

Lesson 01: The IT Map

Lesson 01 — The IT Map

A tour of what IT people work on — and why nobody can do all of it.

IT Landscape

Module 1

Why you're here

This course is built for three audiences:

- **Career switchers** — you're thinking about moving into tech and want a map before the deep dive.
- **Non-tech managers** — you work next to engineers and want to follow the conversation.
- **Motivated relatives of techies** — you want to understand what they actually do all day.

Goal: leave this course knowing which area you'd want to learn more about.

IT is a city

Think of IT as a city.

- **Frontend** = storefronts
- **Backend** = warehouses
- **Networking** = roads
- **Databases** = libraries
- **Security** = police
- **Cloud** = the utility grid

Every business runs on the whole city. Most engineers live in one or two neighborhoods.

The 7 neighborhoods

1. **Frontend** — what the user clicks: web pages, mobile apps.
 2. **Backend & APIs** — the brain behind the storefront.
 3. **Databases & Data** — where state lives.
 4. **Networking & The Internet** — how anything talks to anything else.
 5. **DevOps & Cloud** — how code gets to real users, 24×7.
 6. **Security** — keeps everyone else's mistakes off the front page.
 7. **AI / ML** — the newest neighborhood, re-shaping the rest.
-

What "writing software" actually looks like

A typical engineer's day:

- Reading existing code
- Writing new code (a smaller share than you'd think)
- Reviewing teammates' code
- Fixing bugs
- Designing how things connect
- Meetings about all of the above

Not "typing furiously into a black screen."

The tool landscape

Names you'll meet in any modern team:

- **GitHub** — where the code lives.
- **VS Code** — the editor most people open.
- **Slack** — where the team talks.
- **Jira** — where the work is tracked.
- **AWS Console** — where the cloud bill lives.
- **ChatGPT / Claude** — the AI assistant in every workflow.
- **Docker** — the box code ships in.
- **Terraform** — infrastructure described as text.

Most IT workers see most of these **every week**.

Languages aren't religions

Python · JavaScript · Go · TypeScript · Rust · SQL

A tool, not a tribe. Most engineers use 2–4 languages daily and pick up new ones over a career.

What an "engineer" actually does

~5%

Writing brand-new code.

~95%

Reading code, debugging, reviewing pull requests, designing how the pieces fit, and talking to other humans.

Roles you'll hear named

- **Software Engineer** — the generalist who ships features.
- **DevOps / SRE** — keeps the production system alive.
- **Data Engineer** — builds the pipelines that move data.
- **Data Scientist** — turns data into answers.
- **ML Engineer** — puts models into production.
- **Security Engineer** — keeps everyone honest.
- **Product Manager** — decides what gets built.
- **UX Designer** — decides how it feels to use.

Lesson 14 expands on every one of these.

The 2026 reality

- **Every team uses cloud.** Owning your own servers is now the exception.
 - **Every team has AI assistants in the loop.** Code, docs, reviews, customer support — touched by an LLM.
 - **"Full-stack solo" is back.** AI handles a lot of glue work, so one person can ship more on their own than five years ago.
-

How to read this course

- **12 lessons**, ~45 min each.
 - **Slides** = the talk.
 - **Theory** = the read.
 - **Homework** = the tool-tour.
 - **No code.** The goal is vocabulary + a mental model, not job-readiness on day one.
-

Recap

- IT is many fields, not one.
- Engineers spend most of their time reading and designing — not typing.
- You don't need to learn it all to be useful.

Coming up: Lesson 02 — what actually happens when you visit a URL.

The IT Map

Reading time: ~10 min · Prereqs: none

Why this matters to you

Over the next eleven lessons you'll meet IT one neighborhood at a time — the internet, the cloud, how software is built, how it runs, what AI is doing to all of it. Before any of that, you need a flyover. A map.

Most people who aren't in IT have a fuzzy picture: "there are programmers, and they make apps, and somehow there's a cloud." That's not wrong, but it's not enough to follow a real conversation at work, or to figure out which corner of this world you'd actually want to learn more about. By the end of this lesson you'll have the vocabulary to follow the rest of the course, and a mental map of where each topic fits. No code, no jargon without translation, no pretending it's simpler than it is.

1. IT as a city

The single best analogy for the modern IT landscape is a city. Imagine you walk into a busy downtown for the first time. You see **storefronts** — that's the *frontend*, the part the public touches. Behind each storefront sits a **warehouse** full of staff, inventory, and paperwork — that's the *backend*, where the real work of fulfilling a request happens. Goods move between buildings on **roads** — that's the *network*. Records live in **libraries** — *databases*. **Police and locks** keep all of it honest — *security*. **City services** — water, power, garbage — quietly run underneath — *DevOps*. And the whole city sits on a **utility grid** someone else owns and bills you for — the *cloud*.

Every real business runs on this whole city. But almost no engineer works on all of it. Most are specialists in one or two neighborhoods, with a passing familiarity with the others.

2. The seven neighborhoods

Here's the city, neighborhood by neighborhood. We'll spend lessons on the most important ones; this is your one-paragraph orientation to each.

Frontend is everything the user touches: web pages, mobile apps, the buttons and forms and animations. The languages of this neighborhood are HTML, CSS, and JavaScript (plus mobile-specific ones like Swift for iOS and Kotlin for Android). Frontend gets the most public attention because it's literally on screen — if a website looks ugly or glitches, everybody notices.

Backend and APIs are the brain behind the storefront. When you click "Buy", the frontend sends a request to a backend somewhere; the backend checks your account, talks to a payment service, updates the database, and sends back an answer. Backend engineers spend their days in languages like Python, Go, TypeScript, Java, and C#.

Databases and data are where state lives — the customers, the orders, the messages, the analytics. There are two big families: *SQL* databases like Postgres and MySQL (structured, predictable) and *NoSQL* like MongoDB and DynamoDB (flexible, scale-friendly). The cost of getting this layer wrong is usually the highest in the whole stack.

Networking and the internet is how anything talks to anything else: DNS, IP addresses, HTTPS, certificates, load balancers. Lesson 02 deep-dives this neighborhood — it's the one whose pieces are most likely to appear in the news as outages.

DevOps and cloud is how code goes from a laptop to running for real users at 3 a.m. on a Tuesday. Pipelines, containers, infrastructure as code, monitoring. Lessons 03 and 08 deep-dive this.

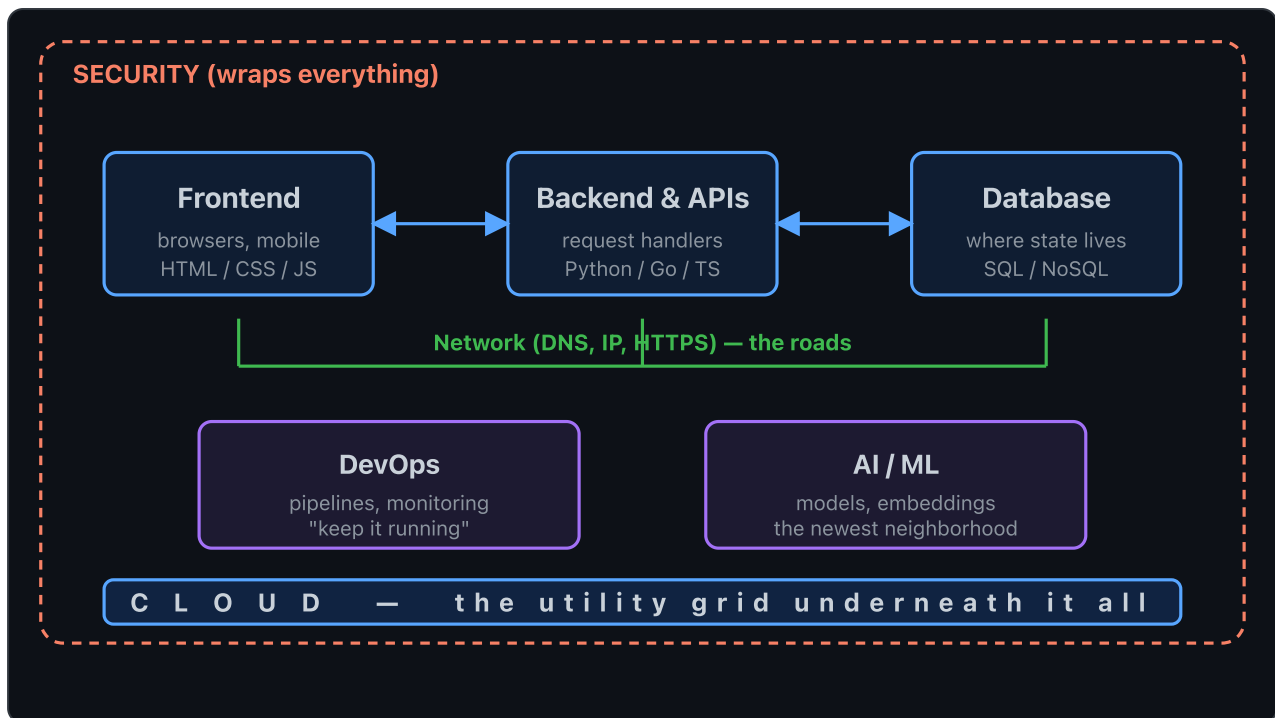
Security is the neighborhood whose job is to keep everyone else's mistakes from becoming front-page news. Cryptography, authentication, vulnerability scanning, incident response. Lesson 11.

AI and ML is the newest neighborhood and the one re-shaping every other one. Module 4 (lessons 10 and 11) covers it in detail.

3. The map, drawn

Here's the same idea as a picture. The seven neighborhoods sit inside a perimeter labelled "Security" (because security wraps around everything, not under it), with the

cloud as the substrate underneath:



4. Roles, briefly

Once you can see the city, you can place the people. **Software Engineer** is the generic term — it can mean any neighborhood, or several. **Frontend Engineer** lives in the storefront, **Backend Engineer** in the warehouse plus the library. **DevOps Engineer** or **SRE** (Site Reliability Engineer) keeps the city services running — pipelines, deployments, monitoring, on-call.

Data Engineer builds the plumbing that moves data into the libraries; **Data Scientist** reads from the libraries to find patterns; **ML Engineer** ships those patterns as actual services that run in production. **Security Engineer** watches the perimeter (and gets called when something gets through it). **Product Manager** talks to every neighborhood, builds none of it, and is responsible for deciding what gets built in what order. **UX Designer** shapes how the storefronts feel.

In real life the lines blur. A "backend engineer" at a 10-person startup may also write some frontend, set up the CI pipeline, and design the database schema. A "frontend engineer" at a 10,000-person company may spend half a year on a single performance optimisation. Lesson 14 revisits this in detail.

5. What “writing code” really looks like in 2026

Here's the part that surprises non-engineers most: *engineers do not spend most of their time typing new code*. A typical day looks more like this. You open the laptop, read three Slack messages, skim a pull request someone left for review overnight, leave a few comments. You pick up a bug ticket, spend forty minutes reading the existing code to figure out where the bug lives. You write maybe twenty lines to fix it. You spend another hour writing a test, running it, watching it fail, fixing the fix, watching it pass. You attend two meetings — one design discussion, one standup. You read someone else's design document and write a paragraph of feedback. You go home.

In 2026, AI assistants — Claude, Copilot, Cursor, Windsurf — write a large slice of the lines that actually get committed. The engineer's job has shifted toward *directing, reviewing, integrating, and owning* the result. The myth of the lone genius typing furiously into a black terminal is dead. The reality is closer to a senior editor at a newsroom, except the staff reporters are very fast and need careful supervision.

