classes get CWE identifiers. CWE-79 is cross-site scripting, CWE-89 is SQL injection, CWE-787 is out-of-bounds write.
8. Note the difference precisely: a CWE is a kind of mistake, a CVE is one specific instance of it in one specific product version.
9. Risk in formal work is expressed as annualized loss expectancy: ALE = single loss expectancy x annual rate of occurrence. In practice the inputs are estimates, so the number is used for ranking, not for accounting.
10. FAIR (Factor Analysis of Information Risk) is the main quantitative alternative and expresses risk as probability distributions rather than single numbers.

Term	Formal meaning	Example
Threat	Actor with intent	Ransomware crew
Vulnerability	The weakness	Unpatched CVE
Exploit	Code using it	Working payload
Risk	Likelihood x impact	Ranked list

11. Tools that observe security posture on a machine you own: `nmap` for open ports, `ss -tlnp` on Linux or `lsof -i -P` on macOS for listening sockets, `osqueryd` for fleet-wide state, and `lynis` for host audit.
12. The word "secure" without a threat model attached is meaningless. Always ask: secure against whom, doing what, at what cost to them?

44.1.6 WORDS - remember these

Confidentiality — only the right people can read it — protection of data from unauthorized disclosure.

Integrity — nobody changed it secretly — assurance that data has not been modified in an unauthorized or undetected way.

Availability — it works when you need it — the property of being accessible and usable on demand by an authorized entity.

Authenticity — you are who you claim — verified identity of a principal or origin of data.

Non-repudiation — you cannot deny you did it — cryptographic proof binding an action to an identity, resistant to later denial.

Threat model — who might attack and how — structured enumeration of assets, adversaries, entry points and mitigations.

Attack surface — every way in — the complete set of points where an untrusted actor can supply input or influence behaviour.

Blast radius — how far damage spreads — the set of assets reachable after a given component is fully compromised.

Risk — likelihood times impact — expected loss, used to rank work rather than to predict exact outcomes.

44.2 Hashing

44.2.1 PLAIN - in simple words

1. A **hash function** takes any amount of data and produces a short, fixed-length fingerprint.
2. Feed it one letter or a two-gigabyte film. The fingerprint is the same length every time.
3. It is a one-way street. From the data you can compute the fingerprint easily. From the fingerprint you cannot get the data back.
4. Change one single bit of the input and about half the bits of the output change, in an unpredictable way.
5. That effect has a name: the **avalanche effect**.
6. The same input always gives the same output, on any machine, forever. It is not random.
7. Four properties matter, and they are worth learning in order.
8. One: it is fast to compute forwards.
9. Two: **preimage resistance**. Given a fingerprint, you cannot find any input that produces it.
10. Three: **second preimage resistance**. Given one file, you cannot find a different file with the same fingerprint.
11. Four: **collision resistance**. You cannot find any two different files with the same fingerprint, even if you get to choose both.

12. Collision resistance is the hardest of the four, and it is always the first one to fall when a hash function is broken.
13. A hash is good for checking that something did not change. A hash by itself is not a secret and not encryption. It hides nothing you can guess.

44.2.2 PLAIN – a picture in your head

1. Imagine a machine that turns any book into a six-word summary.
2. The summary is always six words, whether the book is a pamphlet or a thousand pages.
3. From the six words nobody can reconstruct the book.
4. Change one comma in the book and the six words come out completely different, not slightly different.
5. So if a friend tells you the six words over the phone, you can check that the copy of the book you received is exactly the one they sent.
6. That is what people do when they publish a checksum next to a download.
7. A **collision** is two genuinely different books producing the same six words. If that ever happens, the summary can no longer prove anything.

Where this comparison breaks: a six-word summary of a book is related to the book's meaning. A cryptographic hash is deliberately not. Two files that differ by a single byte give summaries with no visible relationship at all, and that unpredictability is the entire point. Also, a real hash has a fixed tiny output while the input is unlimited, so collisions must mathematically exist. The claim is only that nobody can find one.

44.2.3 PLAIN – a worked example

1. Here are real hashes, computed in the sandbox used to write this chapter, with OpenSSL 3.0.13.
2. The input is the seven characters `kedbyte`, with no newline.

```
md5      31d826fc69ac61899d792d7c21411dd5
sha1     ea431b1d4e594752169a04adea3245395d4ea7eb
sha256   7e88ddebb634ffbb28b3037dd122c08cf849ce53bd
         f3a8d0a08e13ac5b0eab3f
sha3-256 72c5e9c053ffabec027977d730987430dc98f9b553
         ac1aa248bc683385b532b1
blake2s  03b9df1f134bdbf34740d2353323dd58f48360c23c
         3af7b393fda6e52c6c14af
```

3. Note the lengths. MD5 is 32 hex characters, which is 128 bits. SHA-1 is 40 characters, 160 bits. SHA-256 is 64 characters, 256 bits.
4. Now change the last letter only, from `kedbyte` to `kedbytf`, and hash again with SHA-256.

```
kedbyte -> 7e88ddebb634ffbb28b3037dd122c08c...
kedbytf -> 718433d73cc98a937bdfce4a6c2e4d90...
```

5. Counted properly, 123 of the 256 output bits differ. The ideal is 128, which is exactly half. That is the avalanche effect measured.
6. SHA-256 of the empty input is a constant worth recognizing, because it turns up in logs and in Git internals: `e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`.
7. On the machine used here, `openssl speed sha256` measured 400 megabytes per second on 8 kilobyte blocks, on one core.
8. That speed is wonderful for verifying downloads and terrible for storing passwords, which is the subject of the next section.

44.2.4 PLAIN – what is really happening inside

1. Most classic hash functions work by chewing the input in fixed-size blocks.
2. SHA-256 uses 512-bit blocks. It keeps eight 32-bit working values, called the state, that start from fixed constants.
3. For each block it runs 64 rounds of mixing: rotations, exclusive-or, additions and a few nonlinear choices between bits.
4. After each block, the mixed values are added back into the state. That step is what makes the process irreversible.
5. At the end, the eight state values are written out as the digest.
6. The padding rule matters. The message gets a single 1 bit, then zeros, then the original length in bits. Without the length, different messages could collide trivially.
7. This design is called Merkle-Damgard, after Ralph Merkle and Ivan Damgard, both 1979 to 1989 era work.
8. It has a known quirk called **length extension**: if you know the hash of a secret message and its length, you can compute the hash of that message with extra data appended, without knowing the secret.
9. That quirk is exactly why you must never authenticate a message by hashing a secret joined to the message. You use HMAC instead, which is built to resist it.
10. SHA-3 has a completely different internal shape called a sponge. It absorbs input into a large 1600-bit state and squeezes output back out, and it is immune to length extension by design.
11. BLAKE3 goes further and splits the input into 1 kilobyte chunks arranged in a tree, so many cores can hash different chunks at the same time.
12. When a hash is "broken", it almost never means someone can reverse it. It means someone found a shortcut to produce collisions faster than brute force. That is still fatal for signatures and certificates.

44.2.5 TECHNICAL – the engineer's version

1. MD5 was published by Ron Rivest in 1992 as RFC 1321, output 128 bits.
2. Practical MD5 collisions were announced by Wang Xiaoyun, Feng Dengguo, Lai Xuejia and Yu Hongbo at the CRYPTO 2004 rump session in August 2004.
3. By 2008 a research team used chosen-prefix MD5 collisions to forge a working certificate authority certificate. In 2012 the Flame malware used the same class of attack to forge a Microsoft code-signing certificate.
4. SHA-1 was published by NIST in 1995, output 160 bits. Theoretical attacks appeared in 2005, also from Wang's group.
5. The first public SHA-1 collision, named SHAttered, was announced on 23 February 2017 by CWI Amsterdam and Google. It cost roughly nine quintillion SHA-1 computations, about 6,500 CPU-years and 110 GPU-years.
6. In 2020 Gaetan Leurent and Thomas Peyrin demonstrated a chosen-prefix SHA-1 collision for a rented compute cost in the tens of thousands of dollars.
7. SHA-2 is a family standardized in FIPS 180-4: SHA-224, SHA-256, SHA-384, SHA-512, and the truncated SHA-512/224 and SHA-512/256. No practical break exists as of 2026.
8. SHA-3 is Keccak, by Guido Bertoni, Joan Daemen, Michael Peeters and Gilles Van Assche. It won the NIST competition in October 2012 and was published as FIPS 202 in August 2015. It is a complement to SHA-2, not a replacement.
9. BLAKE3 was released in January 2020 by Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves and Zooko Wilcox-O'Hearn. It is not a NIST standard.
10. Security levels, stated honestly. A hash with n bits of output gives about n bits against preimage attacks and only n/2 bits against collisions, by the birthday bound. SHA-256 therefore offers 128-bit collision resistance.

Function	Output bits	Status 2026
MD5	128	Broken 2004
SHA-1	160	Broken 2017
SHA-256	256	Recommended
SHA-3-256	256	Recommended
BLAKE3	256 default	Fast, not FIPS

11. Where each is still acceptable: MD5 and SHA-1 only for non-security uses such as cache keys and non-adversarial deduplication. Never for signatures, certificates, integrity against an attacker, or passwords.
12. Git historically used SHA-1 for object names and has a SHA-256 object format available since Git 2.29 in 2020, still not the default in 2026. Git also ships a hardened SHA-1 that detects the SHAttered attack pattern.
13. Commands to observe: `sha256sum file` and `md5sum file` on Linux, `shasum -a 256 file` on macOS, `openssl dgst -sha256 file` anywhere, `certutil -hashfile file SHA256` on Windows.
14. For message authentication use HMAC (RFC 2104, 1997), for example HMAC-SHA-256, or use a modern AEAD cipher which includes authentication.

44.2.6 WORDS - remember these

Hash function — a fingerprint machine — a deterministic map from arbitrary input to fixed-length output.

Digest — the fingerprint itself — the output value of a hash function.

Collision — two inputs, one fingerprint — two distinct messages producing the same digest.

Preimage resistance — cannot work backwards — infeasibility of finding any input matching a given digest.

Avalanche effect — one bit in, half the bits out — small input change causes statistically independent output.

Length extension — appending without the secret — a Merkle-Damgard weakness allowing extension of a hashed message.

HMAC — a hash with a key done properly — keyed message authentication code defined in RFC 2104.

Checksum — a change detector — a short value used to detect accidental corruption, not necessarily cryptographic.

44.3 Password storage done properly

44.3.1 PLAIN - in simple words

1. If you build a login system, you must store something that lets you check a password without storing the password.
2. Storing the password itself, readable, is the worst possible choice. One database leak and every user is exposed everywhere they reused it.
3. So we store a hash instead. The user types a password, we hash it, we compare hashes. We never need the original.
4. But a plain fast hash is almost as bad, and here is why.
5. An attacker who steals the database can guess billions of passwords per second on one graphics card and hash each guess.
6. Worse, if everyone's password is hashed the same way, the attacker can precompute a giant table once and reuse it against every stolen database.
7. Those precomputed tables are called **rainbow tables**.

8. The fix for that is a **salt**: a random value, different for every user, mixed in before hashing and stored alongside the hash.
9. Two users with the same password now get completely different stored values, so a precomputed table is useless and every user must be attacked separately.
10. A salt is not a secret. It is stored in the database in plain view. Its job is uniqueness, not secrecy.
11. Then there is a second idea: a **pepper**, a single secret value added for everyone, stored somewhere other than the database.
12. If the attacker steals only the database, the pepper is missing and the guessing cannot even start.
13. And the last idea is the important one: make the hash deliberately slow and memory-hungry.
14. If checking one password takes a quarter of a second and a lot of memory, a legitimate login is unaffected and mass guessing becomes uneconomic.

44.3.2 PLAIN - a picture in your head

1. Think of a hotel that must check whether you know the door code, without keeping a list of door codes.
2. Fast hash: the clerk has a machine that instantly turns a code into a coloured token, and keeps only the tokens. A thief who steals the tokens buys the same machine and tries every code in an afternoon.
3. Salt: each guest's code is combined with a printed sticker number unique to that guest before the machine runs. The thief must now redo the whole afternoon separately for each guest.
4. Pepper: the machine also needs a small key that the manager keeps at home. Stealing the tokens gets you nowhere without visiting the manager.
5. Slow hash: the machine takes a quarter of a second per attempt and needs a whole desk covered in paper to work. One guest checking in does not care. A thief trying a billion codes now needs a warehouse and years.

Where this comparison breaks: the hotel thief cannot buy a hundred thousand machines. A password attacker can rent that many in a cloud for an hour. The slow hash is not making attack impossible; it is changing the price. Price is the real currency of security, and it is a moving target because hardware gets faster. Parameters chosen in 2015 are too weak in 2026.

44.3.3 PLAIN - a worked example

1. Real timings, measured in the sandbox for this chapter, single core.

Method	Time per check	Attacker friendly?
Raw SHA-256	0.00000074 s	Extremely
PBKDF2 600k	0.42 s	No
scrypt N=2^17	1.08 s	No

2. Read the first row again. A raw SHA-256 took 740 nanoseconds. That is roughly 1.3 million guesses per second on one Python core.
3. On a graphics card the numbers are far worse for the defender. Published hashcat benchmarks on an NVIDIA RTX 5090 report about 220 billion MD5 hashes per second and about 28 billion SHA-256 hashes per second.
4. The same card manages only about 300,000 bcrypt hashes per second at a low work factor. That is roughly a hundred thousand times slower per guess.
5. That single ratio is the entire argument for slow password hashing.
6. Now watch the salt work. The same password, PBKDF2-HMAC-SHA256 with 600,000 iterations, with two different salts.

```
salt 00000000... -> 0460eeec7ddf8b5f91f2037b3e2ab248
salt a3f10000... -> 52e0053a9dda9d910cb58befaafe9e67
```

7. Same password, unrelated results. No shared precomputation is possible.
8. A stored record in practice looks like this. It is one string containing the algorithm, its parameters, the salt and the hash, so you can change parameters later without breaking old accounts.

```
$argon2id$v=19$m=19456,t=2,p=1$c29tZXNhbHQ$aG...
 |        |    |                |          |
 algo     ver  parameters       salt       hash
```

9. The parameters live inside the record on purpose. When you raise them next year, old users still verify with the old settings, and you re-hash their password at their next successful login.

44.3.4 PLAIN - what is really happening inside

1. On registration: generate a fresh random salt of at least 16 bytes from a cryptographic random source, run the slow hash, store algorithm, parameters, salt and result.
2. On login: read the stored record, take the salt and parameters out of it, hash the submitted password the same way, compare.
3. Compare in constant time. A comparison that stops at the first different byte leaks information through timing. Every good library has a constant-time compare function.
4. Never tell the user which part was wrong. “Wrong username or password” and nothing more, or you have built an account enumeration tool.
5. The slow hash resists attack in three different ways, and they are not the same thing.
6. Time hardness: many sequential iterations, so you cannot go faster than the chain allows.
7. Memory hardness: the algorithm needs a large block of memory that must stay filled during the computation. Graphics cards have thousands of small cores but limited memory per core, so memory cost hurts attackers far more than defenders.
8. Parallelism resistance: the memory accesses depend on earlier results, so the work cannot simply be split.
9. bcrypt has time hardness and a small fixed 4 kilobyte memory need. scrypt and Argon2 add real memory hardness.
10. The pepper is applied differently in good designs: not concatenated into the password, but used as an HMAC key over the password before hashing, or used to encrypt the resulting hash. Both keep it separable and rotatable.
11. Where does the pepper live? A hardware security module, a cloud key management service, or at minimum an environment variable on the application server that is not in the database backup.