What people argue about

Three debates you'll hear in any IT room this year:

- **Will AI replace engineers?** Some say within five years; others say never. Reality so far: AI assistants are making existing engineers measurably faster but have not, in most teams, reduced headcount. Junior-engineer hiring is the part most under pressure.
- **Microservices vs monolith.** One camp argues every new system should be split into many small services from day one. The other camp argues you should start with one well-organised codebase and split only when scale forces you to. Lesson 06 returns to this.
- **Full-stack vs specialist.** The pendulum is swinging back toward full-stack — AI assistants close knowledge gaps, so one engineer can plausibly own frontend, backend, and infrastructure for a small product. The deep-specialist career is far from over, but the generalist is having a moment.

Reading on

- **The Pragmatic Programmer** — Hunt & Thomas (20th anniversary edition). The classic introduction to software as a craft. Reads well even if you never plan to write code.
- **Roadmap.sh** — community-maintained role-by-role learning roadmaps. A great "what does a Frontend / Backend / DevOps engineer actually study?" reference.
- **Lesson 14 of this course** — revisits roles and how to enter the field, with current 2026 hiring patterns.

LAST-REVIEWED: 2026-05-22

Homework — The IT Map

Estimated time: ~45 min · No code required

Three short tool-tour exercises. Goal: get your hands on the actual surfaces IT people use every day — not to learn them deeply yet, but to confirm they exist and to start recognising the vocabulary from this lesson in the wild. Take screenshots and jot a few notes as you go — future lessons will keep referring back to what you saw here.

How to do these

Open each linked site in a new tab. No installs, no accounts (except as explicitly called out), no money. If a step asks for something you can't find in 30 seconds, write down "couldn't find <thing>" and move on — that's a perfectly valid observation about modern UIs.

Task 1 — Map yourself onto the seven neighborhoods

From the theory page, list the seven neighborhoods of IT in your own notes: Frontend, Backend & APIs, Databases & Data, Networking & The Internet, DevOps & Cloud, Security, AI / ML.

Now, in one sentence each:

1. Pick the **neighborhood you already use most** day-to-day and write one sentence (e.g., *"I use Frontend every day as a Notion user."* or *"I use Databases every time I run a report in our CRM."*).
2. Pick the **two neighborhoods you most want to understand** by the end of this 12-lesson course, and write one sentence each on *why* — what conversation at work or what question in life you'd be able to engage with if you understood them.

✓ **Self-check: you can name all 7 neighborhoods from memory (close this page and try); for at least 3 of them, you've named one specific tool, app, or**

person at work that lives in that neighborhood.

Task 2 — Tool tour: open GitHub

Open github.com. **Don't** sign up — you don't need an account for this.

1. In the search box at the top, type `react` and press Enter. Open the top result (it should be `facebook/react`).
2. On the repo page, find and write down:
 - how many **stars** the project has (top-right area);
 - roughly how many **contributors** (look in the right sidebar, "Contributors" section);
 - when the **last commit** was (just above the file list — "latest commit" with a relative timestamp).
3. Scroll down to the **README** (the long block of text below the file list). Read the first 2 paragraphs.
4. Click the **Issues** tab. Notice the count and skim the titles of the top 3 open issues. (You don't need to understand them — just notice they exist.)

✓ **Self-check:** you can name 3 things you saw on the repo page (e.g., "*stars badge*", "*README*", "*Issues tab*", "*file tree*", "*branch selector*"); you can write one sentence about *what the project does* in your own words; you have a rough sense of whether this project is *actively maintained* based on the last-commit date.

Task 3 — Tool tour: read a real job listing

Open linkedin.com/jobs or indeed.com. Search for "**Software Engineer**" (or, if you prefer, "*Data Engineer*", "*DevOps Engineer*", or "*Security Engineer*" — any of the role names from the theory page).

1. Pick **one listing** from a company you've heard of.
2. Scroll to the **Requirements** (or "*Qualifications*" / "*What you'll need*") section.

3. Highlight (or copy into your notes) **5 technologies or tools** mentioned — e.g., `Python`, `AWS`, `React`, `PostgreSQL`, `Kubernetes`, `Terraform`.
4. For each of the 5, write one sentence answering: *which neighborhood from this lesson does it belong to?* (frontend / backend / database / network / DevOps / security / AI). If you don't recognise a term, write "*don't know — look up in lesson XX*" — that's the vocabulary the rest of this course will give you.

✓ **Self-check: you've identified 5 specific technologies from a real job posting; for at least 3 of them, you can confidently say which neighborhood they live in; you have a written list of the unfamiliar terms to revisit as the course progresses.**

What "done" looks like

You finish this homework when you have, in your notes:

- The 7 neighborhoods listed, with your "*already use*" and "*want to learn*" picks (Task 1).
- A short description of the `facebook/react` repo and 3 page elements you noticed (Task 2).
- A real job listing's URL plus 5 technologies mapped to their neighborhoods (Task 3).

That's it — you've now seen the two surfaces (GitHub + a real job description) you'll keep bumping into for the rest of this course, and you've started building your own internal map of the IT city.

Lesson 02: What the Internet Actually Is

Lesson 02 — What the Internet Actually Is

DNS, IP, HTTPS, browsers, CDNs — what happens between pressing Enter and the page appearing.

IT Landscape

Module 1

The question

*You type **wikipedia.org** and hit Enter.
What happens?*

We'll answer it in 11 slides.

Step 1: Your computer needs an address

- `wikipedia.org` is a **name**.
- Computers talk in **numbers** — IP addresses like `208.80.154.224`.
- Translating names ↔ numbers is what **DNS** does.

DNS = the phonebook of the internet.

DNS in one diagram

Browser → **Recursive resolver** (e.g. 8.8.8.8, 1.1.1.1)

↓

Root → **.org authoritative** → **wikipedia.org authoritative**

↓

IP returned (208.80.154.224)

Each step is cached for seconds-to-days — usually you only hit the resolver.

Step 2: Open a connection

- Your computer opens a **TCP connection** to that IP.
- Port **443** — the standard port for HTTPS.

TCP = the dial tone of the internet. It's the reliable, in-order byte stream every web app rides on top of.

Step 3: Wrap it in TLS

- **HTTPS = HTTP + encryption.**
- The browser asks for a **certificate**.
- The certificate must be signed by a **Certificate Authority** the browser trusts (Let's Encrypt, DigiCert, ...).
- The two sides derive a shared key. Everything after is encrypted.

 **Padlock icon** = encryption + verified identity.

Step 4: Send the HTTP request

```
GET / HTTP/2  
Host: wikipedia.org  
Accept: text/html  
User-Agent: Mozilla/5.0 ...
```

This is — literally — what your browser sends down the encrypted pipe.

Step 5: The server responds

```
HTTP/2 200 OK
Content-Type: text/html
Content-Length: 84211

<html>...</html>
```

200 OK = "here you go". Other common ones: **301** redirect, **404** not found, **500** server error.

CDNs: the same page, served from near you

Wikipedia's origin lives in a couple of data centers. You don't.

- A **CDN** (Content Delivery Network) keeps copies at **edge nodes** all over the world.
 - You talk to the closest copy — maybe 50 ms away, instead of 200 ms.
 - Brands: **Cloudflare**, **Akamai**, **AWS CloudFront**, **Fastly**.
-

The browser renders

1. Parses the HTML.
2. Fetches CSS, JavaScript, images, fonts — **each one is another HTTP request**.
3. Runs the JavaScript.
4. Paints pixels.

One "page load" is often **hundreds of requests** in the background.

Why this matters when something breaks

- **DNS issue** → this one site unreachable, other sites still work.
- **Certificate expired** → padlock warning, scary red page.
- **Slow page** → could be DNS, your network, the CDN, the origin server, or the page's own JavaScript. Any of them.

Knowing the layers turns "the internet is broken" into a question you can actually narrow down.

What this course won't teach you

Honest list of things we're *not* covering:

- **BGP** — how ISPs route between each other.
- **IPv6** details and dual-stack realities.
- The **TLS handshake math**.
- **TCP congestion control**.

They exist, they matter, they're someone's full-time job. See the *Reading on* list in the theory page.

Recap

- **Name** → **number** = DNS.
- **Encrypted call** = HTTPS (TCP + TLS + HTTP).
- **CDN** = a "near you" copy of the page.

Coming up: Lesson 03 — what "the cloud" actually sells.

What the Internet Actually Is

Reading time: ~10 min · Prereqs: Lesson 01

Why this matters to you

Every IT job, every modern product, every "the WiFi is broken" conversation touches the internet at some point. The thing is — almost nobody outside IT can describe what actually happens when you type a URL and press Enter. Five different machines you've never heard of are involved before the page paints. Once you know the basic flow, a huge slice of everyday tech problems — "this site is slow", "this site is down", "my browser shows a padlock warning", "VPN broke everything" — goes from "magic gone wrong" to "I can guess which layer is at fault". That diagnostic instinct is the entire payoff of this lesson.

1. Names vs numbers: DNS

Computers on the internet talk in **IP addresses** — numeric tags like `208.80.154.224` (IPv4) or `2620:0:861:ed1a::1` (IPv6). People, sensibly, prefer `wikipedia.org`. **DNS** — the Domain Name System — is the global phone book that translates names into numbers.

When you press Enter, your browser asks itself "have I looked up this name recently?" If yes (a cached answer), it skips ahead. If no, it asks the operating system, which asks a *recursive resolver* — usually your ISP, your router, or a public resolver like Cloudflare's `1.1.1.1` or Google's `8.8.8.8`. The resolver does the legwork on your behalf: it asks a **root server** "who knows about `.org`?", then asks that **top-level-domain server** "who knows about `wikipedia.org`?", then asks that **authoritative server** "what's the IP for `wikipedia.org`?" The answer comes back the way it came, and every step along the way caches it for a while — that's the **TTL** (time-to-live) on each record.

The recursive-resolver step is where a lot of real-world plumbing lives. Corporate filtering, Pi-hole ad-blocking on your home network, parental controls, and country-

level censorship typically intercept DNS here. If someone tells you "switch your DNS to 1.1.1.1", they're changing which resolver does the looking-up on your behalf.

2. The phone call: TCP + TLS

Now your browser has an IP. Next it needs to open a conversation. **TCP** — Transmission Control Protocol — is the reliable byte-stream protocol of the internet. Think of it as the dial tone: it doesn't care what you say, only that everything you send arrives in order, with nothing lost. Your browser opens a TCP connection to that IP on **port 443**, the universal port for HTTPS traffic.

Then **TLS** layers on top — that's the "S" in HTTPS, Transport Layer Security. The browser asks the server for its *certificate*, validates that it was signed by a **Certificate Authority** (CA) the browser trusts — brands like Let's Encrypt, DigiCert, Sectigo, GlobalSign — and then the two sides do a brief cryptographic handshake to derive a shared symmetric key. From that moment on, everything in the conversation is encrypted. The little padlock icon in the URL bar means two things at once: TLS succeeded, *and* the certificate's name matches the hostname you typed.

3. The conversation: HTTP

Inside the encrypted tunnel, the actual conversation is **HTTP** — Hypertext Transfer Protocol. It's a simple request/response pattern. Your browser sends something like:

```
GET / HTTP/2
Host: wikipedia.org
Accept: text/html
User-Agent: Mozilla/5.0 ...
```

The server replies with a status code, some headers, and the body:

```
HTTP/2 200 OK
Content-Type: text/html
Cache-Control: max-age=3600

<html>...
```

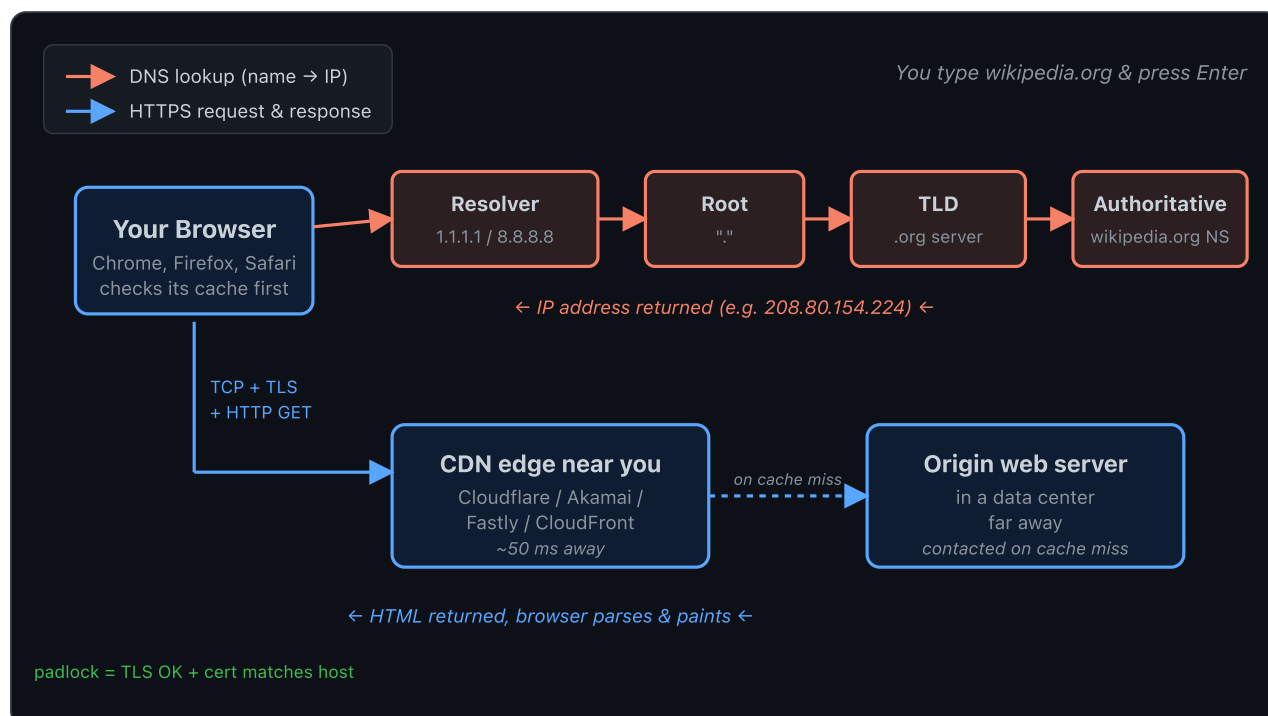
Methods you'll hear named: **GET** (read), **POST** (create / submit), **PUT** (replace), **DELETE** (remove). Status codes: **200** OK, **301** permanent redirect, **404** not found, **500** generic server error, **504** upstream timeout. HTTP/2 and HTTP/3 are newer wire-format upgrades (smaller, faster, multiplexed) but the request/response model is unchanged from the original 1990s design.

4. The "near you" trick: CDNs

A real website is hosted in maybe one or two data centers. Its readers, meanwhile, are spread across every continent. If every request for Wikipedia had to fly from Madrid to San Francisco and back, the web would feel like 1998. The fix is the **CDN** — Content Delivery Network. A CDN keeps cached copies of a site at hundreds of *edge locations* close to users. When you request Wikipedia from Madrid, you talk to a CDN node sitting in Madrid — about 50 ms away — instead of the origin server 9000 km west. The big names here are **Cloudflare**, **Akamai**, **Fastly**, **AWS CloudFront**, **Google Cloud CDN**, and the smaller-but-cheap **Bunny**.

5. The whole flow, drawn

Here's everything in one picture. Orange arrows = DNS lookup ("what's the IP?"). Blue arrows = the actual encrypted content fetch.



6. What happens when things break

Once you have the picture, common symptoms map almost mechanically to layers of the flow:

- **"One site won't load, others work."** Likely the DNS record for that site, or that site's CDN edge in your region. Not your internet.
- **"All sites are slow."** Local network or ISP. Your DNS lookups are slow, your TCP connections are slow, everything downstream is slow.
- **"Padlock warning."** The TLS certificate is expired, self-signed, or doesn't match the hostname. Could also be a real man-in-the-middle — rare but the warning is exactly the same.
- **"Page spins, then 504."** The server you're talking to is up — it answered with 504 — but *its* upstream dependency timed out. Common in micro-service architectures.

What people argue about

Three current debates inside the networking world:

- **HTTP/3 over QUIC vs HTTP/2 over TCP.** QUIC (the protocol under HTTP/3) is faster on lossy mobile networks and eliminates head-of-line blocking. It's also slower to roll out, harder for corporate middleboxes to inspect, and adoption is still patchy — many CDNs and clients support it, many enterprise proxies don't.
- **Centralisation of DNS resolvers.** Cloudflare's [1.1.1.1](#) and Google's [8.8.8.8](#) now answer a huge share of the world's DNS queries. Critics argue this is a privacy concentration risk and a single point of pressure for censorship orders. Supporters argue these resolvers are faster, more private (DNS-over-HTTPS), and more reliable than most ISP resolvers.
- **CDN consolidation.** When Cloudflare or AWS CloudFront has a bad day, a startling slice of the public internet has a bad day with them. The 2024 and 2025 outages re-opened the question of whether this dependence is fixable, or just the price of modern web performance.

Reading on

- **High Performance Browser Networking** — Ilya Grigorik. Free online at hpbn.co. The canonical readable deep-dive on everything in this lesson, plus the parts we skipped.
- **Cloudflare Learning Center** — learning.cloudflare.com. Short, well-illustrated explainers for every term in this lesson (DNS, TLS, CDN, BGP, DDoS, and more). Great when you want a second angle.
- **Julia Evans' *dns-zines*** — short, friendly, accurate comic-style explainers of how DNS actually works. Excellent companion to section 1 of this lesson.

LAST-REVIEWED: 2026-05-22

Homework — What the Internet Actually Is

Estimated time: ~45 min · No code required

Three short tool-tour exercises. Goal: turn the abstract picture from the theory page (DNS → TCP+TLS → HTTP → CDN) into something you've actually *seen* on your own machine. You'll inspect your DNS resolver, watch a real page load request-by-request in your browser, and read a TLS certificate. Take a screenshot at each step.

How to do these

Open each linked site in a new tab. No installs and no accounts required. Tasks 2 and 3 are easiest in **Chrome**, **Edge**, or **Firefox** on a desktop / laptop — mobile browsers don't expose dev-tools the same way. If a step doesn't match what you see exactly, write down what's *different* — browsers update their UIs constantly.

Task 1 — See your own DNS

First we'll find out which DNS resolver your computer is currently using, and what the public internet sees as your IP address.

1. Open dnsleaktest.com. Click the **"Standard test"** button. After a few seconds you'll see a list of one or more DNS server IPs and the owning *ISP / provider* (e.g., [Google](#), [Cloudflare](#), your home ISP, your VPN provider).
2. Write down: the DNS server IP(s) and the owning provider name.
3. Now open ipinfo.io. Write down: your **IP address**, your **ISP** (the "org" line), and your **approximate city**.
4. Bonus (optional): if a friend or family member is in a different country, ask them to do the same two steps and compare. Different country → different ISP → almost certainly a different resolver.

✓ **Self-check:** you can answer *"what DNS resolver am I using right now?"* with a specific name (e.g., *"Cloudflare 1.1.1.1"* or *"my home ISP's resolver"*); you

can answer *"is my IP a residential or VPN IP?"* based on the ISP / org line; you understand that **both** answers can change depending on the network you're on (home vs. cafe vs. office vs. VPN).

Task 2 — Inspect a real request

Now we'll watch a real page load, request-by-request, using the browser's built-in dev-tools.

1. Open wikipedia.org in Chrome, Edge, or Firefox.
2. Right-click anywhere on the page → **Inspect** (or press `F12` / `Cmd-Opt-I` on macOS, `Ctrl-Shift-I` elsewhere).
3. Switch to the **Network** tab in the panel that appears.
4. **Reload the page** (`Cmd-R` on macOS, `Ctrl-R` elsewhere). You'll see a list of every individual request the browser made — HTML, CSS, JS, images, fonts.
5. Sort the list by **Size** (click the column header). Find the **largest single request**. Click it.
6. In the side panel that opens, look at the **Headers** tab. Identify:
 - the HTTP **method** (GET, POST, ...);
 - the response **status code** (200 OK, 304 Not Modified, ...);
 - the `Content-Type` response header (e.g., `image/jpeg`, `text/html`, `application/javascript`);
 - approximately **how many total requests** the page made (look at the bottom-left summary line — *"N requests"*).
7. Take a screenshot of the Network tab.

✓ **Self-check:** you can name (a) the HTTP method, (b) the status code, (c) the `Content-Type` of the largest request, and (d) the rough total number of requests a real page makes (usually dozens to hundreds — not one). You now know where to look the next time someone says *"the page is slow"* — the **Network** tab is the first place to check.

Task 3 — Trace the certificate

Finally, we'll read the TLS certificate that makes the padlock icon appear in your URL bar.

1. Still on wikipedia.org, click the **padlock icon** in the URL bar.
2. Click "Connection is secure" → "Certificate is valid" (Chrome / Edge wording) or "More information" → "View Certificate" (Firefox). A certificate viewer opens.
3. Identify and write down:
 - the **Issuer** — this is the Certificate Authority (CA) that signed it (e.g., "Let's Encrypt", "DigiCert", "GlobalSign");
 - the "**Valid until**" / "**Not After**" date — this is when the certificate expires;
 - the **Subject** or **Common Name** — the hostname(s) this certificate is valid for (often a wildcard like `*.wikipedia.org` or a SAN list).
4. Repeat on a smaller site — your bank, your school, a local shop's site. Notice the differences: different CAs are common; some banks still use **EV (Extended Validation)** certificates where the legal entity name appears in the certificate detail; expiry windows vary (Let's Encrypt = 90 days, others = 1 year).

✓ **Self-check:** for both sites you inspected, you can answer (a) *who issued the cert*, (b) *when it expires*, (c) *for which hostname(s) it's valid*. You understand that the padlock means "*the connection is encrypted AND the cert's name matches the URL bar*" — **not** "*this site is trustworthy*."

What "done" looks like

You finish this homework when you have, in your notes:

- The DNS resolver IP + provider you're currently using, plus your public IP and ISP from `ipinfo.io` (Task 1).
- A screenshot of the Network tab on a real Wikipedia load, plus the method / status / Content-Type / total-request count of the largest single request (Task 2).

- The CA, expiry date, and hostname(s) for the Wikipedia certificate, plus the same three for one other site (Task 3).

That's the whole DNS → TCP+TLS → HTTP chain from the theory page, observed once on your own machine. The next time a site is slow, broken, or warns you about a certificate, you'll know which of these three layers to look at first.

Lesson 03: The Cloud, Demystified

Lesson 03 — The Cloud, Demystified

AWS, GCP, Azure — what they actually sell, and why every team has a bill.

IT Landscape

Module 1

What "the cloud" is, in one sentence

Renting computers, storage, and services from someone else's data center — by the minute.

That's it. No mystical fog. Just **somebody else's machines, billed by usage.**

Who runs "the cloud"?

The **big three** hyperscalers:

- **AWS** (Amazon Web Services) — ~31% market share.
- **Microsoft Azure** — ~25%.
- **Google Cloud** (GCP) — ~12%.

Plus runners-up: **Oracle Cloud**, **Alibaba**, **IBM**, **Cloudflare**.

Note: share figures shift annually — see [refresh-checklist.md](#).

What they sell, layer by layer

The stack

- **SaaS** — finished apps
Gmail, Slack, Figma
- **PaaS** — managed building blocks
databases, queues, runtimes
- **IaaS** — raw infrastructure
compute, storage, network

Different teams care about different layers.