The honest version: the pepper only helps in the specific case where the database leaks but the application server does not. That case is common, from SQL injection and stolen backups, so peppering is worth doing. But it is a second line, not a substitute for a proper slow hash.

44.3.5 TECHNICAL - the engineer’s version

1. bcrypt: Niels Provos and David Mazieres, USENIX 1999, based on the Blowfish key schedule. Cost parameter is a power of two. OWASP recommends a work factor of at least 10 as of the current Password Storage Cheat Sheet.
2. bcrypt truncates input at 72 bytes. Enforce a 72-byte maximum, or pre-hash with SHA-256 and base64 the result before feeding bcrypt, being careful about null bytes.
3. scrypt: Colin Percival, 2009, standardized as RFC 7914 in 2016. Parameters N (cost), r (block size), p (parallelism).
4. Argon2: winner of the Password Hashing Competition in July 2015, by Alex Biryukov, Daniel Dinu and Dmitry Khovratovich. Standardized as RFC 9106 in September 2021. Three variants: Argon2d, Argon2i, and Argon2id.
5. Use Argon2id. It is the hybrid: data-independent memory access in the first pass to resist side-channel attacks, data-dependent afterwards to resist time-memory trade-offs.

6. Current OWASP recommended parameters, any one of which is acceptable and equivalent in strength.

Algorithm	Parameters	Notes
Argon2id	m=47104, t=1, p=1	46 MiB memory
Argon2id	m=19456, t=2, p=1	19 MiB memory
scrypt	N=2^17, r=8, p=1	128 MiB memory
bcrypt	cost 10 minimum	72-byte limit

7. PBKDF2 is the fallback where FIPS validation is required. OWASP currently lists 600,000 iterations for PBKDF2-HMAC-SHA256 and 210,000 for PBKDF2-HMAC-SHA512. PBKDF2 is defined in RFC 8018.
8. Salt: 16 bytes minimum from a CSPRNG. Every modern library generates and embeds it for you. Do not hand-roll this.
9. Attack taxonomy, defensively stated. A **dictionary attack** tries a wordlist plus rules such as appending digits. A **brute force** tries all combinations, feasible only for short passwords. **Credential stuffing** replays username and password pairs stolen from another site.
10. Credential stuffing is the dominant real attack because password reuse is near universal. The attacker does not need to break your hashes at all if the user used the same password somewhere weaker.
11. That is the actual disaster of reuse: your security becomes equal to the security of the worst site the user ever signed up for.
12. Defences against stuffing: rate limiting per account and per network, device fingerprinting, checking submitted passwords against known-breached lists using a k-anonymity range API, and above all offering MFA.
13. NIST SP 800-63B revision 4, published 31 July 2025, is the current authority. Its key rules for verifiers: require at least 8 characters and encourage at least 15; allow at least 64; allow all printable characters including spaces and Unicode.
14. The same document says verifiers **shall not** impose composition rules such as requiring mixed character types, and **shall not** require periodic password rotation without evidence of compromise.
15. The reasoning is empirical, not aesthetic. Forced rotation causes predictable transformations such as incrementing a trailing digit, and composition rules push users toward the same small set of substitutions.
16. Practical advice to give a user: use a password manager, generate long random passwords per site, and use a long passphrase only for the few secrets you must type from memory, such as the manager's own master password and the disk encryption password.
17. Length beats complexity. A five-word random passphrase from a large word list carries roughly 64 bits of entropy; an eight-character password with the usual substitutions carries far less in practice.

44.3.6 WORDS - remember these

Salt — a unique random extra per user — public per-record value preventing precomputation and hash reuse.

Pepper — one secret extra for everyone — site-wide secret kept outside the password database, applied via HMAC or encryption.

Rainbow table — a precomputed cracking table — time-memory trade-off structure for reversing unsalted hashes.

Work factor — the slowness dial — a tunable cost parameter of a password hashing function.

Memory hardness — needs a lot of RAM to compute — property that raises the cost of parallel hardware attacks.

Credential stuffing — reusing stolen logins elsewhere — automated replay of breached username and password pairs.

Argon2id — the current recommended choice — hybrid memory-hard password hash, RFC 9106, winner of the 2015 Password Hashing Competition.

Account enumeration — learning which users exist — information leak from differing error messages or response times.

44.4 Symmetric encryption

44.4.1 PLAIN - in simple words

1. **Symmetric encryption** means both sides share one secret key. The same key locks and unlocks.
2. It is fast, which is why every real system uses it for the actual data.
3. There are two shapes of symmetric cipher.
4. A **block cipher** transforms a fixed-size chunk at a time, typically 16 bytes, and you must decide how to handle a message longer than one chunk.
5. A **stream cipher** produces an endless keystream of pseudo-random bytes which you combine with the message one byte at a time.
6. The dominant block cipher is **AES**, with key sizes of 128, 192 or 256 bits. Its block is always 128 bits regardless of key size.
7. The way you chain blocks together is called a **mode of operation**, and choosing it wrongly is the most common cryptographic mistake in the world.
8. The worst mode is ECB, which encrypts every block independently. Identical input blocks give identical output blocks, so the shape of your data leaks straight through.
9. Modern practice is not to choose a cipher and a mode separately. You choose an **authenticated encryption** scheme that also detects tampering.
10. Encryption without authentication is not enough. An attacker who cannot read your message can still flip bits in it and change what it says when decrypted.
11. Almost every cipher mode also needs a per-message unique value called an **IV** or a **nonce**. Reusing one with the same key is catastrophic.

44.4.2 PLAIN - a picture in your head

1. Think of a stencil-based code where each 16-letter chunk of your message is replaced by a scrambled 16-letter chunk, using a fixed rule set by the key.
2. ECB mode is applying that stencil to each chunk independently.
3. Now imagine a photograph converted this way. Every patch of plain white sky becomes the same scrambled patch. The scrambled version still shows the outline of the picture.
4. That is why the encrypted penguin image is famous in every cryptography course. The bird is still clearly visible after ECB encryption.
5. CBC mode fixes it by mixing each chunk with the previous encrypted chunk before scrambling, so identical inputs stop looking identical.
6. CTR mode does something different: it does not encrypt your data at all. It encrypts a counter to make a keystream, then combines that with your data.
7. GCM is CTR plus a running checksum computed with the key, so any change to the ciphertext is detected on decryption.

Where this comparison breaks: real block ciphers are not letter substitution. AES mixes bits across the whole block through several layers, so one changed input bit changes the whole output block. The stencil picture also suggests the key changes the alphabet, when in reality the key changes a schedule of operations applied over many rounds.

44.4.3 PLAIN - a worked example

1. Here is ECB failing, computed in the sandbox for this chapter with a real AES-128 key.
2. The plaintext is `YES YES YES YES YES YES YES YES`, which is exactly two identical 16-byte blocks.

```
AES-ECB block 1: 15516924691d0a3e5f81ff297ad6c86e
AES-ECB block 2: 15516924691d0a3e5f81ff297ad6c86e
identical: True

AES-CBC block 1: 15516924691d0a3e5f81ff297ad6c86e
AES-CBC block 2: 234d8347677995c4547c148bbfa58f4f
identical: False
```

3. Look at the ECB output. The repetition in the plaintext is visible in the ciphertext with no key and no effort at all.
4. CBC produced different blocks for identical input because each block was mixed with the previous ciphertext block first.
5. Now the nonce reuse problem, stated without giving an attack recipe.
6. In CTR and GCM, the ciphertext is the plaintext combined with a keystream derived from key and nonce.
7. Encrypt two different messages with the same key and the same nonce, and both use the identical keystream.
8. Combining the two ciphertexts cancels the keystream out entirely, leaving a direct relationship between the two plaintexts.
9. For GCM specifically, nonce reuse does something worse: it leaks the authentication subkey, which lets an attacker forge messages that your system will accept as genuine.
10. The rule that follows is absolute. Never reuse a nonce with the same key. Generate it randomly with enough bits, or use a strict counter you can prove never repeats.

44.4.4 PLAIN - what is really happening inside

1. AES works on a 4 by 4 grid of bytes called the state.
2. Each round does four steps: substitute every byte through a fixed lookup table, shift the rows sideways by different amounts, mix each column mathematically, and add the round key with exclusive-or.
3. AES-128 does 10 rounds, AES-192 does 12, AES-256 does 14.
4. The **key schedule** expands the single key into one distinct round key per round, so no two rounds are the same.
5. The substitution table is the only nonlinear step. Without it the whole cipher would collapse into simple algebra and be trivially breakable.
6. In CBC mode, encryption is a chain: block n is combined with ciphertext n-1 before encryption, and the first block uses the IV. Chains cannot be parallelized when encrypting, though decryption can be.
7. In CTR mode, block n of the keystream is the encryption of nonce plus n. No chain, so everything parallelizes and you can decrypt from the middle.
8. GCM adds GHASH, a multiplication in a finite field over the ciphertext and any associated data, producing a 16-byte authentication tag.
9. On decryption, GCM computes the tag again and compares. If it does not match, it returns an error and no plaintext. That refusal is the whole value of authenticated encryption.
10. ChaCha20 is a different design: 20 rounds of additions, rotations and exclusive-ors on a 4 by 4 grid of 32-bit words, producing 64 bytes of keystream per invocation.
11. It uses no lookup tables, so it has no cache-timing side channel and it is fast in plain software on processors without AES hardware instructions.
12. Poly1305 is its companion authenticator, a one-time message authentication code using arithmetic modulo a large prime.

44.4.5 TECHNICAL - the engineer's version

1. AES is Rijndael, by Joan Daemen and Vincent Rijmen. It won the NIST competition in October 2000 and was published as FIPS 197 in November 2001.
2. Modern processors have hardware support: Intel AES-NI since Westmere in 2010, ARMv8 cryptographic extensions since 2011. Throughput on a single modern core is commonly several gigabytes per second.
3. Mode summary, with the practical verdict for 2026.

Mode	Authenticated?	Verdict
ECB	No	Never use
CBC	No	Legacy only
CTR	No	Only inside AEAD
GCM	Yes	Standard choice
ChaCha20-Poly1305	Yes	Standard choice

4. AES-GCM is specified in NIST SP 800-38D (2007). Its nonce is 96 bits by convention, and the standard limits a single key to about 2^32 random nonces before collision risk becomes unacceptable.
5. AES-GCM-SIV, RFC 8452 (2019), is nonce-misuse resistant: repeating a nonce leaks only whether two plaintexts were equal, not the keystream. Use it when you cannot guarantee nonce uniqueness.
6. ChaCha20-Poly1305 is RFC 8439 (2018), by Daniel J. Bernstein. It is a TLS 1.3 mandatory-to-implement suite alongside AES-GCM and is preferred on mobile processors without AES acceleration.
7. XChaCha20-Poly1305 extends the nonce to 192 bits, making random nonce generation safe without counters.
8. CBC without authentication enabled the padding oracle class of attacks, including the 2002 Vaudenay result and the 2010 attacks on ASP.NET. That history is why encrypt-then-MAC or AEAD is mandatory now, not optional.
9. Key sizes in practice: AES-128 gives 128-bit classical security and is fine. AES-256 is chosen for long-term data and for regulatory reasons, and is the sensible default against future quantum search attacks.
10. Grover's algorithm, published by Lov Grover in 1996, would give a quantum computer a square-root speed-up on key search, notionally halving AES-256 to 128 bits of effective strength. Doubling the key size answers it.
11. Never write your own mode, your own padding, or your own comparison of authentication tags. Use libsodium, the Go `crypto/cipher` AEAD interface, Java's `javax.crypto` with GCM, or Python `cryptography`'s AESGCM class.
12. Commands to observe: `openssl enc -aes-256-cbc` for files, `openssl speed -evp aes-256-gcm` for throughput, `cryptsetup benchmark` on Linux for disk encryption performance including AES-XTS.
13. Disk encryption uses XTS mode rather than GCM, because a disk sector has no room to store an authentication tag. That is a real trade-off: disk encryption protects confidentiality, not integrity.

44.4.6 WORDS - remember these

Symmetric encryption — one shared key both ways — encryption where the same key encrypts and decrypts.

Block cipher — scrambles fixed chunks — a keyed permutation over a fixed block size, 128 bits for AES.

Mode of operation — how chunks are chained — the construction turning a block cipher into a scheme for arbitrary-length messages.

ECB — the mode that leaks patterns — Electronic Codebook, encrypts each block independently, never acceptable.

IV / nonce — the per-message unique value — initialization vector or number used once, must never repeat under one key.

AEAD — encryption that also detects tampering — Authenticated Encryption with Associated Data, for example AES-GCM.

Authentication tag — the tamper seal — short value verified on decryption, causing refusal if the ciphertext changed.

ChaCha20-Poly1305 — the software-fast alternative — stream cipher plus one-time authenticator, RFC 8439.

44.5 Asymmetric encryption, signatures and the quantum question

44.5.1 PLAIN - in simple words

1. Symmetric encryption has one hard problem: how do two strangers agree on a shared key over a network anyone can read?
2. **Asymmetric encryption** solves it with a pair of matched keys.
3. The **public key** can be published to the whole world. The **private key** never leaves its owner.
4. Anything encrypted with the public key can only be decrypted with the private key.
5. That direction gives you confidentiality: anyone can write to you, only you can read it.
6. Run it the other way and you get something different and equally useful.
7. If you transform a message with your private key, anyone with your public key can check it. That is a **digital signature**.
8. A signature proves three things at once: the message came from the holder of that private key, it has not been altered, and the signer cannot plausibly deny it later.
9. Encryption alone proves none of those. Anyone with your public key can encrypt to you, so a decrypted message tells you nothing about who sent it.
10. There is also a way to agree a shared secret without either side sending it: **Diffie-Hellman key exchange**.
11. Asymmetric operations are slow, hundreds to thousands of times slower than symmetric ones.
12. So every real system is **hybrid**: use asymmetric maths once to agree a symmetric key, then use the fast symmetric cipher for all the data.
13. That is exactly what TLS does, which Chapter 32 covered in detail.

44.5.2 PLAIN - a picture in your head

1. Picture an open padlock that you hand out freely, of which you alone keep the key.
2. Anyone can put a letter in a box, snap your padlock shut, and post it. Only you can open it.
3. Signing is the mirror image: you press a seal that only your ring can make, and anyone with a picture of your seal can confirm it is genuine.
4. Diffie-Hellman is a different trick. Two people each mix a private colour into a shared public colour, swap the mixtures in the open, then each mix their own private colour into the mixture they received.
5. Both end with the same final colour. An observer saw the two mixtures but cannot unmix them to reach it.

Where this comparison breaks: paint mixing is only a metaphor for a one-way mathematical operation. Real Diffie-Hellman uses exponentiation in a group where reversing it is the discrete logarithm problem. Also, a physical padlock protects against a person with a hacksaw, while a cryptographic one protects only against computation, and computation gets cheaper every year.

44.5.3 PLAIN – a worked example

1. RSA rests on multiplication being easy and factoring being hard.
2. Multiplying two 1024-bit primes takes microseconds. Recovering those primes from the 2048-bit product is beyond any known machine.
3. The largest RSA modulus ever publicly factored is RSA-250, a 829-bit number, broken in February 2020 using about 2,700 core-years.
4. That is why 2048-bit RSA keys are still standard and 1024-bit ones were deprecated by browsers around 2013 to 2014.
5. Elliptic curves get the same strength from much smaller numbers.