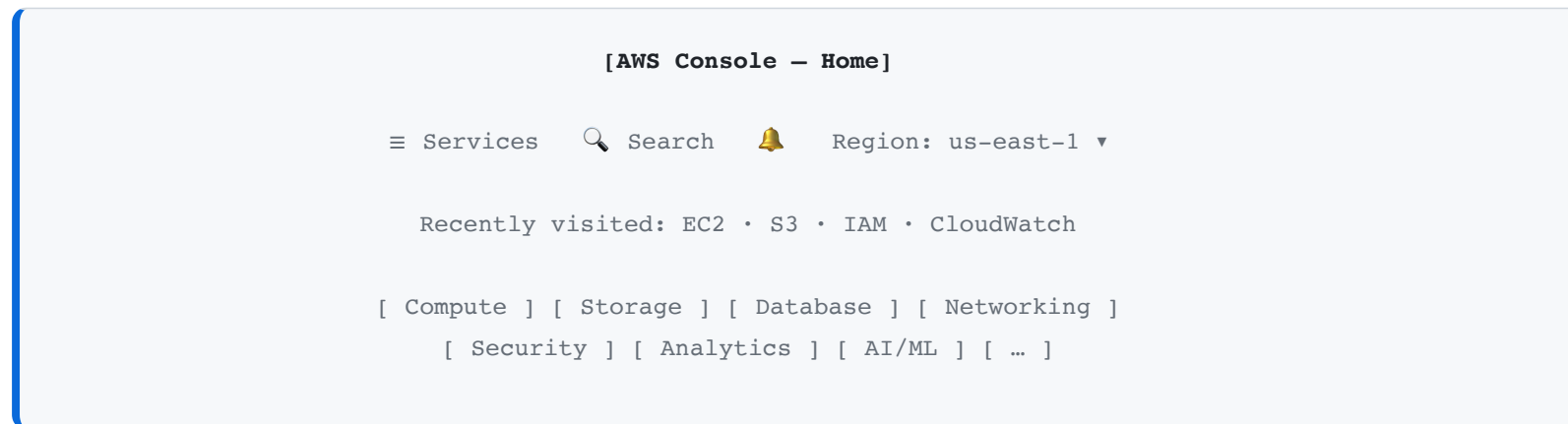
An ops engineer lives in IaaS. A backend dev lives in PaaS. Everyone uses SaaS.

The flagship products

Provider	Compute	Storage	Database	Functions
AWS	EC2	S3	RDS	Lambda
GCP	Compute Engine	Cloud Storage	Cloud SQL	Cloud Run
Azure	Virtual Machines	Blob Storage	Azure SQL	Functions

Don't memorize. **Recognize.** You'll meet these names in every job listing.

What the console looks like



Hundreds of services in one menu. **Most teams use 5–10.**

Refresh: screenshot subject to refresh — see [refresh-checklist.md](#) L03.

The pricing model that surprises everyone

- By the **second** (or **millisecond** for Lambda).
- By the **GB-stored** — per month.
- By the **GB-transferred out** (egress!).
- By the **API call**.

You can leave one EC2 instance running and burn \$200 in a month without noticing.

Free tiers exist

Provider	Free for	Plus
AWS Free Tier	12 months	Always-free tier
GCP Free Tier	90 days + \$300 credit	Always-free tier
Azure Free Account	12 months + \$200 credit	Always-free services

Cap = enough to **learn**. Not enough to host a real product.

"Cloud-native" means...

Cloud-native

Built **for** cloud:

- Auto-scaling
- Ephemeral storage
- Managed services everywhere
- Pay only when running

Lifted in

An old app moved **into** cloud VMs:

- Still behaves like on-prem
- Gets the bill, not the elasticity
- "Lift-and-shift"

Most new companies are cloud-native. Most enterprises are mid-migration.

Where the cloud is going in 2026

- **AI services baked in** — AWS Bedrock, GCP Vertex AI, Azure AI Foundry. The cloud is now also an AI-model marketplace.
 - **More "serverless" / managed everything** — fewer VMs, more "give us your code, we'll run it".
 - **Multi-cloud reality** — most companies use 2+ providers, intentionally or accidentally.
-

Why this matters to non-IT readers

This is where **the bill** lives.

- Cloud cost is now a **top-3 line item** in many companies.
- "FinOps" (Cloud Financial Operations) is a real job title.
- Even non-engineers in the org get pulled into cloud-cost conversations.

Finance, procurement, product — all now read AWS invoices.

Recap

- **Cloud = rented computers**, billed by the second.
- **Three big providers** (AWS, Azure, GCP) plus niche ones.
- **Pricing surprises matter** — egress, idle resources, cross-region.

Coming up: Plan B / Module 2 — How software is actually built.

The Cloud, Demystified

Reading time: ~10 min · Prereqs: Lessons 01–02 · *Snapshot 2026 — cloud products and prices evolve quarterly*; see [refresh-checklist.md](#).

Why this matters to you

"The cloud" is no longer a buzzword or a strategic question — it's the default. Whether you join a three-person startup or a Fortune 500 bank, your team is almost certainly running on AWS, Microsoft Azure, or Google Cloud, and quite likely on two of them at once. Cloud cost is now a top-three line item at many companies, which means conversations about *where things run* and *what they cost* spill out of the engineering room into product, finance, and the boardroom. Knowing the three layers of the cloud cake and the basics of how the bill is computed lets you sit at that table and follow along — even if you never personally launch a server.

1. What "the cloud" actually is

Strip away the marketing and "the cloud" is one idea: **renting computing infrastructure from someone else's data center, billed by usage**. Instead of buying servers, racking them, cooling them, and replacing them every four years, you swipe a credit card and get a virtual machine in two minutes. Instead of capacity-planning for the peak, you pay only for what you actually use.

This was a fringe idea in 2006, when Amazon launched S3 (cheap object storage) and EC2 (rentable virtual machines) and almost nobody believed a real company would put production workloads on someone else's hardware. By the mid-2010s the migration was in full swing. By 2026 the defaults have flipped: new companies start in the cloud; on-premise is the deliberate, defended choice. The shift is from **capex** (big up-front purchases of hardware) to **opex** (monthly operating expense), plus instant scale, plus no more "rack and stack" weekends.

2. The three layers: IaaS, PaaS, SaaS

Cloud offerings stack into three layers. The further up you go, the more the provider does for you — and the less control (and flexibility) you have.

- **IaaS — Infrastructure as a Service.** Raw virtual machines, raw block storage, raw networking. You manage the operating system, the runtime, the application, the patches. Examples: AWS `EC2`, Google `Compute Engine`, Azure `Virtual Machines`. Most "lift and shift" migrations land here — it's the layer that looks most like the old data centre.
- **PaaS — Platform as a Service.** The provider runs the database, the queue, the runtime, the container scheduler; you bring code and config. Examples: AWS `RDS` (managed Postgres/MySQL), `Lambda` (functions), `Fargate` (managed containers); GCP `Cloud Run`, `BigQuery`; Azure `Functions`, `Azure SQL`. Less ops work, more vendor lock-in.
- **SaaS — Software as a Service.** The provider runs the whole application; you use it via a browser or an API. Examples: Gmail, Notion, Figma, Slack, Stripe, Salesforce. Yes — Stripe and Slack are "cloud services" too. From the user's seat there's no server to think about at all.

Different teams live at different layers. Platform engineers spend their days in IaaS and PaaS. Frontend engineers might only ever touch PaaS and SaaS. Finance just sees the consolidated invoice.

3. The big three (and the rest)

Three providers dominate global market share. The numbers below are the 2025 figures; treat them as a snapshot, not a constant.

- **AWS — Amazon Web Services.** The biggest, the broadest catalog (200+ services), often the most expensive in any given category, and the one most teams hire for. Roughly **31% market share**. Started the modern cloud era.
- **Microsoft Azure.** Second-largest, deeply integrated with Microsoft 365, Active Directory, and the Windows server stack. Default choice in enterprises that are already Microsoft shops. Roughly **25%**.
- **Google Cloud (GCP).** Third by share but strong in data, analytics, AI/ML (Vertex AI), and Kubernetes ergonomics — GCP gave Kubernetes to the world. Roughly **12%**.

- **Others. Cloudflare** (edge-first compute and storage), **Oracle Cloud** (Oracle workloads and aggressive pricing), **IBM Cloud**, **Alibaba** (dominant in China). Niche but real.

Most large companies use **two** clouds — either deliberately, for redundancy and bargaining leverage, or accidentally, because an acquired company brought its own. Most startups use **one**.

4. The pricing surprise

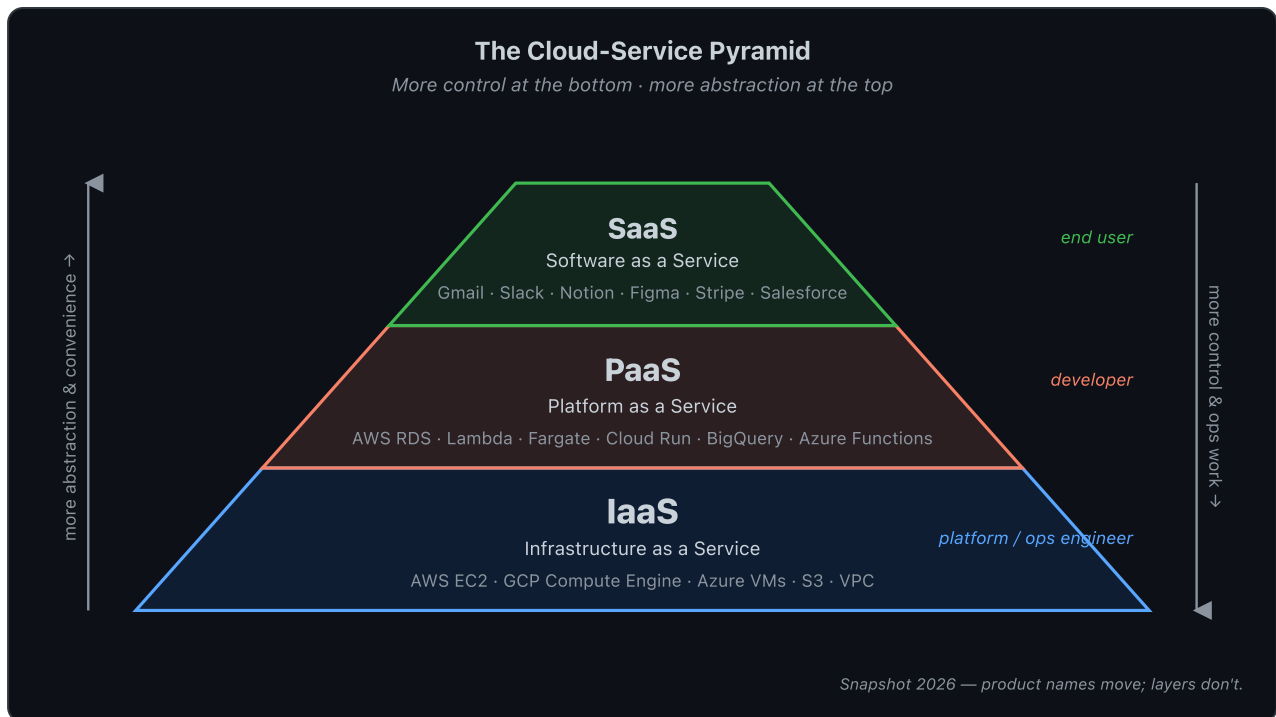
Cloud bills are measured in tiny units — seconds of compute (or milliseconds, for serverless functions), GB-hours of stored data, GB transferred *out*, per-API-call. The surface looks simple. Reality contains four classic surprises:

- **Egress.** Moving data *out* of the cloud (to your users, or to another cloud) is expensive. Moving data *in* is free. This single asymmetry shapes architecture decisions across the industry.
- **Idle resources.** A forgotten EC2 instance keeps billing 24/7 until somebody kills it. A single mid-sized VM left running by mistake can quietly burn \$200–\$500 in a month.
- **Cross-region traffic.** Even inside one provider, moving data between regions (say, US East to EU Frankfurt) costs money. Architectures that span regions for redundancy pay for that redundancy.
- **Premium support.** Production accounts typically pay a multi-thousand-dollar-per-month support tier on top of the resource bill, just to get a human on the phone when things break.

This is why **FinOps** (Cloud Financial Operations) is now a recognised profession, complete with certifications and a foundation. Lesson 10 returns to deployment and operational economics.

5. The picture: the cloud-service pyramid

The three layers fit together like a pyramid — broad and infrastructural at the bottom, narrow and user-facing at the top. Example products on each layer:



6. "Cloud-native" vs "lifted into the cloud"

Two very different patterns hide under the word "cloud":

- **Cloud-native.** The application was designed around managed services from day one. It assumes auto-scaling fleets, ephemeral storage, managed message queues, identity-as-a-service, and serverless functions. Restarts and node failures are normal, not emergencies. New companies almost always build this way.
- **Lift-and-shift.** An existing on-premise application was packaged up and moved into VMs inside the cloud, where it behaves exactly the way it did on the company's old hardware — same architecture, same scaling assumptions, same fragility. Most large enterprises are still mid-migration, with a mix of both patterns living side by side.

Lift-and-shift pays the cloud bill but rarely gets the cloud's elasticity benefits. Cloud-native gets both — at the cost of a redesign and an irreversibly deeper dependence on the provider's catalog.

What people argue about

Three live debates in the cloud world right now:

- **Single-cloud vs multi-cloud.** Multi-cloud is fashionable for redundancy and negotiating leverage with providers. Critics counter that the operational tax — two sets of identity systems, two consoles, two networking models, twice the training — outweighs the benefits for most teams. There is no industry consensus.
- **Cloud repatriation.** A small but high-profile cohort of companies (37signals being the loudest) are moving workloads *back* from the cloud onto self-owned hardware, arguing they now have predictable steady-state load and the cloud bill is no longer worth it. Most companies have not followed. The jury is still out on whether this is a trickle or a trend.
- **Hyperscaler lock-in.** The deeper you go into a provider's managed services (Lambda, DynamoDB, BigQuery, Cosmos DB), the harder it is to leave. Some teams deliberately stay close to "boring" portable primitives (just VMs, just Postgres, just S3-compatible storage). Others embrace the managed services for the velocity gains and accept the lock-in. The right answer depends on the company — and the bill.

Reading on

- **AWS Free Tier** — aws.amazon.com/free. Sign up, poke around the Console, click through the launch wizards without launching anything. No dollars required for the first 12 months of small resources, plus an always-free set.
- **The Cloud Resume Challenge** — cloudresumechallenge.dev. A popular hands-on entry path into cloud careers: build and host a simple resume site using real cloud primitives. Free, community-maintained.
- **FinOps Foundation** — finops.org. Community, certifications, and reading material if cloud cost is your angle into the field. Especially useful for finance and product folk who own a cloud P&L line.

LAST-REVIEWED: 2026-05-22

Homework — The Cloud, Demystified

Estimated time: ~45 min · No code required · No paid resources created

Three short tool-tour exercises. Goal: turn the abstract picture from the theory page (IaaS / PaaS / SaaS, regions, the billing model) into something you've actually *clicked through* in a real cloud console. You will sign up for AWS Free Tier, compare the three major consoles side-by-side, and read a simulated cloud bill. Take a screenshot at each step.

Money rule — read this first

These tasks involve real cloud-provider sign-up flows. **Do not click "Launch", "Create", or "Deploy" on any actual paid resource.** The Free Tier is \$0 only if you don't exceed its limits — the easiest way to stay at \$0 is to never create the resource in the first place. You will need a real credit card for AWS sign-up; that's normal and expected. Close every tab before walking away. If you accidentally launch something, immediately go back and click "Terminate" / "Delete".

Task 1 — AWS Free Tier sign-up & the EC2 launch wizard (don't launch)

First we'll create an AWS account and walk to the edge of launching a virtual machine — *without* actually launching it. The goal is to see what the wizard asks you for, not to run anything.

1. Open aws.amazon.com/free and sign up. You'll need an email address, a credit card, and a phone number for verification. The Free Tier itself is \$0 if you stay inside its limits.
2. Once you land in the AWS Management Console, use the search bar at the top to find **EC2** and open it.
3. Click **Instances** in the left sidebar, then click the orange **Launch instances** button. The launch wizard appears.

4. **STOP** — do not click "Launch instance" at the bottom. Instead, scroll through the wizard and look at every section it asks you about: *Name, Application and OS Images (AMI), Instance type, Key pair (login), Network settings / Security group, Configure storage*. Notice the "*Free tier eligible*" green label next to certain AMI and instance-type choices.
5. Take a screenshot of the launch wizard showing those dropdowns.
6. Close the tab **without** clicking Launch. (If you did click Launch by accident: go back to *Instances*, select the instance, *Instance state* → *Terminate*.)

✓ **Self-check:** you can name **three things** the wizard asked you to pick (e.g., AMI, instance type, key pair, security group, storage size); you can point to where the "*Free tier eligible*" filter / label is in the AMI & instance-type sections; you understand that **nothing is running and nothing is being billed** because you never clicked the final Launch button.

Task 2 — Compare the three major consoles

Same primitive (a virtual machine), three different consoles. We'll open AWS, GCP, and Azure side-by-side and notice how each provider frames the same idea.

1. **AWS:** sign in to the console from Task 1. In the search bar, open **EC2** → **Instances**. This is AWS's "list of virtual machines" page.
2. **GCP:** open console.cloud.google.com. Any free Google account works — you don't need to enable billing or accept the \$300 free trial. Use the left/top menu to find **Compute Engine** → **VM instances**. (You may have to "create a project" first — that's free and doesn't create any chargeable resources.)
3. **Azure:** open portal.azure.com. A Microsoft account works. Use the top search bar to find **Virtual machines**. This is Azure's equivalent page.
4. Take a screenshot of each provider's compute landing page.
5. Write **one sentence per provider** noting one thing that looks *different* about how that console is organised. Examples: "*Azure groups everything into 'Resource Groups' in the left nav*", "*GCP forces you to pick a project from a header dropdown before anything shows up*", "*AWS's region selector is in the top-right and the whole console rebuilds when you change it*".

✓ Self-check: you can map the three names — **EC2 (AWS) = Compute Engine (GCP) = Virtual Machines (Azure)** — from memory; you have a one-sentence note for each provider about one organisational quirk; you understand that the underlying primitive ("a Linux VM you SSH into") is the same on all three — the marketing names and the menu structure are what differ.

Task 3 — Read a real cloud bill (using the calculator, \$0 real cost)

Cloud pricing is the part that surprises people. We'll model the same tiny workload in AWS's official price calculator and see what makes the bill move.

1. Open calculator.aws. Click **Create estimate**, search for **EC2**, and click *Configure*.
2. Configure these inputs (don't worry about the rest — defaults are fine):
 - **Region:** `US East (N. Virginia) us-east-1`
 - **Operating system:** Linux
 - **Instance type:** `t3.small`, running **24/7** (730 hours / month)
 - **Storage (EBS):** 50 GB, general purpose (`gp3`)
 - **Data transfer out to the internet:** 100 GB / month

Note the **estimated monthly cost**. Write it down.

3. Now change the region from `us-east-1` to `EU (Frankfurt) eu-central-1`. Note the new monthly cost. Write it down.
4. Now change **Data transfer out** from 100 GB to **1 TB** (1024 GB). Note the new monthly cost. Write it down.
5. Take a screenshot of the final estimate breakdown (showing compute, storage, and data-transfer line items separately).

✓ Self-check: you can answer (a) approximately how much an **idle** `t3.small` costs per month (compute + 50 GB storage + 100 GB egress, all in `us-east-1`); (b) **yes/no**: does region change the price for the same instance type? (c) when egress went from 100 GB to 1 TB, which line item *dominated* the bill — compute, storage, or data transfer? You now understand why *"the cloud is*

expensive" is usually a statement about **egress and idle resources**, not about compute per se.

What "done" looks like

You finish this homework when you have, in your notes:

- A screenshot of the EC2 launch wizard, plus three things it asked you to pick — without having launched anything (Task 1).
- Three screenshots (one per provider) of the compute landing page in AWS, GCP, and Azure, plus a one-sentence "what's different" note for each (Task 2).
- Three monthly cost numbers from the calculator (us-east-1 / 100 GB, eu-central-1 / 100 GB, eu-central-1 / 1 TB) and a one-line answer to "which line item dominated when egress went from 100 GB to 1 TB?" (Task 3).

That's the whole cloud picture from the theory page, observed in real consoles you can log into right now: *the primitive (a VM) is the same everywhere; the consoles look different; the bill is dominated by whatever you forgot to turn off — usually egress or idle resources*. Next time someone says "*just put it in the cloud*", you'll know what questions to ask first.

Lesson 04: Modern Web Stack

Lesson 04 — Modern Web Stack

Frontend / Backend / API — what each is, why apps split that way.

IT Landscape

Module 2

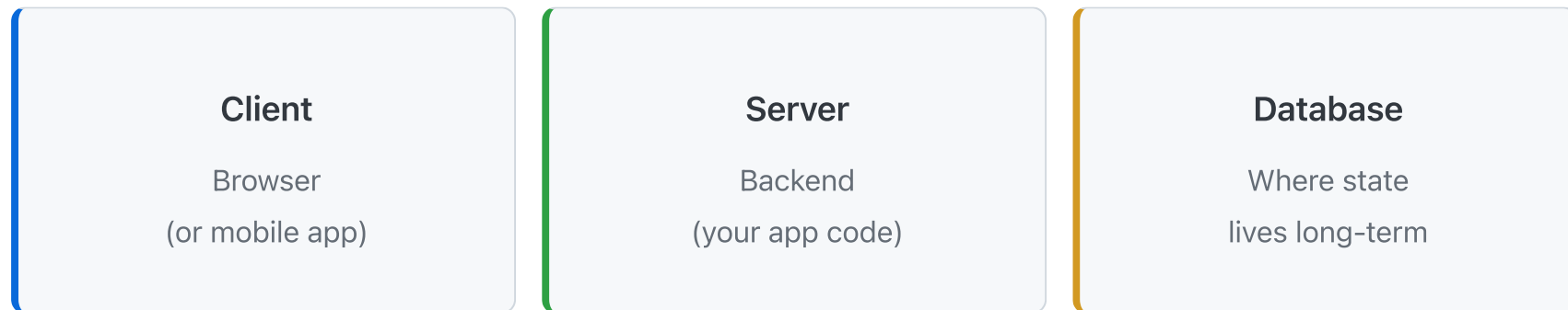
What is a "web app"?

*A website that **does** something — not just shows you something.*

Examples: **Gmail, Google Docs, Notion, Spotify Web Player, Slack.**

Contrast: a restaurant's menu page. That's a *brochure* — you read it and leave. A web app remembers you, accepts your input, talks back.

The 3-tier model in one diagram



Client — *HTTP request* → Server — *SQL query* → Database

Almost every app on the internet is some flavor of this.

Tier 1 — Frontend (the browser)

- **HTML** — structures the page.
- **CSS** — styles it (colors, layout, fonts).
- **JavaScript** — makes it interactive.

The frontend = **everything the user actually sees and clicks**. Lives in the browser. Sends HTTP requests when it needs data.

Note: "frontend" is sometimes called "client-side" — same thing.

Tier 2 — Backend (the server)

- An **HTTP server** (Nginx, Node, Django, Rails) plus your **app code**.
- Receives a request → talks to a database → returns JSON or HTML.
- Runs on a Linux box in a data center somewhere — or on serverless infra (Lesson 06).