Security level	RSA / DH	Elliptic curve
112 bits	2048	224
128 bits	3072	256
192 bits	7680	384
256 bits	15360	512

6. Read the last row. To match a 512-bit curve you would need a 15,360-bit RSA key. That is why modern systems moved to curves.
7. Curve25519, published by Daniel J. Bernstein in 2005, gives roughly 128-bit security with 32-byte keys and is the default in SSH, Signal and TLS 1.3.
8. Its signature counterpart is Ed25519, standardized in RFC 8032 in 2017, with 64-byte signatures.
9. A sense of the speed gap: on a typical modern core, an X25519 key agreement takes tens of microseconds while an AES-GCM operation on a kilobyte takes well under one microsecond. Hence the hybrid design.

44.5.4 PLAIN – what is really happening inside

1. RSA key generation: pick two large random primes p and q, multiply to get the modulus n, choose a public exponent e, usually 65537, and compute the matching private exponent d.
2. Encryption raises the message to the power e modulo n. Decryption raises the result to the power d modulo n. The maths brings you back to the start.
3. Raw RSA is unsafe. Real use requires padding: OAEP for encryption, PSS for signatures. The old PKCS#1 v1.5 padding produced the Bleichenbacher attack family from 1998 onwards and its many modern echoes.
4. Signing does not actually sign the message. It hashes the message first, then signs the hash. That is why a broken hash breaks signatures, which is what SHA-1 collisions meant for certificates.
5. Elliptic curve cryptography replaces "multiply numbers" with "add points on a curve". The private key is a number, the public key is a point reached by adding a fixed base point to itself that many times.
6. Going forwards is fast; recovering the number from the point is the elliptic curve discrete logarithm problem, and no efficient classical method is known.
7. Diffie-Hellman in curve form is simple: each side multiplies the other's public point by their own private number, and both land on the same point.
8. **Forward secrecy** comes from throwing the Diffie-Hellman private values away after the session. An attacker who later steals the long-term key still cannot decrypt yesterday's recorded traffic.
9. Now the quantum part. Peter Shor published an algorithm in 1994 that, on a sufficiently large fault-tolerant quantum computer, factors integers and computes discrete logarithms in polynomial time.
10. That breaks RSA, finite-field Diffie-Hellman and every elliptic curve scheme at once. It does not break symmetric ciphers or hashes in the same way; those need only larger sizes.
11. No such machine exists in 2026. Published estimates for breaking 2048-bit RSA remain in the range of millions of physical qubits with error correction, against current devices measured in the low thousands.

12. But traffic recorded today can be stored and decrypted later. That is **harvest now, decrypt later**, and it makes migration urgent for any data that must stay secret for ten or twenty years.

44.5.5 TECHNICAL - the engineer's version

1. Diffie-Hellman was published by Whitfield Diffie and Martin Hellman in 1976 in "New Directions in Cryptography". Ralph Merkle's related work preceded it. RSA followed in 1977 from Ron Rivest, Adi Shamir and Leonard Adleman.
2. Elliptic curve cryptography was proposed independently by Neal Koblitz and Victor Miller in 1985.
3. British researchers at GCHQ, James Ellis, Clifford Cocks and Malcolm Williamson, developed equivalent ideas between 1969 and 1974; the work was classified until 1997.
4. NIST approved three post-quantum standards on 13 August 2024.

Standard	Name	Purpose
FIPS 203	ML-KEM	Key encapsulation
FIPS 204	ML-DSA	Digital signatures
FIPS 205	SLH-DSA	Hash-based signatures

5. ML-KEM was submitted as CRYSTALS-Kyber, ML-DSA as CRYSTALS-Dilithium, SLH-DSA as SPHINCS+. The initial announcement was 6 August 2024 and the Secretary of Commerce approved them on 13 August 2024.
6. A fourth signature scheme, FN-DSA, based on FALCON, was still in draft as FIPS 206 as of 2026. On 11 March 2025 NIST selected HQC as a fifth algorithm and a backup key encapsulation mechanism, with its own standard expected around 2027.
7. Sizes are the practical cost. ML-KEM-768 has a 1,184-byte encapsulation key and a 1,088-byte ciphertext, against 32 bytes each for X25519.
8. ML-DSA-65 signatures are about 3,309 bytes and SLH-DSA signatures range from roughly 7,856 bytes to about 49,856 bytes, against 64 bytes for Ed25519. That is why signature migration lags key exchange migration.
9. SLH-DSA exists as insurance. It relies only on hash function security, so if lattice mathematics turns out to have an unexpected weakness, hash-based signatures still stand.
10. Deployment state as of 2026, separating fact from claim. Established fact: the hybrid key agreement X25519MLKEM768 is on by default in current Chrome, Firefox and Edge. Cloudflare reported in late October 2025 that over half of human-initiated traffic to its network used post-quantum key agreement.
11. Also established: signatures and certificates have barely moved, because they need certificate authorities, root programmes and every client to change together.
12. Active research: how to shrink post-quantum certificate chains, and how to do post-quantum authentication for constrained devices.
13. Marketing claim: any product described as "quantum-proof" or "unbreakable" without naming FIPS 203, 204 or 205. Ask which standard, which parameter set, and whether it is hybridized with a classical algorithm.
14. Hybrid is the current professional consensus: run the classical and the post-quantum algorithm together, so a break in either alone is not fatal. Experts disagree on how long to keep hybrids. Some say they add complexity and should be retired soon; others argue for a decade.
15. Commands to observe: `openssl s_client -connect host:443` and read the negotiated group, `openssl list -kem-algorithms` on OpenSSL 3.5 and later, and `ssh -Q kex` to list key exchange algorithms your SSH build supports.

44.5.6 WORDS - remember these

Public key — the half you publish — the shareable component of an asymmetric key pair.

Private key — the half you never share — the secret component whose exposure destroys the pair's security.

Digital signature — proof of who and unaltered — a value verifiable with a public key, providing authenticity and non-repudiation.

Key exchange — agreeing a secret in the open — a protocol producing a shared secret without transmitting it, such as ECDH.

Forward secrecy — yesterday stays safe — property where compromise of long-term keys does not expose past sessions.

Hybrid encryption — asymmetric once, symmetric after — the standard construction combining key agreement with an AEAD cipher.

Shor's algorithm — the quantum threat to RSA — polynomial-time quantum factoring and discrete logarithm, published 1994.

ML-KEM — the new key exchange standard — FIPS 203, lattice-based key encapsulation, approved 13 August 2024.

Harvest now, decrypt later — record today, break tomorrow — the threat model justifying migration before quantum computers exist.

44.6 Authentication and identity

44.6.1 PLAIN - in simple words

1. **Authentication** answers one question: are you who you claim to be?
2. There are three classic kinds of evidence, called factors.
3. Something you **know**: a password, a PIN, an answer to a question.
4. Something you **have**: a phone, a hardware key, a smart card.
5. Something you **are**: a fingerprint, a face, a voice.
6. Multi-factor authentication means using evidence from two different kinds, not two things of the same kind.
7. A password plus a security question is not two factors. Both are things you know, and both leak in the same database breach.
8. Not all second factors are equally strong, and the gap between them is much wider than most people think.
9. A six-digit code from an app is decent. A code sent by text message is weak. A hardware key is in a different class entirely.
10. The reason hardware keys are different is that they refuse to be phished. The others can all be relayed by an attacker in real time.
11. A **passkey** is that same phishing-resistant technology, with the secret stored on your phone or laptop and unlocked by your fingerprint or face.

44.6.2 PLAIN - a picture in your head

1. Think of collecting a parcel from a counter.
2. The password is knowing the reference number. Anyone who overhears it can collect your parcel.
3. The app code is a slip of paper that changes every thirty seconds. Better, but if someone phones you and talks you into reading it out within those thirty seconds, they can still collect the parcel.

4. The hardware key is different in kind. It is a stamp that only works on that one counter, in that one building.
5. Take the stamp to a fake counter in the next street and it produces nothing, because it checks the address of the counter before stamping.
6. That address check is why phishing fails against it. The user can be fully fooled and the attack still does not work.

Where this comparison breaks: a stamp can be stolen physically. A hardware key can too, which is why it usually also requires a PIN or a fingerprint. And the key does not check a street address but the exact web origin, which the browser supplies and the user cannot override by being persuaded.

44.6.3 PLAIN – a worked example

1. Here is how the six-digit code in your authenticator app is actually made.
2. When you scan the QR code, the site gives your app a shared secret, usually 20 bytes, encoded in base32.
3. Both sides take the current Unix time in seconds and divide by 30, throwing away the remainder. That gives a counter that changes every 30 seconds.
4. Both compute HMAC-SHA-1 of that counter using the shared secret.
5. That produces 20 bytes. A truncation step picks 4 bytes from a position determined by the last nibble of the hash, giving a 31-bit number.
6. Take that number modulo 1,000,000 and pad to six digits. That is your code.

```
secret + floor(unixtime / 30)
   -> HMAC-SHA-1
   -> dynamic truncation (31 bits)
   -> mod 1000000
   -> "042318"
```

7. Nothing is transmitted to generate it. That is why the app works on a plane with no signal.
8. Servers usually accept the previous and next window too, to tolerate clocks being a little out.
9. Now see the weakness clearly. The code is just six digits typed by a human. A convincing fake login page collects it and replays it within 30 seconds.
10. That is not a flaw in the maths. It is a flaw in the shape of the interaction, and no amount of better hashing fixes it.

44.6.4 PLAIN – what is really happening inside

1. SMS one-time codes are weaker still, for reasons outside your control.
2. **SIM swapping**: an attacker persuades a mobile operator to move your number to their SIM, using social engineering on a support agent. Your codes now arrive on their phone.
3. The mobile signalling network itself has known interception weaknesses, and messages travel through third parties in plain form.
4. Push-approval prompts fixed the typing problem but created a new one: **MFA fatigue**, where an attacker with your password sends prompt after prompt until you approve one to make it stop, often at three in the morning.
5. This is not theoretical. The September 2022 Uber breach began with a contractor's stolen password plus repeated push prompts, finished off by a message pretending to be internal IT support.
6. Number matching, where the login screen shows two digits you must type into the prompt, is the standard mitigation and is now on by default in the major identity providers.
7. Now the hardware key. Registration: the key generates a fresh key pair for that one website, keeps the private key inside the hardware, and sends the public key to the site.
8. Login: the site sends a random challenge. The browser adds the exact origin it is talking to. The key signs both together.

9. If the origin is a look-alike domain, the signature is over the wrong origin and the real site rejects it. There is no code for a human to relay.
10. The private key never leaves the device. There is no shared secret in the site's database, so a breach of the site leaks only public keys.
11. A **passkey** is the same protocol with the key material held by the operating system or password manager and typically synchronized across your devices through your platform account.
12. That synchronization is the trade-off. It makes recovery practical, at the cost of the cloud account becoming part of the security boundary.

44.6.5 TECHNICAL - the engineer's version

1. HOTP is RFC 4226 (2005), counter-based. TOTP is RFC 6238 (2011), time-based, with a default 30-second step and 6 digits. The default HMAC is SHA-1, which is acceptable here because HMAC-SHA-1 is not broken by collisions.
2. FIDO2 combines two specifications: W3C Web Authentication, WebAuthn Level 2 became a W3C Recommendation in April 2021 with Level 3 in progress, and CTAP2 from the FIDO Alliance for the link between browser and authenticator.
3. WebAuthn signs a client data structure containing the challenge, the origin and the type. Origin binding is the phishing resistance, and it is a property of the protocol, not of user training.
4. Attestation optionally proves what kind of authenticator was used. Enterprises use it to require certified hardware; consumer sites usually should not, for privacy reasons.
5. Passkeys are discoverable WebAuthn credentials. The FIDO Alliance State of Passkeys report of 7 May 2026 put around 5 billion passkeys in use, with 75 percent of surveyed consumers having enabled at least one and 68 percent of surveyed organizations deploying them.
6. Comparison of second factors, ranked honestly.

Factor	Phishing resistant	Main weakness
SMS code	No	SIM swap, relay
TOTP app	No	Real-time relay
Push approve	No	Fatigue attacks
Number match	Partly	Still relayable
FIDO2 key	Yes	Loss, recovery

7. NIST SP 800-63B revision 4, published 31 July 2025, classifies restricted authenticators and requires that risks of the public switched telephone network be assessed; SMS remains permitted but discouraged.
8. Biometrics are convenient and have real limits. They are not secrets, they cannot be revoked, and matching is probabilistic with a false accept rate and a false reject rate that trade against each other.
9. Apple published a figure of roughly 1 in 50,000 false accept for Touch ID and 1 in 1,000,000 for Face ID, per attempt, with limited retries.
10. The right mental model: on a phone, the biometric unlocks a key stored in secure hardware locally. It is a convenient local gate, not an identity sent over the network.
11. Session management matters as much as login. Rotate session identifiers on privilege change, set `HttpOnly`, `Secure` and `SameSite` on cookies, and give tokens the shortest lifetime the product can tolerate.
12. Account recovery is where most authentication systems actually fail. If a help desk can reset MFA after a phone call, the strength of your authenticator is irrelevant. The 2023 MGM Resorts intrusion is the textbook case.

44.6.6 WORDS - remember these

Authentication — proving who you are — verification of a claimed identity against registered authenticators.

Factor — a category of evidence — knowledge, possession or inherence.

TOTP — the app's six-digit code — time-based one-time password, RFC 6238, derived from a shared secret and the clock.

SIM swap — hijacking your phone number — social engineering a carrier into porting a number to an attacker's SIM.

MFA fatigue — approving to stop the buzzing — repeated push prompts intended to wear the user down.

WebAuthn — the browser standard for hardware login — W3C API binding a signature to a challenge and an origin.

Passkey — a synchronized hardware-grade credential — discoverable WebAuthn credential held by an OS or password manager.

Origin binding — the key checks the address — protocol-level inclusion of the verified origin in the signed data.

44.7 Authorization and access control

44.7.1 PLAIN - in simple words

1. Authentication asks who you are. **Authorization** asks what you may do.
2. They are different questions, they fail differently, and mixing them up causes a large share of real breaches.
3. Being logged in is not permission. Every request must be checked, not just the first one.
4. There are four common ways to decide.
5. **Discretionary access control**: the owner of a thing decides who may use it. This is how files on your laptop work.
6. **Mandatory access control**: a central policy decides, and even the owner cannot override it. This is how classified military systems work.
7. **Role-based access control**: permissions attach to roles, and people get roles. Editor, viewer, administrator.
8. **Attribute-based access control**: a rule engine decides from facts about the user, the resource, the action and the context.
9. Two principles run through all of them.
10. **Least privilege**: every person and every program gets the minimum access needed to do its job, and no more.
11. **Separation of duties**: the person who requests a payment is not the person who approves it.
12. And one classic trap, the **confused deputy**: a program with more privilege than you does something on your behalf that you were not allowed to ask for.

44.7.2 PLAIN - a picture in your head

1. A hospital. Authentication is the badge reader at the door confirming you are Dr Sharma.
2. Authorization is the separate question of which wards, which drug cabinets and which patient records Dr Sharma may open.
3. Discretionary control is a doctor lending their office key to a colleague.
4. Mandatory control is the pharmacy safe, where policy overrides everyone, including the head of the department.
5. Role-based control is the badge saying "consultant", and consultants have a fixed list of doors.

6. Attribute-based control is a rule saying: a doctor may open a record if they are the treating clinician, on shift, in this hospital, today.
7. Least privilege is not giving the night cleaner a master key to the drug store.
8. Separation of duties is two signatures for a controlled substance.
9. The confused deputy is the porter who is trusted to enter any room, being asked by a stranger to fetch a file from a locked office. The porter has the right; the stranger did not.

Where this comparison breaks: a hospital has physical friction, and a human porter might feel that a request is odd. Software has no instinct. A service with broad rights will carry out a strange request a million times a second without hesitating once.

44.7.3 PLAIN – a worked example

1. Here is a real least-privilege event from the running example in this book.
2. The reader's GitHub personal access token could push code, but one push was refused because the change touched a CI workflow file and the token did not carry the `workflow` scope.
3. That refusal is authorization working exactly as designed.
4. The token was authenticated perfectly. GitHub knew whose token it was. It simply was not authorized for that class of change.
5. Notice the security value. If that token had leaked, an attacker could have pushed ordinary code, which review would catch, but could not have silently rewritten the pipeline that builds and deploys.
6. Now a broken example, stated as a pattern rather than a recipe. A web application shows an invoice at a path containing an invoice number, and the server checks that you are logged in but never checks the invoice belongs to you.
7. Changing the number in the address bar then shows somebody else's invoice. That is an insecure direct object reference, and it is the single most common serious web flaw found in real assessments.
8. The fix is not to hide or scramble the number. The fix is that every lookup asks the database for the invoice with that number and that owner.
9. Compare the two designs in one table.

Design	Check performed	Result
Broken	Is user logged in	Any invoice visible
Correct	Is invoice user's	Only own invoices

44.7.4 PLAIN – what is really happening inside

1. Access decisions have a standard shape with two named parts.
2. The **policy decision point** works out the answer. The **policy enforcement point** actually blocks or allows the request.
3. If the enforcement point sits only in the user interface, you have no security at all. Hiding a button does not remove the endpoint behind it.
4. Enforcement belongs on the server, as close to the data as possible.
5. The strongest pattern is to make the check impossible to forget: scope every query by owner at the data layer, so an unscoped query cannot even be written.
6. Row-level security in PostgreSQL, or a mandatory tenant filter in the object-relational mapper, both achieve this.
7. Confused deputy in software, concretely. Your service holds cloud credentials that can read any file in a bucket. A user supplies a file name. Your service reads it and returns it.
8. The user has borrowed your privilege. The service was authorized; the user was not. Server-side request forgery is the same disease with network access instead of file access.
9. The cure is to never let untrusted input select the target of a privileged operation directly. Map user input through a list of allowed values that you control, and check ownership before acting.

10. Capability-based systems attack the problem from the other side. Instead of saying "who are you, and are you allowed", you hold an unforgeable token that already names exactly what you may do.
11. A pre-signed cloud storage URL is a capability. It carries its own authority, is scoped to one object and one action, and expires.

44.7.5 TECHNICAL - the engineer's version

1. The Bell-LaPadula model, published in 1973 by David Bell and Leonard LaPadula, formalized mandatory confidentiality: no read up, no write down.
2. The Biba model, 1977, is its integrity mirror: no write up, no read down.
3. Role-based access control was formalized by David Ferraiolo and Richard Kuhn in 1992 and standardized as ANSI INCITS 359-2004.
4. Attribute-based access control is described in NIST SP 800-162 (2014). XACML is the long-standing policy language; Open Policy Agent with its Rego language is the common modern implementation.
5. The confused deputy problem was named by Norm Hardy in a 1988 paper about a compiler that could be tricked into overwriting its own billing file.
6. In cloud practice: AWS IAM policies, Azure RBAC and Google Cloud IAM are all attribute-flavoured RBAC. The recurring failure is an over-broad wildcard in the action or resource field.
7. OAuth 2.0, RFC 6749 (2012), is a delegation framework, not an authentication protocol. OpenID Connect layers authentication on top of it. Confusing the two produces real vulnerabilities.
8. Scopes in OAuth are coarse permissions granted to an application. They are not a substitute for per-object ownership checks inside your service.
9. Broken Access Control is ranked A01 in the OWASP Top 10 for both the 2021 and 2025 editions, and in the 2025 edition server-side request forgery was folded into it.
10. Practical controls to implement: deny by default, centralize the authorization function rather than scattering conditionals, log every denial, and write tests that assert a user cannot reach another user's objects.
11. Commands and tools to observe: `aws iam simulate-principal-policy` for AWS, `kubectl auth can-i` for Kubernetes, `getfacl` and `ls -l` for POSIX file permissions, `getenforce` and `ausearch -m avc` for SELinux decisions.
12. On Linux, capabilities split the historic all-or-nothing root privilege into pieces such as `CAP_NET_BIND_SERVICE`. Granting only that is far better than running a web server as root to bind port 80.

44.7.6 WORDS - remember these

Authorization — what you are allowed to do — the decision about permitted actions on resources, distinct from identity.

Least privilege — only what the job needs — granting the minimum rights required, for the minimum time.

Separation of duties — two people, two steps — splitting a sensitive operation so no single actor can complete it.

RBAC — permissions through job titles — role-based access control, ANSI INCITS 359-2004.

ABAC — permissions from rules and facts — attribute-based access control, NIST SP 800-162.

Confused deputy — borrowing someone else's power — a privileged component misused by a less privileged caller.

IDOR — changing the number in the URL — insecure direct object reference, a missing per-object ownership check.

Policy enforcement point — where the block happens — the component that actually denies or permits a request.

44.8 The web application vulnerability classes, defensively

44.8.1 PLAIN - in simple words

1. Almost every web vulnerability is one of two mistakes.
2. Mistake one: data supplied by a user was treated as instructions.
3. Mistake two: a check that should have happened did not happen on this path.
4. **Injection** is mistake one against a language: SQL, shell, LDAP, XPath.
5. **Cross-site scripting** is mistake one against a browser: your page ends up carrying somebody else's script.
6. **Cross-site request forgery** is a browser being tricked into sending a request it should not, using cookies it already holds.
7. **Broken access control** is mistake two: the user is logged in but nobody checked the thing belongs to them.
8. **Server-side request forgery** is your server being used as a proxy to reach places the user cannot reach.
9. **Path traversal** is a file name from a user escaping upwards out of the directory you intended.
10. **Insecure deserialization** is rebuilding an object graph from untrusted bytes, which in many languages can run code as a side effect.
11. **XML external entities** is an XML parser being told to fetch files or URLs while parsing.
12. **Mass assignment** is a form binder copying every submitted field onto your object, including the one called `is_admin`.

44.8.2 PLAIN - a picture in your head

1. Imagine a clerk who reads out a sentence you wrote on a form, into a machine that obeys sentences.
2. You write your name. The clerk reads "Ravi" and the machine files it.
3. Someone else writes a name that continues into a new sentence, and the machine obeys that too, because the clerk never knew where the name ended.
4. The bad fix is to inspect names for suspicious words. Attackers write sentences you did not think of.
5. The good fix changes the shape of the conversation. The clerk hands the machine the sentence template and the name through a separate slot.
6. The machine now knows, structurally, that the name is a name. It cannot become an instruction no matter what is written in it.
7. That is exactly the difference between building a query by gluing strings together and using a parameterized query.

Where this comparison breaks: the clerk analogy suggests the danger is only in one direction. In reality the same data may pass through several machines with different rules, a database, then a browser, then a shell, and each needs its own correct handling. Data safe for one is not automatically safe for another.

44.8.3 PLAIN - a worked example

1. The flawed pattern for SQL injection, shown so you can recognize it in a code review. This is the shape to reject, not code to run.

```
# WRONG: query text is built from user input
q = "SELECT * FROM users WHERE email = '" + email + "'"
db.execute(q)
```

2. Because the value is glued into the query text, a value containing a quote character can end the string early and the rest is parsed as query structure. Nothing after that is under your control.
3. The correct version, which removes the class of bug entirely.

```
# RIGHT: structure and data travel separately
q = "SELECT * FROM users WHERE email = ?"
db.execute(q, (email,))
```

4. Why this fixes it **by construction**: the database receives the query structure first and compiles it, then receives the value as a typed parameter. The value can never be reinterpreted as syntax.
5. Escaping and filtering are weaker answers. They depend on getting every edge case right, in every encoding, forever. Parameterization does not.
6. The same principle applies to shell commands. Do not build a command string; pass an argument list to the operating system directly, so the shell never parses your data.
7. The real-world consequence of getting this wrong: the MOVEit Transfer incident of May 2023 began with a SQL injection flaw, CVE-2023-34362, which the Cl0p group used at scale. Public tracking put the total above 2,600 organizations and tens of millions of individuals.

44.8.4 PLAIN - what is really happening inside

1. Cross-site scripting comes in three shapes, and the difference matters for the fix.
2. **Stored**: the attacker's text is saved in your database, and every later visitor receives it. Highest impact.
3. **Reflected**: the text comes in on the request and is echoed straight back in the response. Needs a victim to follow a crafted link.
4. **DOM-based**: the server never sees it. Client-side JavaScript reads part of the page address and writes it into the page.
5. The fix is contextual output encoding. Encode when writing out, not when reading in, because the correct encoding depends on where the value lands: HTML body, attribute, JavaScript string, URL or CSS each differ.
6. Modern frameworks encode by default. The vulnerabilities cluster around the escape hatches, the functions with names like `dangerouslySetInnerHTML` or `v-html`.
7. A **Content Security Policy** header is the second layer. It tells the browser which script sources are allowed, so injected inline script does not execute even if it reaches the page.
8. A strict policy uses per-response nonces or hashes rather than a domain allowlist, because allowlists are routinely bypassed through permissive files on the allowed domains.
9. Cross-site request forgery works because browsers attach cookies for a site automatically, whatever page triggered the request.
10. The modern structural defence is the `SameSite` cookie attribute. `Lax` is the default in current Chrome and Firefox and blocks cookies on cross-site POST requests. `Strict` is stronger and sometimes inconvenient.
11. Keep anti-CSRF tokens as well, because `SameSite` behaviour varies across browsers and versions and does not protect against same-site attackers.

44.8.5 TECHNICAL - the engineer's version

1. The current edition is the **OWASP Top 10:2025**, presented at OWASP Global AppSec on 6 November 2025, built from data across more than 2.8 million applications plus roughly 175,000 CVE-to-CWE mappings.

Rank	2025 category	Change
A01	Broken Access Control	Still first
A02	Security Misconfiguration	Up from 5
A03	Software Supply Chain	New framing
A04	Cryptographic Failures	Down from 2
A05	Injection	Down from 3

2. The remaining five in 2025 are A06 Insecure Design, A07 Authentication Failures, A08 Software or Data Integrity Failures, A09 Security Logging and Alerting Failures, and A10 Mishandling of Exceptional Conditions.
3. Two categories are new or newly framed: Software Supply Chain Failures, and Mishandling of Exceptional Conditions. Server-side request forgery, which had its own slot in 2021, was consolidated into Broken Access Control.
4. Defences by class, stated as the fix a reviewer should insist on.

Class	Fix that works by construction
SQL injection	Parameterized queries
Command injection	Argument arrays, no shell
XSS	Contextual encoding plus CSP
CSRF	SameSite plus tokens
Path traversal	Canonicalize, verify prefix
XXE	Disable DTD and entities
SSRF	Allowlist, block metadata IP

5. Path traversal, CWE-22: resolve the path to its canonical absolute form first, then confirm it starts with the intended base directory. Checking for the literal two-dot sequence before resolving fails against encodings.
6. XXE, CWE-611: disable document type definitions entirely in the parser. In Java that means setting `disallow-doctype-decl` to true; in Python use `defusedxml`. Many parsers changed to safe defaults after 2018, but not all.
7. Insecure deserialization, CWE-502: never deserialize untrusted data into arbitrary types. Use a data-only format such as JSON with an explicit schema. Java native serialization, Python `pickle`, PHP `unserialize` and .NET `BinaryFormatter` are all unsafe on untrusted input by design.
8. Mass assignment, CWE-915: bind to an explicit allowlist of fields, or use a separate data transfer object per endpoint rather than the database model.
9. SSRF, CWE-918: block link-local metadata addresses such as 169.254.169.254, re-validate after every redirect, and resolve the hostname once and connect to that exact address to avoid time-of-check to time-of-use tricks.
10. Log4Shell, CVE-2021-44228, disclosed 9 December 2021 with a CVSS score of 10.0, was an injection flaw in a logging library: attacker-controlled text that reached a log call could trigger a remote lookup. It is the clearest reminder that logging is also an input-handling path.
11. Tools that observe: `zap` and Burp Suite for interactive testing, `semgrep` and CodeQL for static analysis, `sqlmap` only against systems you own or are authorized in writing to test.

44.8.6 WORDS - remember these

Injection — data treated as instructions — untrusted input reaching an interpreter as part of its command structure.

Parameterized query — structure and data kept apart — precompiled statement with typed placeholders, immune to SQL injection.

XSS — someone else's script on your page — cross-site scripting, stored, reflected or DOM-based.

Content Security Policy — a browser rulebook for scripts — response header restricting which sources may execute.

CSRF — your browser sending a request it should not — forged cross-site request riding on ambient cookies.

SameSite — the cookie's travel restriction — cookie attribute limiting sending on cross-site requests, `Lax` by default.

SSRF — your server fetching what the user cannot — server-side request forgery against internal or metadata endpoints.

Mass assignment — the form that set `is_admin` — automatic binding of request fields to object properties without an allowlist.

44.9 Memory safety

44.9.1 PLAIN - in simple words

1. Some languages let a program read and write memory outside the space it asked for. C and C++ are the important ones.
2. They do this on purpose. Bounds checking costs time, and these languages were designed when every cycle mattered.
3. A **buffer overflow** happens when a program copies more bytes into a space than that space can hold, and the extra bytes land on whatever is next.
4. What is next, on the stack, is often bookkeeping: saved registers and the address the function will return to.
5. Overwrite that return address and the processor jumps somewhere the programmer never intended when the function finishes.
6. **Use-after-free** is using a pointer to memory that was already released. The allocator may have handed that space to someone else.
7. **Double-free** is releasing the same block twice, which corrupts the allocator's own bookkeeping.
8. **Integer overflow** is a size calculation wrapping around past the largest representable value, producing a tiny allocation for a huge copy.
9. All four are the same underlying story: the program lost track of how big something is or how long it lives.

44.9.2 PLAIN - a picture in your head

1. Think of a row of numbered pigeonholes on a wall, each one letter wide.
2. You are told to write an eight-letter word into a four-letter hole.
3. Nothing stops you. The last four letters spill into the next holes.
4. Now suppose the hole immediately after yours holds the room number the messenger will visit next.
5. You have just changed where the messenger goes, without touching the messenger.
6. A **stack canary** is a small agreed word placed just before that room number. Before leaving, the messenger checks the word is unchanged. If your spill overwrote it, the program stops.

Where this comparison breaks: real overflows are not always sequential and not always on the stack. Heap corruption works by damaging the allocator's own records, which is subtler and does not pass through a canary at all. And the canary only detects a contiguous overwrite, not a targeted single-word write.