The backend is the only thing trusted to read or write the database. The client never talks to the DB directly.

The handshake between them — APIs

An API is a contract: "ask me this way, I'll answer this way."

Most modern APIs speak **REST** (over HTTP, JSON-shaped) or **GraphQL** (single endpoint, you specify the shape).

Illustrative one-liner — you do *not* need to run this:

```
fetch('/api/users/42') // server returns { "id": 42, "name": "Ada" }
```


Real example — a modern site under the hood

What happens when you open `notion.so`:

1. Browser downloads ~100 KB of JavaScript (a React app).
2. That JS calls `/api/v3/getSpaces`.
3. Backend hits a **Postgres** database.
4. Backend returns JSON.
5. React renders the sidebar — the page lights up.

Five steps. Three tiers. **Same shape as Gmail, Slack, your bank.**

Frontend frameworks you'll hear named (as of 2026)

Name	From	What it's for
React	Meta	Biggest mindshare
Vue	Independent	Lightweight alternative
Svelte	Independent	Compiles away — small bundles
Angular	Google	Enterprise, opinionated
Next.js	Vercel	React + a backend bundled
Astro / SolidJS	Independent	Newer, niche

Most teams pick one and stick with it for years.

Backend frameworks you'll hear named (as of 2026)

Language	Common frameworks
Python	Django, FastAPI
Node.js	Express, NestJS
Ruby	Rails
Java	Spring Boot
C#	ASP.NET
PHP	Laravel
Go	<code>net/http</code> + chi / gin
Elixir	Phoenix

Many roads up the same mountain.

Static sites vs dynamic apps

Static

Built once, served to all.

- Blogs
- Documentation
- Marketing pages

Tools: Hugo, Jekyll, Astro

Dynamic

Rendered per user / per request.

- Dashboards
- Social feeds
- E-commerce

Tools: anything with a backend

Mobile fits in too

- **Native** — Swift for iOS, Kotlin for Android. Fastest, separate codebases.
- **Hybrid / cross-platform** — React Native (Meta), Flutter (Google). One codebase, both stores.
- **Mobile web** — just a website on your phone, possibly installable as a **PWA** (Progressive Web App).

All three usually talk to the **same backend + same API** as the web app.

What you'll see on a job description

- "Looking for a **React** developer with **REST API** experience."
- "Backend engineer — **Python/Django, Postgres, AWS**."
- "Full-stack — **TypeScript** everywhere."

You now have the decoder ring. Frontend, backend, API, database — you can place every one of these words.

Recap

- **3 tiers:** client (browser) ↔ server (backend) ↔ database.
- **APIs** glue them together — REST and GraphQL are the two big shapes.
- **Frontend and backend frameworks** are tools, not religions. Many roads up the same mountain.

Coming up: Lesson 05 — what writing code actually looks like in 2026.

Modern Web Stack

Reading time: ~10 min · Prereqs: Lessons 01–03 · *Snapshot 2026 — framework names move; the three-tier shape doesn't.*

Why this matters to you

Almost every product you touch online — Gmail, your bank's website, your company's HR portal, the shop where you bought your last pair of sneakers — shares the same basic shape: a browser talking to a server talking to a database. Once you can see that shape, you can ask the right questions: *what's slow?*, *where does the data live?*, *what changed when?*, *who owns this piece if it breaks?* This lesson hands you the shape and the vocabulary. After it, the rest of Module 2 — programming (Lesson 05), architecture (Lesson 06), and databases (Lesson 07) — all slot neatly in. You don't need to write code to use this vocabulary; you need it to read a job posting, sit in a planning meeting, or scope a vendor.

1. What "web app" means in 2026

A **web app** is a website that *does* something rather than just *shows* something. Webmail (Gmail, Proton Mail). Collaborative documents (Google Docs, Notion). Streaming (Spotify, Netflix). Online banking. Booking tools. Your company's Jira. The line between "web app" and "desktop app" has blurred — many apps you double-click (Slack, VS Code, Discord, Notion desktop) are literally websites wrapped in a desktop window using a framework called **Electron**. Mobile is similar: a huge share of the apps on your phone are either thin shells around a web experience or native apps talking to the same backend that the web version uses.

One mental model fits almost all of them: **three tiers**. Once you can name the tiers, you can read any product like a map.

2. The three-tier model

Modern web apps split into three layers, each with one job.

- **Tier 1 — the client.** Whatever's on the user's device: a browser tab, a mobile app, sometimes a desktop window. Its job is to draw the screen and capture clicks and typing.
- **Tier 2 — the server.** Code running in a data center, listening for requests. Its job is to apply the rules (who's allowed to do what?) and to mediate every conversation with the database.
- **Tier 3 — the database.** Where state lives long-term: your account, your messages, your photos, your order history. Its job is to remember.

The client doesn't talk to the database directly — it talks to the server, and the server is the only thing trusted to read or write the database. This separation isn't bureaucracy; it's how you prevent a malicious or buggy client from running `DELETE * FROM users;`. Walk through one click: you press *Send* in Gmail → your browser sends an HTTP request to Google's mail server → the server checks your auth, writes a row, fans the message out to recipients' inboxes (more database writes) → returns success → your browser updates the screen. That tiny dance happens millions of times a second across the internet.

3. Frontend — what runs in your browser

The **frontend** is everything that runs on the client. At the bottom it's just three languages, all of which your browser speaks natively:

- **HTML** — the structure (headings, paragraphs, buttons).
- **CSS** — the style (colors, spacing, fonts, layout).
- **JavaScript** — the behavior (what happens when you click, type, or scroll).

For anything beyond a brochure site, a **framework** sits on top of those three languages to keep big UIs sane. As of 2026 the most common are **React** (made at Meta, biggest mindshare and largest job market), **Vue**, **Svelte**, and **Angular** (Google, enterprise-leaning). **Next.js** is "React plus a backend bundled together" and is currently the default new-project choice for many teams. A modern frontend ships somewhere between 100 KB and 1 MB of JavaScript to your browser the first time you visit a site; the browser then runs that JavaScript, the JavaScript draws the UI, and after that it chats with the server in the background as you click around.

4. Backend — what runs in the data center

A **backend** is an **HTTP server** (a program that listens for web requests) plus your **app code** (the rules of *your* product). Common language-plus-framework combinations as of 2026:

- **Python + Django** or **FastAPI**
- **Node.js + Express** or **NestJS**
- **Ruby + Rails**
- **Java + Spring Boot**
- **C# + ASP.NET**
- **Go** with its standard library (frameworks are deliberately minimal)
- **PHP + Laravel** (still huge — WordPress, Wikipedia, much of Facebook's origin)
- **Elixir + Phoenix** (smaller, beloved for real-time apps)

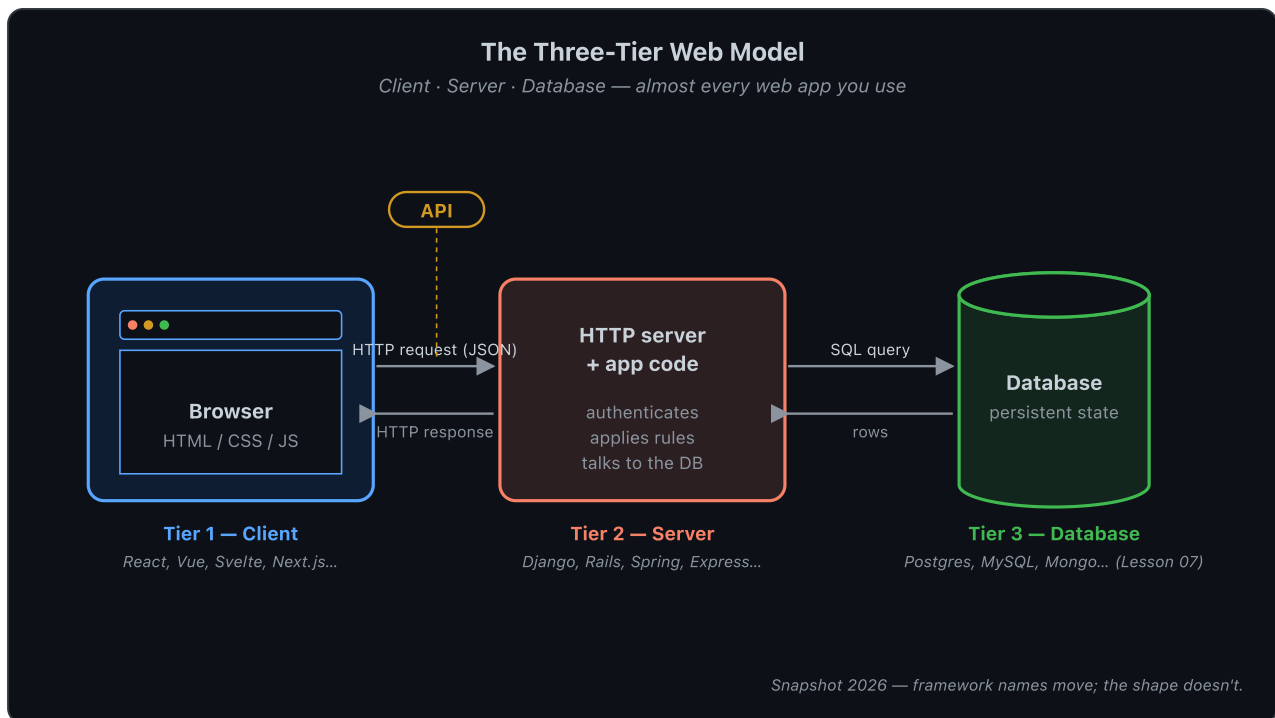
None of these is "the right one" — each language community has matured tooling and shipped real products at huge scale. Companies pick based on what their team already knows, what their hiring pool looks like, and what their product needs. The backend's job is the same regardless of language: receive a request, decide if the user is allowed to do this, talk to the database, return a response.

5. APIs — the contract between tiers

An **API** (Application Programming Interface) is a contract: *"ask me this way, I'll answer this way."* Most modern web APIs are **REST** — HTTP verbs (`GET`, `POST`, `PUT`, `DELETE`) plus URL paths plus JSON bodies. For example, `GET /api/users/42` returns `{ "id": 42, "name": "Ada" }`. A growing share use **GraphQL**, which exposes one endpoint and lets the client say what shape of data it wants. Newer **gRPC** is common for service-to-service traffic inside a company, where speed matters more than human readability. The big point: APIs aren't a separate technology — they're how the frontend and backend agree to talk, and how third-party apps (your phone's Slack app, Zapier, the AWS CLI) talk to a service from the outside.

6. The picture

Here is the whole three-tier model in one diagram. Notice the API label on the arrow between the client and the server — that's the contract you just read about.



7. Mobile, briefly

Mobile apps come in three flavors. **Native** apps are built per platform — **Swift** for iOS, **Kotlin** for Android — and feel most at home on the device, at the cost of maintaining two separate codebases. **Hybrid / cross-platform** uses one codebase to target both: **React Native** (Meta) and **Flutter** (Google) are the two large frameworks as of 2026. **Mobile web** is just a website your phone visits — sometimes installable to the home screen as a **PWA** (Progressive Web App), which can run offline and send notifications. All three flavors usually talk to the same backend and the same API as the web app, which is precisely the payoff of clean tiers in the first place.

What people argue about

Three live debates in modern web development right now:

- **REST vs GraphQL.** GraphQL solves real problems (overfetching, underfetching, one round trip for a complex screen) but adds server-side complexity and a learning curve; REST critics counter that GraphQL is "just

rebuilding SQL over HTTP, badly". Both still ship in production at scale and neither is winning outright.