44.9.3 PLAIN - a worked example

1. A stack frame, drawn as it grows downwards in memory on a typical system.

```
    higher addresses
    +---------------------------+
    |  caller's data            |
    +---------------------------+
    |  return address           |  <- overwriting this
    +---------------------------+     redirects execution
    |  saved frame pointer      |
```

```
+----------------------------+
|  stack canary              |  <- checked before return
+----------------------------+
|  local buffer [64 bytes]   |  <- copy starts here
+----------------------------+     and grows upwards
lower addresses
```

2. The copy starts at the bottom of the buffer and moves up towards the canary, the frame pointer and the return address.
3. So a copy of 80 bytes into a 64-byte buffer reaches exactly the fields that control what happens when the function returns.
4. Integer overflow feeding an allocation, shown as a pattern to reject.

```
/* WRONG: count * size can wrap around */
buf = malloc(count * size);
memcpy(buf, src, count * size);
```

5. If `count` times `size` exceeds the maximum of the type, the product wraps to a small number. The allocation succeeds at that small size, and the copy then writes far past it.
6. The correct pattern checks for overflow first, or uses a checked helper such as `calloc`, or uses a language where sizes cannot silently wrap.

44.9.4 PLAIN - what is really happening inside

1. Four defences are standard on modern systems, and each raises cost rather than removing the bug class.
2. **Stack canaries**, from StackGuard in 1998: a random value written before the return address and verified before returning.
3. **Non-executable memory**, called NX or DEP: pages are either writable or executable, not both, so injected data cannot be run as code.
4. Attackers answered with return-oriented programming, which chains together small fragments of code that already exist in the program. Nothing new is injected, so NX does not help.
5. **Address space layout randomization**: load the program and its libraries at unpredictable addresses so the attacker cannot know where anything is.
6. Attackers answer with information leaks that reveal one address, from which the rest follows.
7. **Control flow integrity**: check at run time that every indirect jump goes to a legitimate target. ARM Pointer Authentication and Intel CET are hardware support for this idea.
8. Honest summary: each of these turns a reliable exploit into an unreliable or expensive one. None of them makes the underlying bug disappear.
9. The real fix is a language that cannot express the bug. Rust checks lifetimes and ownership at compile time. Go, Java, C#, Python and JavaScript check bounds at run time and manage memory automatically.
10. That is not free. Garbage collection has pauses, run-time checks cost a few percent, and Rust's rules are hard to learn. But the cost is bounded and the bug class is gone.

44.9.5 TECHNICAL - the engineer's version

1. Scale, from vendor data. Microsoft reported at BlueHat in 2019 that about 70 percent of the CVEs it assigned each year since 2006 were memory safety issues. Google reported a similar proportion for Chrome.
2. Google's Android security team reported that memory safety vulnerabilities fell from about 76 percent of Android vulnerabilities in 2019 to about 24 percent in 2024, as new code was written in Rust and Java rather than C++.
3. Government guidance, with dates. CISA, the NSA, the FBI and international partners published "The Case for Memory Safe Roadmaps" in December 2023.

4. The United States Office of the National Cyber Director published "Back to the Building Blocks: A Path Toward Secure and Measurable Software" in February 2024, explicitly recommending memory-safe languages.
5. The NSA and CISA published "Memory Safe Languages: Reducing Vulnerabilities in Modern Software Development" in June 2025.
6. None of these are laws. They are procurement pressure and expectation setting, and they are already changing what buyers ask for.

Weakness	CWE	Typical impact
Out-of-bounds write	CWE-787	Code execution
Out-of-bounds read	CWE-125	Memory disclosure
Use-after-free	CWE-416	Code execution
Integer overflow	CWE-190	Undersized buffer

7. Heartbleed, CVE-2014-0160, disclosed 7 April 2014, was an out-of-bounds read in OpenSSL's TLS heartbeat handling. It leaked up to 64 kilobytes of process memory per request, including private keys, and it left no trace in ordinary logs.
8. Tools that find these before shipping: AddressSanitizer and MemorySanitizer, UndefinedBehaviorSanitizer, Valgrind, the compiler flags `-fstack-protector-strong` and `-D_FORTIFY_SOURCE=3`, and fuzzing with libFuzzer, AFL++ or OSS-Fuzz.
9. Check a binary's own defences with `checksec --file=./binary` on Linux, or `otool -hv` on macOS. Confirm ASLR is enabled system-wide with `sysctl kernel.randomize_va_space`, where 2 means full randomization.
10. Where experts disagree: whether to rewrite existing C code, or only write new code in safe languages. The rewrite camp cites persistent bug rates; the incremental camp cites the risk of reintroducing logic bugs in mature, heavily tested code. Practice mostly favours the incremental route.

44.9.6 WORDS - remember these

Memory safety — the program cannot stray outside its own memory — guaranteed absence of out-of-bounds and lifetime errors.

Buffer overflow — writing past the end of a space — out-of-bounds write, often CWE-787.

Use-after-free — using something already returned — dereferencing a pointer to released memory, CWE-416.

Stack canary — a tripwire before the return address — random guard value verified before function return.

ASLR — moving the furniture every boot — address space layout randomization.

NX / DEP — memory is writable or runnable, not both — hardware-enforced non-executable pages.

Control flow integrity — checking every jump is legitimate — run-time or hardware validation of indirect branch targets.

Sanitizer — a bug detector built into a test build — instrumentation such as AddressSanitizer that reports violations at run time.

44.10 Network attacks conceptually

44.10.1 PLAIN – in simple words

1. Three network attacks matter for a working engineer.
2. **Man-in-the-middle**: someone sits between you and the server, reading and possibly changing what passes.
3. **Spoofing**: someone lies about an address, so your traffic goes to them instead of the real destination.
4. **Denial of service**: someone sends so much traffic or so many expensive requests that the real users cannot get through.
5. TLS answers the first one. It encrypts, so a listener sees nothing useful, and it authenticates the server by certificate, so an impostor cannot pass.
6. Chapter 32 covered how that certificate check works. The one-line version: the server proves it holds the private key for a name signed by an authority your device already trusts.
7. Spoofing works at two levels. On your local network, **ARP spoofing** lets a machine claim to be the router.
8. On the wider internet, **DNS spoofing** makes a name resolve to the wrong address.
9. Denial of service comes in three flavours: raw volume, connection exhaustion, and expensive requests that cost the server far more than the attacker.
10. **Amplification** is the nastiest volume trick: send a small question with a faked return address to a public server, and it sends a large answer to the victim.

44.10.2 PLAIN – a picture in your head

1. Picture a post room in an office building.
2. ARP spoofing is somebody standing up and repeatedly announcing "I am the mail chute", until everyone believes them and hands over their post.
3. There is no verification step in the announcement, by design. The protocol simply believes the last thing it heard.
4. DNS spoofing is somebody rewriting the internal phone directory, so calls for accounts reach a different desk.
5. Amplification is posting a one-line letter that says "send me your full catalogue", with the victim's return address on it, to ten thousand companies at once.
6. Each catalogue is heavy, none of them is your problem, and all of them arrive at the victim.

Where this comparison breaks: office post is slow and physical, so the flood is visible. On a network the same trick delivers terabits per second within seconds, and there is no time for a human to react. That is why the defences must be automatic.

44.10.3 PLAIN – a worked example

1. Amplification factors, measured by researchers and widely published.

Service	Amplification	Notes
DNS open resolver	about 28 to 54x	Very common
NTP monlist	about 556x	Mostly patched
memcached	up to 51,000x	UDP port 11211

2. Read the last row. A single byte sent can produce tens of thousands of bytes delivered to the victim.
3. That is not theoretical. On 28 February 2018 GitHub absorbed a 1.35 terabit per second memcached amplification attack, the record at the time. It was mitigated within about ten minutes by routing traffic through a scrubbing provider.
4. Scale has moved a long way since. Cloudflare reported mitigating attacks of 22.2 terabits per second in September 2025, 29.7 in October 2025, and 31.4 during December 2025. All were attributed to the Aisuru botnet family, running largely on compromised consumer devices.

5. The December record lasted about 35 seconds. There is no human response time available in that window.
6. Attacks are not only about volume. In October 2023 the HTTP/2 Rapid Reset technique, CVE-2023-44487, produced request-rate records by opening and immediately cancelling streams, costing the server far more than the client.

44.10.4 PLAIN - what is really happening inside

1. ARP has no authentication. A device asks "who has 192.168.0.1", and any machine on that network segment can answer. Most systems accept unsolicited answers and cache them.
2. That is why untrusted local networks matter, and why a VPN or plain TLS everywhere is the practical answer for a laptop in a cafe.
3. DNS cache poisoning aims to get a false answer accepted and cached by a resolver, so every later user of that resolver is sent to the wrong place.
4. Classic DNS used a 16-bit transaction identifier and no signatures, which Dan Kaminsky showed in July 2008 was far too little entropy.
5. The immediate fix was source port randomization, which multiplied the guessing space. The structural fix is DNSSEC, which signs records.
6. The reader's own machine uses 1.1.1.1 as its resolver rather than the router. That reduces exposure to a compromised or mis-configured home router, and it moves trust to Cloudflare instead. It is a trade, not a removal of trust.
7. Encrypted DNS, DNS over HTTPS in RFC 8484 and DNS over TLS in RFC 7858, hides the query from the local network and from the internet provider. It does not prove the answer is correct; only DNSSEC does that.
8. For denial of service, the defences are layered and none works alone.
9. **Rate limiting** by address, by account and by expensive operation.
10. **Filtering** at the network edge, dropping obviously forged or malformed packets before they reach an application.
11. **Anycast**, where the same address is announced from many locations, so an attack is split across dozens of data centres instead of hitting one.
12. **Scrubbing**, where traffic is diverted through a provider that removes attack packets and forwards the rest.
13. And the structural fix nobody can do alone: source address validation at the internet provider, described in BCP 38, published as RFC 2827 in 2000. Amplification only works because forged source addresses still get forwarded.

44.10.5 TECHNICAL - the engineer's version

1. What TLS does and does not do against interception. It authenticates the server's name and protects the channel. It does not protect against a client that has been made to trust an extra certificate authority, which is how corporate interception proxies work, as Chapter 32 described.
2. Certificate Transparency, RFC 6962 and its successor RFC 9162, provides public append-only logs so that mis-issued certificates for your domain can be detected after the fact. Monitoring those logs is a real control.
3. HSTS, RFC 6797, removes the plaintext first request. Preloading puts your domain in a list shipped inside browsers.
4. Layer 2 defences: dynamic ARP inspection and DHCP snooping on managed switches, and 802.1X port authentication for wired and wireless access.
5. DNS defences: DNSSEC validation on the resolver, DNS cookies in RFC 7873, and response rate limiting on authoritative servers.
6. Denial of service categories map to network layers. Volumetric floods saturate bandwidth, protocol attacks such as SYN floods exhaust connection state, and application-layer attacks exhaust processing.

7. SYN cookies, invented by Daniel J. Bernstein in 1996, remove the need to keep state for half-open connections. Enable with `sysctl net.ipv4.tcp_syncookies=1` on Linux.
8. Observation commands: `tcpdump -i any -n` and Wireshark for packet capture, `arp -a` to see the current address table, `dig +dnssec example.com` to check signature records, `mtr` for continuous path measurement, and `ss -s` for socket state counts under load.
9. Historical scale markers, for calibration.

Year	Event	Peak
2016	Mirai, Dyn and OVH	about 1.1 Tbps
2018	GitHub memcached	1.35 Tbps
2025	Aisuru, December	31.4 Tbps

10. Mirai was notable for what it infected: consumer routers and cameras with factory default passwords. Its source code was published in September 2016, which is why variants are still active a decade later.

44.10.6 WORDS - remember these

Man-in-the-middle — someone sitting between you and the server — an active interception attack on a communication channel.

ARP spoofing — claiming to be the router — forged address resolution replies poisoning a local network's address cache.

DNS cache poisoning — putting a lie in the phone book — injecting a forged record that a resolver stores and reuses.

Amplification — a small question, a huge answer — reflection attack exploiting protocols with large response-to-request ratios.

Anycast — one address, many locations — routing technique spreading traffic across distributed sites.

Scrubbing — cleaning the traffic elsewhere — diverting traffic through a provider that filters attack packets.

BCP 38 — stop forged source addresses at the door — RFC 2827 ingress filtering, the structural cure for amplification.

SYN cookies — answering without remembering — stateless response encoding that defeats half-open connection floods.

44.11 Malware and how infection actually happens

44.11.1 PLAIN - in simple words

1. **Malware** is any program written to work against the interests of the person whose machine it runs on.
2. A **virus** attaches itself to another file and spreads when that file is run by a person.
3. A **worm** spreads by itself across a network, needing no human at all.
4. A **trojan** pretends to be something useful so you install it willingly.
5. **Ransomware** encrypts your files and demands payment for the key.
6. **Spyware** watches you: keystrokes, screenshots, messages, location.
7. A **rootkit** hides other malware by lying to the operating system about what files and processes exist.
8. A **bootkit** goes lower still and loads before the operating system does, so the system it lies to never sees the truth.
9. A **botnet** is many infected machines taking orders from one place.
10. A **cryptominer** quietly spends your electricity and processor on mining.

11. **Fileless malware** never writes a program to disk; it lives in memory and in legitimate system tools, so file scanning finds nothing.

44.11.2 PLAIN - a picture in your head

1. Think of a building and the ways an unwanted person gets in.
2. Someone posts a package that a staff member opens: that is a malicious attachment.
3. Someone phones pretending to be head office and gets a door code: phishing.
4. A supplier's cleaning contractor is bribed, and their key still works: supply chain compromise.
5. A window at the back has been unlocked for two years because nobody checked: an unpatched internet-facing service.
6. Someone leaves a labelled envelope in the car park hoping it gets carried inside: removable media.

Where this comparison breaks: a building has one entrance list. A modern company has thousands of accounts, hundreds of cloud services and dozens of suppliers, and no single person knows all of them. The unlocked window is usually one nobody remembered existed.

44.11.3 PLAIN - a worked example

1. Realistic infection routes, ranked by how often they appear in incident reports, with the control that actually helps.

Route	Frequency	Control that helps
Phishing / stolen login	Very high	Phishing-resistant MFA
Unpatched public service	High	Patch within days
Malicious attachment	High	Macro blocking, sandbox
Supply chain	Growing	Pinning, provenance
Drive-by download	Medium	Browser updates
Removable media	Low	Device control policy

2. Ransomware in particular follows a pattern that has barely changed since 2019: get in, look around quietly for days or weeks, steal a copy of the data, find and destroy the backups, then encrypt everything at once.
3. Stealing the data first is deliberate. It creates a second threat, to publish, so that having good backups is no longer a complete answer.
4. Chainalysis reported that tracked ransomware payments fell about 35 percent in 2024, to roughly 813 million dollars, as more victims refused to pay.
5. The FBI Internet Crime Complaint Center reported 20.9 billion dollars in total reported losses for 2025, from more than one million complaints, in its annual report published in April 2026.

44.11.4 PLAIN - what is really happening inside

1. Detection works in three different ways, and each fails differently.
2. **Signatures**: match known bad bytes or known bad file hashes. Precise, no false alarms, and useless against anything new or repacked.
3. **Heuristics**: score suspicious structure, such as packed code that unpacks itself, or a document containing a macro that reaches the network.
4. **Behaviour**: watch what the program does at run time. A process that enumerates files, opens each one, writes back encrypted content and deletes shadow copies is ransomware regardless of what its bytes look like.
5. **EDR**, endpoint detection and response, is the behavioural approach with central recording. It keeps a timeline of process launches, network connections and file changes so an analyst can reconstruct what happened.

6. That recording is often more valuable than the blocking. Without it you cannot answer the only question that matters after an incident: what did they touch?
7. Why antivirus alone is not enough: it is mostly signature and heuristic, it sees one machine, and it cannot judge whether a legitimate administrator tool is being used legitimately.
8. Attackers exploit exactly that gap with **living off the land**: using PowerShell, `certutil`, `wmic`, `rundll32` and remote management tools that are already installed and already trusted.
9. For ransomware, the honest advice is unglamorous. Backups are the only reliable answer, and only if they are tested, versioned, and stored where the production credentials cannot delete them.
10. The rule of thumb is three copies, on two kinds of media, with one held offline or immutable. A backup that a compromised administrator account can erase is not a backup.

44.11.5 TECHNICAL - the engineer's version

1. History markers. The Morris worm of 2 November 1988 infected several thousand machines and led to the founding of the first CERT.
2. ILOVEYOU spread by email in May 2000. Code Red and Nimda hit in 2001. SQL Slammer in January 2003 doubled its population every 8.5 seconds.
3. Stuxnet, discovered June 2010, used four zero-day flaws and stolen code-signing certificates to damage centrifuges. WannaCry, 12 May 2017, and NotPetya, 27 June 2017, both used the EternalBlue SMB exploit; NotPetya caused damages widely estimated above 10 billion dollars.
4. Persistence mechanisms to look for on a compromised host: scheduled tasks, systemd units, launch agents, registry Run keys, WMI event subscriptions, browser extensions, and shell profile files.
5. Firmware and boot-level threats are real but rare. LoJax in 2018 was the first UEFI rootkit found in the wild; BlackLotus in 2023 was the first observed to bypass UEFI Secure Boot on patched systems.
6. Ransomware defensive controls that measurably work: phishing-resistant MFA on all remote access, patching internet-facing services within days, network segmentation so one workstation cannot reach every server, least-privilege service accounts, and immutable backups.
7. Tools that observe: `ps`, `lsof`, `netstat`, Sysinternals Process Explorer and Autoruns on Windows, `osquery` for fleet inspection, YARA for writing your own detection rules, and VirusTotal for reputation checks on hashes.
8. Never analyze suspicious samples on a working machine. Use an isolated virtual machine with no network path back to anything you care about.

44.11.6 WORDS - remember these

Worm — spreads without a human — self-propagating malware exploiting network services directly.

Trojan — useful on the outside, hostile inside — malware disguised as legitimate software.

Ransomware — your files held hostage — encrypting malware, now usually combined with data theft and threatened publication.

Rootkit — malware that hides other malware — code that subverts the operating system's view of files and processes.

Fileless malware — nothing on disk to scan — in-memory execution using existing signed system tools.

EDR — a flight recorder for endpoints — endpoint detection and response, behaviour-based with central telemetry.

Living off the land — using your own tools against you — abuse of legitimate administrative binaries to avoid detection.

44.12 Social engineering

44.12.1 PLAIN - in simple words

1. Most breaches do not start with clever code. They start with a person being helpful.
2. **Phishing** is a message designed to make you click, log in or pay.
3. **Spear phishing** is the same, tailored to you personally, using details from your company website and social media.
4. **Pretexting** is inventing a believable story: a new supplier, an audit, a delivery problem, a colleague locked out before a big meeting.
5. **Business email compromise** is a targeted fraud in which someone with apparent authority instructs a payment or a change of bank details.
6. **Vishing** is the same by voice call. **Smishing** is the same by text message.
7. **MFA fatigue** is pushing approval prompts until someone approves one.
8. Every one of these works by putting a person under pressure and giving them a way to make the pressure stop.
9. The three levers are always the same: authority, urgency, and a plausible reason not to follow the normal process.

44.12.2 PLAIN - a picture in your head

1. Imagine a new employee on their third day.
2. An email arrives from the finance director, who is travelling, about a confidential acquisition, asking for a transfer today, before close.
3. Every element is designed to disable the normal check: seniority, secrecy, time pressure, and a reason the usual approver is unreachable.
4. The employee is not stupid. They are being professional in a situation engineered so that professionalism produces the wrong action.
5. That is why blaming users is both unkind and useless. The attack was designed against the process, using the person as the tool.

Where this comparison breaks: the picture suggests one dramatic moment. Most real social engineering is patient and boring, a few emails over weeks establishing that the sender is a normal part of the working day.

44.12.3 PLAIN - a worked example

1. Real costs, from published figures.
2. The FBI Internet Crime Complaint Center recorded 2.77 billion dollars in business email compromise losses in 2024 and 3.04 billion dollars in 2025.
3. Phishing remained the most-reported crime type in the 2025 report, with 191,561 complaints and 215.8 million dollars in direct losses.
4. The September 2022 Uber intrusion combined a purchased contractor password, repeated push prompts, and a message claiming to be internal IT support.
5. The September 2023 attacks on MGM Resorts and Caesars Entertainment began with phone calls to help desks, persuading staff to reset credentials and MFA for real employees. MGM disclosed an effect of roughly 100 million dollars on quarterly results.
6. Recognition signs worth teaching, in plain language.

Sign	What it looks like
Urgency	Must be done today
Authority	From a senior person
Secrecy	Do not tell the team

Sign	What it looks like
Process bypass	Skip the usual approval
Channel switch	Move to WhatsApp or SMS
New bank details	Same supplier, new account

44.12.4 PLAIN - what is really happening inside

1. Attackers do reconnaissance first: company website, LinkedIn, job adverts, conference talks, code repositories, and out-of-office replies.
2. Job adverts leak the technology stack. Out-of-office replies leak who is away and who covers for them. Both are used directly.
3. Domain look-alikes are cheap. A domain differing by one character, or using a different top-level domain, costs a few dollars and reads correctly to a tired human.
4. Since 2023, generative AI has removed the last easy signal: bad grammar. Convincing text in any language, and cloned voices from short public audio samples, are both established capability now, not speculation.
5. So detection must move away from "does it look wrong" to "is this the normal process".
6. The organizational answers, in order of effectiveness.
7. Phishing-resistant authentication, meaning FIDO2 or passkeys, which removes credential phishing as a category rather than reducing it.
8. Out-of-band verification for money and for access: any change of bank details or any MFA reset is confirmed by calling a number already on file, never a number in the message.
9. A named, blameless reporting route, with a target of reporting in minutes. The measure that matters is time to report, not click rate.
10. Removing single points of authority, so no one person can move money or reset an executive's access alone.
11. Help desk identity proofing: for high-privilege accounts, require a verification step the caller cannot talk their way around, such as a manager callback or an in-person or video check against a record.

44.12.5 TECHNICAL - the engineer's version

1. Email authentication is the technical floor. SPF, RFC 7208, lists which servers may send for a domain. DKIM, RFC 6376, signs messages. DMARC, RFC 7489, ties the two to the visible From address and sets a policy.
2. Publish DMARC with `p=reject` once alignment is confirmed. A monitoring-only policy of `p=none` left in place for years is a common half-measure.
3. These stop exact-domain spoofing. They do not stop look-alike domains or compromised legitimate accounts, which is where most real fraud now lives.
4. Add external sender banners, attachment sandboxing, link rewriting with time-of-click checks, and lookalike-domain monitoring through certificate transparency logs.
5. MITRE ATT&CK identifiers to use in reporting: T1566 phishing, T1566.001 spearphishing attachment, T1598 phishing for information, T1621 multi-factor authentication request generation.
6. Training works only in a specific form: short, frequent, role-relevant, and focused on process rather than on spotting fakes. Punitive simulated phishing programmes reliably reduce reporting, which makes things worse.
7. Measure two things: median time from first delivery to first report, and the proportion of high-risk transactions confirmed out of band.

44.12.6 WORDS - remember these

Phishing — a message engineered to make you act — social engineering by email or messaging, MITRE T1566.

Spear phishing — phishing written just for you — targeted variant using researched personal or organizational detail.

Pretexting — a believable cover story — fabricated scenario used to justify an unusual request.

Business email compromise — fraud by fake authority — payment or data fraud using impersonated or compromised business accounts.

Vishing and smishing — the phone and text versions — voice and SMS social engineering.

Out-of-band verification — confirm on a different channel — validating a request using contact details held independently of the request.

DMARC — the policy that ties email checks together — RFC 7489 alignment and enforcement policy over SPF and DKIM.

44.13 Supply chain security

44.13.1 PLAIN - in simple words

1. Your program is mostly other people's code. A typical application pulls in hundreds of packages, and those pull in more.
2. You trust every one of them, and everyone who can publish an update to any of them.
3. **Dependency confusion**: your build tool asks for an internal package name and a public registry answers first, with a package somebody else uploaded.
4. **Typosquatting**: a package with a name one character away from the one you meant, waiting for a typing mistake.
5. **Compromised maintainer**: a real package, a real author, a stolen account.
6. **Malicious update**: version 3.4.1 is fine and 3.4.2 is not, and your build accepted it automatically because you asked for "the latest".
7. The uncomfortable part is that this attacks the trusted path. The code is signed, the download is over TLS, and the certificate is valid. Everything is correct except the contents.

44.13.2 PLAIN - a picture in your head

1. Think of a kitchen buying ingredients from many suppliers.
2. Typosquatting is a sack labelled almost like your usual flour brand.
3. Dependency confusion is a stranger delivering to your door claiming to be your in-house supplier, and your staff accepting because the name matched.
4. Compromised maintainer is your genuine supplier, whose delivery van was taken over last night.
5. A lock file is writing down the exact batch number you accepted, so tomorrow's delivery must match or be questioned.
6. An SBOM is the ingredient list on the finished product, so when a contamination notice is issued you can tell in minutes whether you used it.

Where this comparison breaks: a kitchen can smell bad flour. Malicious code is designed to behave normally until a condition is met, and often only in the published archive rather than in the source repository everyone reads.

44.13.3 PLAIN - a worked example

1. Four real incidents, with the mechanism and the lesson.

Incident	Year	Mechanism
event-stream	2018	Handed-over maintainership

Incident	Year	Mechanism
SolarWinds	2020	Compromised build system
Codecov	2021	Altered uploader script
xz-utils	2024	Multi-year social engineering

2. event-stream, November 2018: a popular npm package's maintainer handed the project to a volunteer who then added a dependency containing code targeting a specific cryptocurrency wallet application. Downloads were in the millions per week.
3. SolarWinds, disclosed December 2020: attackers compromised the build pipeline for Orion and inserted the SUNBURST backdoor into signed releases. Roughly 18,000 customers installed it; a much smaller number were then targeted. The signature was valid because the build itself was subverted.
4. Codecov, April 2021: a flaw in a Docker image creation process leaked credentials, which were used to modify the Bash Uploader script. The change sat undetected from 31 January to 1 April 2021 and exfiltrated environment variables, meaning secrets, from users' CI runs.
5. xz-utils, March 2024: CVE-2024-3094, CVSS 10.0. An account building trust since 2021 became a co-maintainer and placed a backdoor targeting OpenSSH in release tarballs of versions 5.6.0 and 5.6.1. Andres Freund found it on 29 March 2024, from half a second of extra CPU time on SSH logins.
6. Read that last sentence again. The most sophisticated open-source supply chain attack yet found was caught by one engineer noticing a performance anomaly, not by any scanner.

44.13.4 PLAIN - what is really happening inside

1. Dependency confusion, explained defensively, was published by Alex Birsan in February 2021 after he reached more than 35 companies including Apple and Microsoft, earning over 130,000 dollars in bug bounties.
2. The mechanism is resolution order. Many package managers, given a name, consult both an internal registry and the public one and prefer the highest version number. Publishing a very high version publicly wins.
3. The fixes are configuration, not vigilance: scope internal packages under a namespace you own, configure the client to use exactly one source per scope, and reserve your internal names publicly so nobody else can take them.
4. Lock files record the exact resolved version and a hash of each package's contents. `package-lock.json`, `poetry.lock`, `Cargo.lock`, `go.sum`.
5. Install in continuous integration with the command that refuses to update the lock file, such as `npm ci` rather than `npm install`.
6. Pinning to exact versions removes surprise updates but creates a second duty: something must tell you when a pinned version becomes vulnerable.
7. An **SBOM**, software bill of materials, is a machine-readable list of every component and version in a build. Its value is answering "are we affected" in minutes rather than weeks, which was the actual pain of Log4Shell.
8. Signing and provenance answer a different question: not what is inside, but who built it, from which source, on which machine.

44.13.5 TECHNICAL - the engineer's version

1. SBOM formats: SPDX, an ISO standard as ISO/IEC 5962:2021, and CycloneDX from OWASP. United States Executive Order 14028 of 12 May 2021 pushed SBOMs into federal procurement, which is why they spread.
2. Sigstore, launched in 2021 under the Open Source Security Foundation, provides keyless signing: a short-lived certificate is issued against an OpenID Connect identity, the signature is recorded in the public Rekor transparency log, and the key is discarded. `cosign` is the common tool.

3. SLSA, Supply-chain Levels for Software Artifacts, defines build integrity levels. Level 3 requires a hardened, non-falsifiable build service that produces signed provenance describing exactly what it built and from where.
4. npm provenance, generally available since 2023, publishes signed statements linking a published package to the exact commit and workflow that built it.
5. Reproducible builds mean identical source yields byte-identical output, so independent parties can rebuild and compare. Debian has driven this since 2013 and reports well over 90 percent reproducibility for its main archive.
6. Practical checklist for a team, in priority order: commit and honour lock files; run `npm audit`, `pip-audit`, `cargo audit` or Dependabot in CI; pin base container images by digest, not tag; scope internal packages; generate an SBOM per release; require signed provenance for anything you deploy.
7. Software Supply Chain Failures is A03 in the OWASP Top 10:2025, promoted and broadened from the 2021 category on vulnerable and outdated components.
8. Tools to observe: `syft` to generate an SBOM, `grype` or `trivy` to scan it, `cosign verify` to check signatures, and `osv-scanner` against the Open Source Vulnerabilities database.

44.13.6 WORDS - remember these

Dependency confusion — the public registry answering an internal name — a resolution-order attack on package managers.

Typosquatting — a package named almost like yours — malicious package relying on a typing or memory error.

Lock file — the exact batch numbers you accepted — a record of resolved versions and content hashes for reproducible installs.

SBOM — the ingredient list — machine-readable inventory of components, in SPDX or CycloneDX format.

Provenance — who built it, from what — signed attestation linking an artifact to its source and build process.

Sigstore — signing without long-lived keys — keyless signing with certificate issuance and a public transparency log.

Reproducible build — same source, same bytes — property allowing independent verification that a binary matches its source.

44.14 Defence in depth for a real system

44.14.1 PLAIN - in simple words

1. **Defence in depth** means assuming each control will fail and putting another one behind it.
2. **Network segmentation** divides a network into zones so a compromised laptop cannot reach the database directly.
3. A **firewall** decides which traffic may pass between zones. Default deny, with a short list of allowed paths, is the only sane setting.
4. **Zero trust** means dropping the idea that being inside the network proves anything. Every request is authenticated and authorized on its own merits.
5. **Secure boot** makes the machine refuse to load unsigned early software.
6. A **TPM** is a small chip that stores keys and measurements the main processor cannot forge.
7. **Measured boot** records a hash of each stage into the TPM, so the machine can later prove which software it actually booted.
8. **Disk encryption** protects data on a powered-off device. It does nothing at all once the machine is running and unlocked.

9. **Sandboxing** and containers limit what a compromised process can reach.
10. And the highest-value activity of all, which almost nobody does properly: **patching**, quickly and everywhere.

44.14.2 PLAIN – a picture in your head

1. A castle is the traditional image, and it is the wrong one, because it suggests one wall with everything soft inside.
2. Use a ship instead. A ship has watertight compartments.
3. A hole in one compartment floods that compartment. The ship stays up.
4. Segmentation is the bulkheads. Least privilege is each compartment holding only what belongs in it.
5. Zero trust is the rule that a door between compartments needs a valid reason every single time, not just because you are already aboard.