- **Server-side rendering vs single-page app.** For a decade SPAs (single-page apps in React, Vue, Angular) replaced server-rendered HTML; now the pendulum is swinging back toward server rendering (**Next.js**, **Remix**, **Astro**, **Phoenix LiveView**) for faster first paint and better SEO. Expect the answer to keep shifting.
- **TypeScript everywhere.** The JavaScript world has largely adopted **TypeScript** (JavaScript with type annotations); Python and Ruby are slowly adding optional typing. Some argue the type-safety tax is too high for small projects and the tooling cost real; most teams who tried it didn't go back.

Reading on

- **MDN Web Docs** — developer.mozilla.org. The canonical free reference for HTML, CSS, and JavaScript, run by Mozilla. If a browser feature exists, MDN documents it.
- **web.dev** — web.dev. Google's home for modern web best-practices: performance, accessibility, PWAs, Core Web Vitals. Short, opinionated, current.
- **Lesson 06 of this course** — returns to the backend at a higher zoom level (monolith vs microservices vs serverless vs event-driven). Read this lesson first, then 06 will make sense.

LAST-REVIEWED: 2026-05-25

Homework — Modern Web Stack

Estimated time: ~45 min · No code required · Just your browser and its dev tools

Three short tool-tour exercises. The goal: take the abstract picture from the theory page (browser ↔ API ↔ database, static vs. dynamic, frontend / backend / database as three separable jobs) and see *it yourself* in real sites you already use. Open your browser's developer tools, click around, and notice the three tiers in the wild. Take a screenshot at each step.

Before you start

All three tasks work in any modern browser (Chrome, Edge, Firefox, Safari). Open developer tools with **F12** (Windows/Linux) or **Cmd + Option + I** (macOS), or right-click any element and choose *Inspect*. You're only looking — you won't change anything on these sites.

Task 1 — See the 3-tier model in action

A real production app is constantly talking to its backend. We'll watch that traffic happen live and notice that the calls look nothing like the URL in the address bar.

1. Open notion.so (sign in, or use any public Notion page somebody has shared with you).
2. Right-click anywhere on the page → **Inspect**. In the dev-tools panel that opens, click the **Network** tab.
3. Reload the page (**Ctrl/Cmd + R**). Watch requests stream in. Wait a few seconds for the traffic to settle.
4. Now click anywhere on the page — open a sub-page, expand a toggle, anything interactive. *New* rows will appear in the Network tab as the frontend

asks the backend for things.

- Find a request whose path starts with `/api/`. Click it. In the side panel, open the **Headers** tab and note the *Request URL*. Then open the **Response** tab and note that the body is **JSON**, not HTML.
- Take a screenshot of the Network tab with the `/api/` row selected and the JSON response visible.

✓ **Self-check:** you can answer (a) is the URL of an `/api/` request the same as the URL in your browser's address bar? (*no — the page URL is something like `notion.so/...`, the API URL starts with `/api/` and goes to a different backend path*); (b) what **format** is the response in? (**JSON**); (c) approximately **how many** `/api/` calls did one click generate? — you now have a number, even if it's "lots". You've just watched the frontend tier (the React app in your browser) talk to the backend tier (Notion's servers) over HTTP, exchanging JSON instead of HTML.

Task 2 — Tell a static site from a dynamic one

Not every site does what Notion does. Some ship a fully-rendered HTML page and call it done; others ship a near-empty shell and let JavaScript fill it in. View Source is the cheapest way to tell them apart.

- Open three sites in three tabs:
 - news.ycombinator.com
 - github.com (your own profile if you have an account, otherwise github.com/torvalds)
 - astro.build
- In each tab, right-click on the page background → **View Page Source** (or **Ctrl/Cmd + U**). This shows you the raw HTML the server sent *before* any JavaScript ran.
- For each site, scan the source and decide: is the visible page content (story titles, profile name, marketing copy) **already in the HTML**, or is the `<body>`

mostly empty with one big `<div id="root"></div>` and a stack of `<script>` tags?

4. Write a one-line verdict per site: "*server-rendered / static*" vs. "*client-rendered SPA*".

✓ **Self-check:** Hacker News and the Astro marketing site are largely **already rendered** — the story titles and marketing copy are right there in the HTML the server sent. Many SPAs (and parts of GitHub) ship a near-empty `<body>` and let JavaScript populate it after load. You can now classify any site you visit by this same trick: *view source → is the content there, or not?* If "not", you're looking at a single-page app that's about to make a bunch of `/api/` calls just like Notion did in Task 1.

Task 3 — Read a job posting through this lens

The 3-tier picture isn't just an academic model — it's literally how teams are organised and how job postings are written. Once you see it, you can't un-see it.

1. Open [linkedin.com/jobs](https://www.linkedin.com/jobs) or [indeed.com](https://www.indeed.com). Search for "**Full-stack Engineer**".
2. Open three different listings from three different companies.
3. In each posting, scroll to the *Requirements*, *Tech stack*, or *You'll work with* section.
4. For each posting, sort the named technologies into three columns:
 - **Frontend:** React, Vue, Svelte, Angular, HTML/CSS, TypeScript on the browser side, Tailwind, Next.js, ...
 - **Backend:** Python, Django, FastAPI, Node.js, Express, Go, Java, Spring, Ruby on Rails, .NET, ...
 - **Database:** PostgreSQL, MySQL, MongoDB, Redis, DynamoDB, ...
5. Note which tier each company seems **most heavy on** — some postings are 80% frontend, others 80% backend, "full-stack" sometimes leans one way more than the other.

6. Park anything that doesn't fit those three columns (e.g., *Docker*, *Kubernetes*, *AWS*, *Terraform*, *GitHub Actions*). Lessons 06 (architecture) and the cloud lesson you've already done cover those.

✓ **Self-check:** for each of your three postings you have a three-column breakdown; you can name which tier each posting leans on most heavily; the words "full-stack" no longer feel mysterious — you can read a job ad and immediately see *which slice of the 3-tier stack the role actually lives in*.

What "done" looks like

You finish this homework when you have, in your notes:

- A screenshot of the Notion **Network** tab with at least one `/api/` request open and its JSON response visible — plus your own answers to the three Task 1 questions (URL difference, response format, approximate call count).
- A one-line verdict per site (Hacker News / GitHub / Astro) on whether it's server-rendered or a client-rendered SPA — based on what you saw in *View Source*.
- Three Frontend / Backend / Database breakdowns — one per job posting — plus a one-line note on which tier each company leans on hardest.

You've now seen the picture from the theory page in three separate places: live in the dev tools (Task 1), in the source of static vs. dynamic sites (Task 2), and in how the industry hires for it (Task 3). Next lesson we move down one level — what the act of *writing* code actually looks like in 2026.

Lesson 05: Programming Today

Lesson 05 — Programming Today

What writing code actually looks like in 2026 — languages, IDEs, AI assistants, git, GitHub.

IT Landscape

Module 2

What is a “programming language”?

A formal language a human writes so a computer can do something predictable with it.

- **Hundreds exist** — new ones invented every year.
 - **Most paying jobs** use one of about ten.
 - Languages are **tools, not religions**.
-

The 2026 language landscape

Language	Where you'll find it
Python	Data, AI/ML, scripting, backends
JavaScript / TypeScript	Everywhere on the web — browsers + Node.js backends
Java	Enterprise systems + Android
C#	Windows, Unity games, Microsoft-shop enterprise
Go	Cloud infrastructure, CLIs (Docker, Kubernetes)
Rust	Systems, performance-critical, replacing C/C++ in places
Swift	iOS apps
Kotlin	Android + JVM backends

Plus **C / C++** — still huge in operating systems, embedded, games.

Note: as of 2026 — see TIOBE / GitHub Octoverse for current rankings.

An IDE in 2026

An **IDE** (integrated development environment) is a text editor with a lot of brains. It:

- Edits code with **syntax coloring** and autocomplete
- **Runs** the code and **debugs** it
- Shows **git changes** inline
- Runs **tests**
- Talks to the **AI assistant** in a side panel

Names you'll hear (as of 2026):

- **VS Code** — Microsoft, free, dominant.
 - **JetBrains IDEs** — PyCharm, IntelliJ, GoLand. Paid, very polished.
 - **Cursor / Windsurf** — VS Code forks with AI built in.
 - **Neovim** — modal, terminal-based, plugin culture.
-

AI coding assistants

The biggest shift in how code gets written since the internet.

- **GitHub Copilot** — inline suggestions, oldest at scale.
- **Cursor** — chat + edits + agent inside the editor.
- **Claude Code** — terminal-first, agentic.
- **Windsurf, Cody, Continue.dev** — same space, different flavors.

Most engineers in 2026 use at least one daily. AI writes a meaningful fraction of new code; the engineer's job has shifted toward **direction, review, and ownership**.

Note: as of 2026 — vendors and capabilities shift quarterly.

Git in one slide

Git remembers every change you make to your code.

- Every project has a `.git` folder.
- `git add` — choose what to save.
- `git commit` — snapshot it with a message.
- `git push` — send it to a shared server.
- **Branching** = work on a side line. **Merging** = bring it back.

That's 80% of daily git.

GitHub in one slide

GitHub is the most popular **place** to keep your git repositories. It also runs:

- **Issues** — bug and feature tracking.
- **Pull requests** — proposed changes that teammates review.
- **Actions** — automated checks that run on every change.
- The home of most **open source** software.

Alternatives: **GitLab**, **Bitbucket**, **Codeberg**, **Gitea**.

A real Python snippet

```
def greet(name):  
    return f"Hello, {name}!"  
  
people = ["Ada", "Linus", "Grace"]  
greetings = [greet(p) for p in people]  
  
if __name__ == "__main__":  
    for line in greetings:  
        print(line)
```

You don't need to run this — just notice the indentation and the readable shape.

A real TypeScript snippet

```
interface User {  
  id: number;  
  name: string;  
}  
  
function greet(user: User): string {  
  return `Hello, ${user.name}!`;  
}  
  
const users: User[] = [  
  { id: 1, name: "Ada" },  
  { id: 2, name: "Linus" },  
];  
  
users.forEach((u) => console.log(greet(u))):
```

Same idea, different language. Notice the explicit types — that's the "TS" in TypeScript.

The open source ecosystem

Most software is built on free, open code.

- **npm** (JS / TS) — ~3M packages.
- **PyPI** (Python) — ~500k packages.
- **Maven** (Java), **crates.io** (Rust), **Go modules**.

You install dependencies with one command:

- `npm install`
- `pip install`
- `go get`

Real apps have **hundreds** of indirect dependencies. Security and licensing matter.

Reading other people's code

Most of a working programmer's day.

Engineers read more code than they write.

Skills:

- Skim a repo's **README**.
 - Follow **imports**.
 - Navigate by **symbol search** (Cmd-T in VS Code).
 - Use AI to **summarize** unfamiliar files.
-

Career paths from “writes code”

Build the product

- Backend engineer
- Frontend engineer
- Full-stack engineer
- Mobile engineer

Adjacent roles

- Data engineer
- DevOps / SRE
- ML engineer
- Engineering manager

Lesson 14 returns to this map.

Recap

- The **language pool** is small once you know which jobs use which.
- The **editor** is just a text editor with a brain bolted on.
- **AI assistants** are the biggest shift in how code gets written since the internet.

Coming up: Lesson 06 — how software is shaped at the bigger-picture level.

Programming Today

Reading time: ~10 min · Prereqs: Lessons 01–04 · *Snapshot 2026 — AI assistants and IDE landscape change quickly; see [refresh-checklist.md](#)*.

Why this matters to you

"Programmers" don't sit in dark rooms typing arcane symbols. In 2026, writing software looks more like an editor-meets-chat-meets-version-control-meets-AI-assistant — a craft that has changed more in the last three years than in the previous twenty. Even if you'll never write a line yourself, knowing what the daily activity actually looks like makes you a much better partner to people who do: product managers, designers, executives, technical recruiters, and anyone who has to estimate, scope, or sell software. This lesson is a quick tour of the toolkit, the languages, and the workflow that defines what "writing code" means today.