Where this comparison breaks: bulkheads are physical and permanent. Network segments are configuration, and configuration drifts. The temporary firewall rule opened for a migration in 2022 is still open, and nobody knows why.

44.14.3 PLAIN – a worked example

1. What disk encryption actually protects against, stated honestly.

Situation	Protected?
Laptop stolen, powered off	Yes
Laptop stolen, sleeping	Often not
Malware while you use it	No
Cloud provider reads disk	Yes
Someone with your password	No

2. That table is the whole point. FileVault on macOS, BitLocker on Windows and LUKS on Linux all defend the same narrow, real, valuable case: physical loss of a powered-off device.
3. The key is normally sealed to the TPM or Secure Enclave and released only if measured boot matches. That is why changing firmware settings can trigger a recovery key prompt: the measurements changed, so the seal did not open.
4. Sandboxing and containers, honestly. A container is a process with restricted namespaces and control groups, sharing one kernel.
5. That is a real boundary and a useful one. It is not the same boundary as a virtual machine, because one kernel flaw crosses it.
6. Where isolation must be strong, use virtual machines, or lightweight hypervisor-backed sandboxes such as Firecracker or gVisor.

44.14.4 PLAIN – what is really happening inside

1. Zero trust in practice is three concrete changes, not a product.
2. One: identity is checked at every request, for users and for services alike.
3. Two: device health is part of the decision. Is the disk encrypted, is the operating system current, is the endpoint agent reporting?
4. Three: access is per-application rather than per-network, so being on the VPN does not grant everything on the subnet.
5. Google's BeyondCorp, published from 2014 onwards, is the reference implementation. NIST SP 800-207, published August 2020, is the reference architecture.
6. Patching is the highest-value activity because most successful intrusions use a flaw that was fixed months earlier.

7. Measure two numbers: how long from a fix being published to it being installed on internet-facing systems, and what fraction of your fleet is current. Days and percentages, not intentions.
8. CISA's Known Exploited Vulnerabilities catalogue, started November 2021, is the practical priority list. It contains only flaws with confirmed exploitation, and it is short enough to act on.
9. The reason patching does not happen is never ignorance. It is fear of breakage, and the answer is automated testing and staged rollout, which is an engineering investment, not a security one.

44.14.5 TECHNICAL - the engineer's version

1. Layers, with a concrete control at each.

Layer	Control	Example
Network	Default-deny egress	Firewall, VPC rules
Host	Secure and measured boot	TPM 2.0, UEFI
Process	Mandatory access control	SELinux, AppArmor
Data	Encryption at rest	LUKS, KMS keys

2. UEFI Secure Boot verifies signatures on the bootloader and kernel. Measured boot extends PCR registers in the TPM with hashes of each stage, enabling remote attestation.
3. TPM 2.0 has been an ISO standard as ISO/IEC 11889 since 2015 and is a Windows 11 requirement. Apple's equivalent is the Secure Enclave.
4. Container hardening: run as a non-root user, drop all Linux capabilities and add back only what is needed, set a read-only root filesystem, apply a seccomp profile restricting system calls, and never mount the Docker socket into a container.
5. Kubernetes specifics: Pod Security Admission at `restricted`, network policies that default to deny, and short-lived projected service account tokens rather than long-lived secrets.
6. Egress filtering is underrated. Most malicious activity needs to call out. A default-deny outbound policy with a small allowlist blocks a large share of post-compromise activity and makes SSRF far less useful to an attacker.
7. Secrets belong in a manager such as HashiCorp Vault, AWS Secrets Manager or the cloud equivalent, retrieved at run time, with automatic rotation.
8. Commands to observe: `nft list ruleset` or `iptables -L -n` for firewall state, `tpm2_pcrread` for boot measurements, `cryptsetup status` for LUKS, `docker inspect` for container privileges, and `kubectl auth can-i --list`.

44.14.6 WORDS - remember these

Defence in depth — assume each layer fails — overlapping independent controls so no single failure is fatal.

Segmentation — watertight compartments — dividing networks so compromise does not spread laterally.

Zero trust — being inside proves nothing — architecture where every request is authenticated and authorized on its own merits.

TPM — the small chip that keeps secrets honestly — hardware root of trust storing keys and boot measurements, ISO/IEC 11889.

Measured boot — a signed record of what actually loaded — hashing each boot stage into TPM registers for later attestation.

Egress filtering — controlling what may call out — outbound network policy limiting destinations a host may reach.

Patch window — how long you stay exploitable — time between a fix being published and being installed.

44.15 Detection and response

44.15.1 PLAIN - in simple words

1. You will not prevent everything. So you must be able to notice, and then act, without making it worse.
2. **Logging** is recording what happened. Useful logs answer who, what, when, from where, and did it succeed.
3. A **SIEM** collects logs from everywhere into one searchable place and raises alerts on patterns.
4. **Intrusion detection** watches network traffic or host behaviour for known bad patterns or unusual activity.
5. A **honeypot** is a deliberately attractive fake system. Nobody has a legitimate reason to touch it, so any interaction is a real signal.
6. The response process has six steps, and they are always taught in this order: prepare, identify, contain, eradicate, recover, learn.
7. The first hour matters most, and the most common expensive mistake is rushing to clean up before you understand what happened.

44.15.2 PLAIN - a picture in your head

1. Think of a shop after a break-in.
2. The instinct is to sweep up the glass and reopen. That destroys the evidence and tells you nothing about how they got in.
3. The professional sequence is: stop the bleeding, photograph everything, then clean.
4. Containment is boarding up the broken window. Eradication is changing every lock they might have copied.
5. Recovery is reopening. Learning is the meeting where you find out the alarm had been silenced for three months.

Where this comparison breaks: a burglar leaves. An intruder in a network is often still present, watching your response, including your incident channel. That is why serious responses move to an out-of-band communication channel.

44.15.3 PLAIN - a worked example

1. The first hour of a suspected breach, in order.

```
 0-10 min  Believe it. Start a timestamped written log.
10-20 min  Assemble: security, an owner who can decide, legal.
20-30 min  Contain: isolate hosts from the network,
           but DO NOT power them off. Memory is evidence.
30-40 min  Preserve: snapshot disks, export logs to a
           separate account the attacker cannot reach.
40-50 min  Revoke: rotate credentials, tokens, sessions,
           starting with anything highly privileged.
50-60 min  Scope: what did that identity touch, and when?
           Decide on notification duties and timers.
```

2. Two rules inside that hour. Do not power a machine off if memory may hold evidence; isolate the network interface instead.
3. Do not discuss the incident on the systems you suspect are compromised. Move to phones or a separate messaging service.
4. What good logging looks like, as fields rather than prose.

Field	Example
Who	user or service identity

Field	Example
What	action and target object
When	UTC, ISO 8601, synced
From	source address, device

5. Add the outcome, success or failure, and a request identifier that follows the request across services. Without that identifier you cannot reconstruct anything in a distributed system.

44.15.4 PLAIN - what is really happening inside

1. Log what matters and only what matters: authentication successes and failures, privilege changes, access to sensitive data, configuration changes, and outbound connections to new destinations.
2. Never log secrets, passwords, full card numbers, tokens or session identifiers. Logs are widely readable and widely backed up.
3. Ship logs off the machine immediately. A log stored only on the compromised host is a log the attacker can edit.
4. Retention is a real decision. Median attacker dwell time before detection has fallen over the last decade but is still measured in days to weeks, so thirty days of logs is often not enough to answer basic questions.
5. Detection tuning is the hard part. An alert that fires fifty times a day is not an alert; it is background noise that teaches people to ignore it.
6. Honeypots and honeytokens are cheap and high signal. A fake credential placed in a document that nobody should use, wired to an alert, tells you the moment someone is looking.
7. Preparation is the step that actually determines outcome: contact lists, decision authority, pre-approved isolation actions, a written plan, and one rehearsal per year.
8. Rehearse with a tabletop exercise. Read a scenario aloud and make people say what they would do. It reliably exposes the missing phone number, the unclear authority, and the backup nobody has tested restoring.

44.15.5 TECHNICAL - the engineer's version

1. The six-phase cycle is NIST SP 800-61, currently revision 3, published April 2025, which restructures incident response around the Cybersecurity Framework 2.0 functions. The older SANS six-step naming remains common.
2. Detection technologies: network intrusion detection with Suricata or Zeek, host detection with Wazuh or OSSEC, and endpoint detection and response agents for process-level telemetry.
3. Detection rules are shared as content: Sigma for log rules, YARA for file and memory patterns, and Suricata rules for network traffic.
4. Log pipelines in common use: the Elastic Stack, Splunk, Grafana Loki, or a cloud-native pairing such as CloudWatch with Athena. The important property is a separate trust boundary from the systems being logged.
5. Time synchronization is not optional. Without NTP and a single time zone, usually UTC, correlation across systems is guesswork.
6. Forensic preservation order, from most to least volatile: CPU registers and caches, memory, network state, running processes, disk, then archived logs. Capture in that order.
7. Notification timers start whether or not you are ready. GDPR Article 33 requires notifying a supervisory authority within 72 hours of becoming aware. India's DPDP Rules 2025 require intimation without delay and a fuller report to the Data Protection Board within 72 hours.
8. Tools to observe and preserve: `journalctl`, `auditd` with `ausearch`, `dmesg`, Velociraptor for remote forensics, and `dd` or cloud snapshot APIs for disk images. Record hashes of every image you take.

44.15.6 WORDS - remember these

SIEM — one searchable place for all logs — security information and event management with correlation and alerting.

IDS — a watcher for known bad patterns — intrusion detection system, network or host based.

Honeypot — bait with no legitimate use — deliberately exposed decoy whose every interaction is a signal.

Dwell time — how long they were inside unnoticed — interval between initial compromise and detection.

Containment — stopping the spread without destroying evidence — isolating affected systems while preserving volatile state.

Eradication — removing their access completely — eliminating footholds, credentials and persistence mechanisms.

Tabletop exercise — a rehearsal with no machines — discussion-based drill testing decisions and contacts.

44.16 Privacy and the law, briefly

44.16.1 PLAIN - in simple words

1. Security is about stopping people who should not have access. Privacy is about limiting what you collect and what you do with it, even when you have every right to hold it.
2. You can be perfectly secure and badly privacy-invading at the same time.
3. **Personal data** is anything relating to an identified or identifiable person. That includes identifiers such as device numbers and addresses, not only names.
4. The core ideas of modern data protection law are short.
5. Have a lawful reason to process. Tell people plainly. Collect only what you need. Keep it only as long as you need it. Keep it secure. Let people see, correct and delete their data.
6. **Data minimization** is the one engineers control directly. Every field you do not collect is a field that cannot leak.
7. If a breach exposes personal data, you usually have a legal duty to tell a regulator, and often the affected people, on a clock measured in hours.

44.16.2 PLAIN - a picture in your head

1. Security is the lock on the filing cabinet.
2. Privacy is the decision about which forms go into the cabinet in the first place, who may open it and why, and when the papers are shredded.
3. A shop with an excellent safe that photographs every customer and keeps the images forever is secure and privacy-invading.
4. A shop that photographs nobody has nothing to lose in a burglary.

Where this comparison breaks: paper is finite and visible. Digital data is copied silently, spreads into backups, analytics systems and third-party tools, and the copy in the log aggregator is usually the one nobody deleted.

44.16.3 PLAIN - a worked example

1. Two regimes a working engineer meets most often.

Point	GDPR	India DPDP
In force	25 May 2018	Act 2023, Rules 2025
Breach to regulator	72 hours	72 hours

Point	GDPR	India DPDP
Top penalty	20M EUR or 4%	Rs 250 crore
Regulator	National authorities	Data Protection Board

2. GDPR fines have two tiers: up to 10 million euro or 2 percent of worldwide annual turnover for most obligations, and up to 20 million euro or 4 percent for the most serious. The higher of the fixed amount and the percentage applies.
3. India's Digital Personal Data Protection Act was passed in August 2023. The Digital Personal Data Protection Rules were notified on 14 November 2025, with the substantive obligations, including breach reporting and security safeguards, phased in over about eighteen months from that date.
4. Under the DPDP Act, failure to take reasonable security safeguards carries a penalty of up to 250 crore rupees, and failure to notify a breach up to 200 crore rupees.
5. Under the DPDP Rules, affected individuals must be informed without delay, and a fuller report reaches the Data Protection Board within 72 hours.

44.16.4 PLAIN - what is really happening inside

1. The engineering consequences of these laws are concrete, not abstract.
2. You need a data inventory: what personal data exists, where it lives, who can reach it, and how long it stays. Most organizations cannot answer this.
3. You need deletion that actually works, including in backups, search indexes, analytics and every third-party processor.
4. You need to answer access requests, which means being able to find one person's data across systems.
5. You need to record consent where consent is your lawful basis, including when it was given, for what, and how it can be withdrawn.
6. Pseudonymization replaces identifiers with tokens but remains personal data, because it can be reversed with the mapping table.
7. Anonymization means the person can no longer be identified by anyone, and it is much harder than it looks. Combining a few "anonymous" fields such as postcode, birth date and sex re-identifies most individuals.
8. The cheapest compliance strategy is the same as the cheapest security strategy: do not collect it, and if you must, delete it on a schedule that runs automatically.

44.16.5 TECHNICAL - the engineer's version

1. GDPR is Regulation (EU) 2016/679, applicable from 25 May 2018. Article 5 sets the principles, Article 6 lawful bases, Articles 12 to 22 data subject rights, Article 25 data protection by design and by default, Article 32 security of processing, Articles 33 and 34 breach notification.
2. Article 33 requires notification to the supervisory authority within 72 hours of becoming aware, unless the breach is unlikely to result in risk. Article 34 requires telling the individuals when the risk is high.
3. The largest single GDPR fine to date is 1.2 billion euro against Meta in May 2023, concerning transfers of European user data to the United States.
4. Other regimes worth knowing by name: the California Consumer Privacy Act as amended by CPRA, Brazil's LGPD, and sector rules such as HIPAA for United States health data and PCI DSS, currently version 4.0.1, for card data.
5. PCI DSS is a contractual standard, not a law, but it is enforced through card scheme agreements and behaves like one in practice.
6. Technical measures named in practice: encryption at rest and in transit, pseudonymization, access logging, retention policies enforced by automation, and documented data processing agreements with every processor.
7. Privacy and security overlap at Article 32 but are not the same duty. A system can satisfy every security control and still be unlawful because it collects data it has no basis to collect.

44.16.6 WORDS - remember these

Personal data — anything about an identifiable person — includes indirect identifiers such as device and network addresses.

Data minimization — do not collect what you do not need — principle limiting collection to what is adequate and relevant.

Breach notification — the legal clock after an incident — duty to inform regulators and individuals, typically within 72 hours.

Pseudonymization — identifiers swapped for tokens — reversible with a separately held mapping, still personal data.

Anonymization — genuinely no longer a person — irreversible, and much harder than it looks in practice.

Lawful basis — your legal reason to process — one of the grounds required before processing personal data.

Data protection by design — build it in, not on — the requirement to embed protection into system design by default.

44.17 Security for the individual developer

44.17.1 PLAIN - in simple words

1. Most of what protects you is a short list of habits, not expensive tools.
2. Keep your machine current. Turn on automatic updates for the operating system and the browser.
3. Encrypt the disk. FileVault, BitLocker or LUKS, with a real passphrase.
4. Use a password manager, and put a hardware key or passkey on the accounts that matter most: email first, then code hosting, then cloud.
5. Email first, because email resets everything else.
6. Never commit a secret. Not a token, not a password, not a private key, not a connection string.
7. Assume that anything committed once is compromised forever, even after you delete it, because history is public and cloned within minutes.
8. Keep your dependencies current and know what they are.
9. Read code with security in mind, which is a specific skill, not general attentiveness.
10. And if you find a flaw in somebody else's software, report it responsibly rather than publishing it or exploiting it.

44.17.2 PLAIN - a picture in your head

1. Think of a workshop. Locked door, tools accounted for, dangerous materials in a cabinet rather than on the bench.
2. Committing a secret is leaving the spare key taped to the outside of the door, then painting over it and believing it is gone.
3. Automated scanners crawl public repositories continuously. Published measurements repeatedly show credentials being found and used within minutes of being pushed.
4. So the moment of exposure is not when someone reads your code. It is when the commit lands.

Where this comparison breaks: a taped key can be removed. A committed secret is in every clone, every fork, every cache, and every scraper's archive. The only real remedy is revocation and rotation, not deletion.

44.17.3 PLAIN - a worked example

1. Practical checklist, grouped, in the order worth doing it.

Area	Action
Machine	Auto-update, disk encryption
Accounts	Manager, passkey on email
Secrets	Manager, never in git
Dependencies	Lock files, audit in CI
Code	Review checklist, static scan

2. Secrets handling in practice. Keep values in environment variables loaded from a file that is listed in `.gitignore`, or in a real secret manager.
3. Commit a `.env.example` with the names and no values, so the next person knows what is needed.
4. Install a pre-commit hook that blocks obvious secret patterns, and enable push protection on the hosting side as a second net.
5. If a secret is exposed, the order is fixed and not negotiable.

```
1. Revoke the credential immediately.
2. Issue a replacement and deploy it.
3. Check logs for use of the old one.
4. Only then clean the history, if worth it.
```

6. Rewriting history is the last step and the least important. The credential is already public; rotating it is what actually ends the exposure.

44.17.4 PLAIN - what is really happening inside

1. Reading code for security means asking a different set of questions from normal review.
2. Where does untrusted input enter, and where does it end up? Trace it from the boundary to every interpreter it touches.
3. Is authorization checked on every path to this object, including the new endpoint added last week?
4. Are errors handled on every branch, and does the failure path fail closed rather than open?
5. Is anything constructed by string concatenation that should be parameterized: SQL, shell commands, HTML, file paths, URLs?
6. Is any cryptography home-made, and is any random value coming from a non-cryptographic generator?
7. Do log statements or error messages contain secrets or personal data?
8. Two kinds of tool help, and they see different things.
9. **Static analysis** reads code without running it. It finds patterns and scales to every file, at the cost of false positives.
10. **Dynamic analysis** runs the system and observes it. It finds real behaviour including configuration problems, but only on the paths it exercises.
11. **Fuzzing** is dynamic analysis with automatically generated inputs, and it is the most effective way to find memory safety and parsing bugs.
12. None of these replaces a person who understands what the application is supposed to protect.

44.17.5 TECHNICAL - the engineer's version

1. Secret scanning: `gitleaks`, `trufflehog`, GitHub secret scanning with push protection, and `detect-secrets` as a pre-commit hook.
2. Static analysis: `semgrep` with community rule packs, CodeQL, `bandit` for Python, `gosec` for Go, `cargo clippy` for Rust lints, `brakeman` for Rails.

3. Dependency scanning: `npm audit`, `pip-audit`, `cargo audit`, `osv-scanner`, Dependabot or Renovate for automated update pull requests.
4. Dynamic analysis: OWASP ZAP for web scanning, `nuclei` for templated checks, `trivy` for container and infrastructure-as-code scanning.
5. Fuzzing: libFuzzer and AFL++ for native code, `go-fuzz` and Go's built-in fuzzing since Go 1.18, Atheris for Python, and OSS-Fuzz for open-source projects that qualify.
6. Coordinated disclosure, the correct process when you find a flaw in software you do not own.
7. Look for a `security.txt` file at `/.well-known/security.txt`, standardized as RFC 9116 in April 2022, or a `SECURITY.md` in the repository, or a bug bounty programme.
8. Report privately with enough detail to reproduce. Propose a disclosure timeline. Ninety days is the widely used norm, following Google Project Zero's policy since 2014, with a shorter clock if active exploitation is observed.
9. Do not test against systems you do not own or have written authorization for. In many jurisdictions, including under India's Information Technology Act 2000 and the United States Computer Fraud and Abuse Act, unauthorized access is a criminal offence regardless of intent.
10. Request a CVE identifier through the vendor if they are a CVE Numbering Authority, or through MITRE otherwise. Coordinate publication with the fix.
11. If the vendor does not respond, escalate to a national CERT, such as CERT-In in India or CISA in the United States, rather than publishing.
12. Keep a written record of every contact attempt with dates. If disclosure ever becomes contested, that record is what protects you.

44.17.6 WORDS - remember these

Secret scanning — automated hunting for committed credentials — pre-commit and server-side detection of key patterns.

Rotation — replacing a credential — issuing a new secret and invalidating the old one, the only real remedy for exposure.

Static analysis — reading code without running it — pattern and dataflow analysis over source, SAST.

Dynamic analysis — testing the running system — behavioural testing against a deployed instance, DAST.

Fuzzing — throwing generated inputs at it — automated input generation to find crashes and parsing flaws.

Coordinated disclosure — report privately, publish together — agreed process between finder and vendor with a timeline.

security.txt — where to send a vulnerability report — RFC 9116 file at a well-known path listing security contacts.

44.18 Famous breaches and what each teaches

44.18.1 PLAIN - in simple words

1. Six incidents, chosen because they are well documented and because each fails in a different way.
2. Read them as a list of things that were possible, not as a list of villains.
3. In every one of them, the technical flaw was known and fixable beforehand.

44.18.2 PLAIN - a picture in your head

1. Think of aviation accident reports. Nobody reads them for blame.
2. They are read because one crew's bad day becomes every other crew's checklist item.
3. Security has the same discipline available and uses it far less often.

Where this comparison breaks: aviation has a legal duty to publish investigations. Most breach reports are written by lawyers, and the technically interesting details are usually the ones removed.

44.18.3 PLAIN - a worked example

1. The six, with root cause and lesson.

Incident	Year	Root cause
Target	2013	Supplier access, flat network
Equifax	2017	Unpatched Apache Struts
WannaCry	2017	Unpatched SMB, no segmentation
SolarWinds	2020	Compromised build pipeline
Colonial Pipeline	2021	VPN password, no MFA
Change Healthcare	2024	Citrix portal without MFA

2. Target, December 2013: credentials belonging to a refrigeration contractor reached the payment network because the network was flat. About 40 million card records and 70 million customer records were exposed.
3. Equifax, 2017: a fix for Apache Struts flaw CVE-2017-5638 was available on 7 March 2017. Intrusion began around 13 May and was discovered on 29 July. About 147 million people were affected; the settlement was up to 700 million dollars.
4. WannaCry, 12 May 2017: ransomware using the EternalBlue SMB exploit, for which Microsoft had shipped a patch on 14 March 2017. It reached about 200,000 machines in 150 countries and disrupted parts of the NHS.
5. SolarWinds, disclosed December 2020: the build system was subverted, so the malicious code carried a valid vendor signature. Roughly 18,000 customers installed the trojanized update.
6. Colonial Pipeline, 7 May 2021: a legacy VPN account with a leaked password and no multi-factor authentication. Fuel supply to the United States east coast was disrupted and a 4.4 million dollar ransom was paid.
7. Change Healthcare, February 2024: a Citrix remote access portal without multi-factor authentication. It became the largest health data breach ever reported in the United States, at 192.7 million individuals, with a reported 22 million dollar ransom payment.

44.18.4 PLAIN - what is really happening inside

1. The lessons, one line each, in the same order.
2. Target: a supplier's access is your access. Segment the network so a contractor's account cannot see a payment terminal.
3. Equifax: the gap between a patch existing and a patch being installed is the whole vulnerability. Measure that gap in days.
4. WannaCry: a patch you did not deploy is a patch you did not have. And a flat internal network turns one infection into all of them.
5. SolarWinds: a valid signature proves who built it, not that the build was honest. Protect the build system as tightly as production.
6. Colonial Pipeline: every remote access path needs multi-factor authentication, including the old one nobody uses.
7. Change Healthcare: the same lesson again, three years later, at a much larger scale. This is the single most repeated root cause in the record.
8. The pattern across all six is dull and consistent. Known flaw, known fix, missing basic control, insufficient segmentation.
9. Not one of them required a novel attack technique.

44.18.5 TECHNICAL - the engineer's version

1. Detection times are the other lesson. Equifax ran for roughly 76 days before discovery. The Codecov modification ran for 60 days. The SolarWinds implant ran for months before Mandiant identified it in December 2020.
2. Notable data-handling failures worth naming separately, because they are about storage rather than intrusion.
3. RockYou, December 2009: 32 million passwords stored in plain text. The resulting list is still the standard wordlist in password auditing tools.
4. LinkedIn, June 2012: 6.5 million password hashes leaked, SHA-1 with no salt. The full set later proved to be about 117 million accounts.
5. Adobe, October 2013: about 153 million records, passwords encrypted with 3DES in ECB mode rather than hashed, with plaintext password hints alongside. Identical passwords produced identical ciphertext.
6. Yahoo, breaches in 2013 and 2014 disclosed in 2016 and 2017: 3 billion accounts, with MD5 hashing on a portion of them.
7. Each of those is a direct illustration of section 44.3. Plain text, fast unsalted hashes, and encryption used where hashing was required.
8. For breach data, the checkable public sources are the Have I Been Pwned service, the annual Verizon Data Breach Investigations Report published each spring since 2008, and the United States Department of Health and Human Services breach portal for health-sector figures.

44.18.6 WORDS - remember these

Root cause — the thing that actually let it happen — the earliest correctable failure, not the final symptom.

Dwell time — how long they stayed unnoticed — interval from compromise to detection, often weeks.

Lateral movement — spreading sideways once inside — using one foothold to reach further systems.

Flat network — everything can reach everything — an unsegmented network where one compromise exposes all.

Patch gap — the window between fix and install — the period during which a known, fixed flaw remains exploitable.

Third-party risk — their weakness becomes yours — exposure arising from suppliers, contractors and integrations.

44.98 Common wrong ideas

1. Wrong: we are too small to be a target. Right: most attacks are automated and indiscriminate. Scanners find every internet-facing service within hours, and ransomware crews prefer small organizations because they pay faster and have weaker controls.
2. Wrong: the padlock in the browser means the site is safe. Right: it means the connection is encrypted to whoever holds a valid certificate for that name. A phishing site can obtain one for free in minutes. It says nothing about who runs the site or what they do with your data.
3. Wrong: antivirus protects me. Right: signature-based scanning misses new, repacked and fileless malware, and cannot judge whether a legitimate administrative tool is being misused. It is one useful layer among several, not a boundary.
4. Wrong: open source is less secure because anyone can read the code. Right: visibility helps defenders more than attackers, since attackers can analyze closed binaries too. The real open-source risk is different: unpaid, unsupported maintainers of critical components, which is exactly what the xz-utils backdoor exploited in 2024.

5. Wrong: a strong password is enough. Right: length and randomness do not help if the password is phished, reused on a breached site, or stolen by malware. Phishing-resistant multi-factor authentication is the control that changes the outcome.
6. Wrong: encryption solves security. Right: encryption protects confidentiality in transit and at rest. It does nothing about broken access control, injection, social engineering, or an attacker using your own application while it is unlocked and running.
7. Wrong: we passed a penetration test, so we are secure. Right: a test is a sample of one moment by one team within a fixed scope. It shows what was found, not what exists.
8. Wrong: internal systems do not need the same protection. Right: that assumption is exactly what turned the Target and WannaCry incidents into catastrophes. Assume the attacker is already inside and design accordingly.
9. Wrong: hashing a password with SHA-256 is fine because SHA-256 is secure. Right: SHA-256 is secure as a hash and terrible as a password function precisely because it is fast. Use Argon2id, scrypt or bcrypt.
10. Wrong: we have backups, so ransomware cannot hurt us. Right: modern ransomware finds and deletes backups first and steals the data before encrypting, so publication remains a threat. Backups must be tested, versioned and beyond the reach of production credentials.

44.99 Chapter summary in 20 lines

1. Security is five promises about data: confidentiality, integrity, availability, authenticity and non-repudiation, always in trade-off.
2. It is a property of a whole system in a context, never a product you buy.
3. Threat modelling is three questions: what am I protecting, from whom, and what happens if I fail. Rank the answers by likelihood times impact.
4. A hash is a one-way fingerprint. MD5 fell to collisions in August 2004 and SHA-1 to the SHAttered attack on 23 February 2017. Use SHA-2, SHA-3 or BLAKE3, and never a bare hash for passwords.
5. Store passwords with a deliberately slow, memory-hard function: Argon2id at 19 MiB with two passes, scrypt, or bcrypt at cost 10 or above, always with a unique per-user salt and ideally a separately stored pepper.
6. NIST SP 800-63B revision 4, published 31 July 2025, forbids composition rules and routine expiry. Length beats complexity, and a manager beats both.
7. Symmetric encryption must be authenticated. Use AES-GCM or ChaCha20-Poly1305, never ECB, and never reuse a nonce with the same key.
8. Asymmetric cryptography solves key agreement and gives signatures, which prove origin and integrity in a way encryption alone never does. Every real system is hybrid: asymmetric once, symmetric thereafter.
9. NIST approved ML-KEM, ML-DSA and SLH-DSA as FIPS 203, 204 and 205 on 13 August 2024, and selected HQC as a fifth algorithm on 11 March 2025. Harvest-now-decrypt-later makes key exchange migration urgent today.
10. Authentication factors are not equal. SMS is weak, app codes are relayable, and only FIDO2 and passkeys resist phishing by binding to the origin.
11. Authorization is a separate question from authentication, checked on every request, at the data layer, with least privilege and deny by default.
12. Broken Access Control has been number one in the OWASP Top 10 in both the 2021 and the current 2025 edition, which was presented on 6 November 2025.
13. Injection is fixed by construction with parameterized queries and argument arrays, never by filtering. Cross-site scripting is fixed by contextual output encoding plus a strict Content Security Policy.
14. Memory safety bugs have been roughly seventy percent of the vulnerabilities at major vendors. Canaries, non-executable memory, ASLR and control flow integrity raise cost; memory-safe languages remove the class.

15. Network defence rests on TLS against interception, DNSSEC and encrypted DNS for name lookups, and rate limiting, filtering, anycast and scrubbing against floods that now exceed 30 terabits per second.
16. Most intrusions begin with a person or a stolen credential, not an exploit. Business email compromise alone cost 3.04 billion dollars in 2025.
17. Your dependencies are your attack surface. Use lock files, pin by digest, generate SBOMs, and require signed provenance. Remember that xz-utils in March 2024 was caught by one engineer noticing half a second of CPU time.
18. Defend in depth: segment, patch fast, encrypt disks, sandbox, and give everything the least privilege it can function with.
19. Prepare for the incident you will have. Log the right fields off-host, rehearse once a year, and in the first hour contain without destroying evidence, preserve, revoke and scope before you clean.
20. Every famous breach in section 44.18 had a known flaw and an available fix. The advantage is almost never in exotic defence; it is in doing the dull things quickly and everywhere.

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