1. What a programming language really is

A programming language is a formal grammar — strict enough that a tool called a **compiler** or an **interpreter** can read it and tell the computer exactly what to do. There are hundreds; most paid jobs use one of about ten. Languages aren't religions — they're tools. Most working engineers move fluently between two and four daily, usually in some mix like "Python at work plus a sprinkle of SQL, a few bash scripts, and a little JavaScript when configuring the build."

Compiled languages (C, C++, Rust, Go) are translated to machine code ahead of time and run very fast. Interpreted languages (Python, Ruby, JavaScript) are read and executed line-by-line at runtime, which trades some speed for flexibility. The line has blurred — modern JavaScript engines compile on the fly, and Python has just-in-time options — but the mental model still helps.

2. The 2026 language landscape

A rough roll-call of what teams actually hire for, *as of 2026*:

- **Python** — everywhere in data, AI, scripting, and a huge slice of backends. If you only ever learn one language, this is the safest pick.
- **JavaScript** and its typed sibling **TypeScript** — the only languages that run natively in browsers, and now powering most backends too via Node.js. TypeScript has displaced bare JavaScript on most new projects.
- **Java** — the workhorse of enterprise systems and Android applications. Still enormous in banking, telecom, and large e-commerce.
- **C#** — the Microsoft / Unity-games equivalent; dominant in Windows-heavy enterprises and a huge slice of game dev.
- **Go** — dominates cloud infrastructure (Docker, Kubernetes, much of the modern CLI ecosystem). The default for new infra tooling.
- **Rust** — replacing C/C++ in performance-critical and memory-safety-critical spots; the darling of systems engineers, embedded teams, and the crypto world.
- **Swift** and **Kotlin** — own iOS and Android development respectively.
- **C** and **C++** — still huge in operating systems, browsers, game engines, and embedded devices.
- **SQL** — its own thing; you'll meet it any time a database comes up (Lesson 07).

Trends *as of 2026*: Rust adoption is growing in places that used to be C++; Go is the default for new infra tooling; TypeScript has displaced bare JavaScript on most new projects; Python's grip on AI and data hasn't loosened. Refresh annually — [refresh-checklist.md](#) calls this out explicitly.

3. An IDE in 2026

An **IDE** ("integrated development environment") is a text editor with a lot of brains bolted on: syntax coloring, autocomplete, jump-to-definition, inline error squiggles, a debugger, git integration, a terminal, and — new in this generation — an AI assistant in the side panel. Think of it as the difference between Notepad and a full word processor: both let you type, but only one catches your typos in real time, suggests the next sentence, and remembers every revision.

The dominant choices *as of 2026* are **VS Code** (Microsoft, free, hugely extensible — sometimes 70%+ market share in developer surveys) and the **JetBrains** family

(PyCharm, IntelliJ, GoLand, WebStorm — paid, very polished, popular at companies with budget for tooling). A growing third camp is **VS Code-with-AI-baked-in**: forks like **Cursor** and **Windsurf** have native AI editing built into the editor rather than added as a plugin. The terminal-and-Vim crowd is still alive and well; **Neovim** with the right plugins is a full IDE for a meaningful slice of engineers, and it's the default in environments where you SSH into a remote machine and edit there.

4. AI coding assistants — the big shift

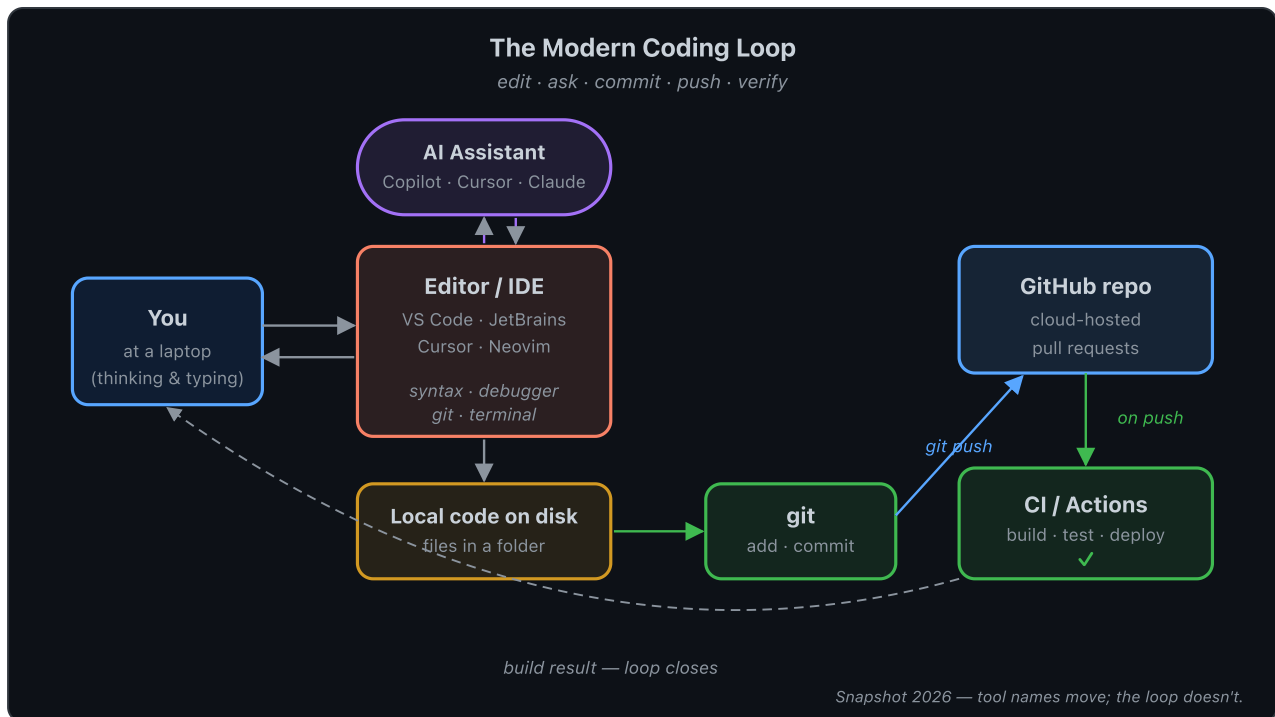
Three years ago this section didn't exist. *As of 2026*, most working engineers use an AI coding assistant daily. The notable players right now:

- **GitHub Copilot** — autocomplete-style; the first to land at scale.
- **Cursor** — popularized "chat plus agentic edits in the same editor."
- **Claude Code** — brought the same energy to the terminal.
- **Windsurf, Cody, Continue.dev** — all play in the same space with slightly different bets.

What they actually do: suggest the next 1–10 lines as you type, answer "what does this file do?" in a side panel, refactor across a whole repo on request, and — increasingly — *do tasks* agent-style, with the engineer reviewing the result. The engineer's job has shifted: less typing, more reading, more reviewing, more deciding what to build, more owning the output. *(This whole section is on [refresh-checklist.md](#) — the brand names will move.)*

5. The picture: the modern coding loop

Here's how the pieces fit together in a typical day. A person edits files in an IDE with an AI assistant whispering suggestions, snapshots the work into git, pushes it to GitHub, and watches automated pipelines run checks on every change.



6. Git in two paragraphs

Git is a version-control system: it remembers every change you make to a project. When you've changed a few files, you `git add` the ones you want to save, then `git commit` to snapshot them with a short message ("fix login bug"). You can branch off the main line, work on a side line in isolation, and merge back when ready. Each commit has a fingerprint; you can go back to any previous state in seconds. Think of it as "Track Changes" for code, except it scales to thousands of contributors on the same project without anyone overwriting each other.

GitHub (and its alternatives **GitLab**, **Bitbucket**, **Codeberg**) is the most popular place to host your git repositories online so teammates can pull a copy, propose changes (**pull requests** / **merge requests**), discuss them inline, and run automated checks (**GitHub Actions**) on every change. GitHub is also where most open-source software lives — including a meaningful share of the dependencies in the apps you use every day. An engineer's "public profile" today is often their GitHub page, not their resume.

What people argue about

- **Will AI replace engineers?** Three years in, AI assistants have made engineers significantly faster, especially at junior and grunt-work tasks. Headcount hasn't dropped meaningfully across the industry; ambition per engineer has

gone up. Predictions for the next 3–5 years range from "no further change" to "10x productivity again." Pick a camp at your peril.

- **TypeScript vs plain JavaScript.** TypeScript has won most new projects, but maintainers of widely-used libraries (e.g., Svelte famously dropped TS internally) sometimes argue the type-safety tax outweighs the benefit at certain scales. The debate is alive in every JS-shop hiring loop.
- **VS Code monoculture.** Microsoft's editor is now so dominant that some engineers worry about the platform risk of one vendor owning the editor, the AI assistant (GitHub Copilot), the cloud (Azure), and the largest code-hosting site (GitHub) all at once.

Reading on

- **Roadmap.sh** — roadmap.sh. Community-curated learning roadmaps for every common role (frontend, backend, DevOps, AI engineer). A solid first map of the territory.
- **GitHub's "Hello World" tutorial** — docs.github.com/get-started/quickstart. About 15 minutes, no prior knowledge needed; you'll have a real repo with a real commit when you're done.
- **learnxinyminutes.com** — every major language on one scrollable page, when you just want to see what the syntax looks like before committing to learning it.

LAST-REVIEWED: 2026-05-25

Homework — Programming Today

Estimated time: ~60 min · No coding required · A free editor, a free GitHub account, and an AI assistant of your choice

Three short tool-tour exercises. The goal: take the abstract picture from the theory page (an editor + a repo + an AI assistant, glued together by git) and see *it yourself* in the actual tools professionals use every day in 2026. You won't write a single line of code — you'll just install, click around, and notice the parts. Take a screenshot at each step.

Before you start

You only need a laptop and a browser. Tasks 1 and 2 use the free tier of public tools (VS Code is free; GitHub is free for public browsing). Task 3 works with any AI chat assistant you can reach — *Claude*, *ChatGPT*, *Gemini*, the *Cursor* playground, or whichever one your organisation already allows. You are not writing or running any code.

Task 1 — Install VS Code and poke around (no coding required)

VS Code is, as of 2026, the editor most professional engineers actually open every morning. Today you'll just install it and learn where the five core panels are — no project, no code, just the geography.

1. Download VS Code from code.visualstudio.com. Pick the build for your operating system. Install it the normal way (drag to *Applications* on macOS, run the installer on Windows, use your package manager on Linux).
2. Open VS Code. You'll see an empty welcome tab — that's fine.

3. Look at the thin vertical strip on the very left edge. It has a stack of icons. Click each one in turn and note what panel opens:
 - The **Explorer** (folder icon) — this is where your files would show up if you opened a project folder.
 - The **Search** panel (magnifying glass) — project-wide find & replace.
 - The **Source Control** panel (branching icon) — empty until there's a git repo open, but this is where every commit you ever make starts.
 - The **Run and Debug** panel (play-with-bug icon) — for stepping through code line by line.
 - The **Extensions** panel (four-squares icon) — the marketplace; this is how VS Code becomes a Python editor, a Rust editor, a SQL editor, etc.
4. Click the gear icon at the very bottom-left and open **Settings**. Scroll for ten seconds and notice there are *thousands* of options. You don't need to change anything.
5. Open the **Command Palette**: **Cmd + Shift + P** (macOS) or **Ctrl + Shift + P** (Windows/Linux). Type a few letters of a word like *theme* or *format* and notice it offers you a filtered list of every command in the editor. This is the single most-used keyboard shortcut in VS Code.
6. Take a screenshot of the editor with the Command Palette open.

✓ **Self-check: you can name each of the 5 sidebar panels (Explorer, Search, Source Control, Run and Debug, Extensions); you can open the Command Palette and find one command by typing a keyword. You now know the geography of the tool every engineer in your team will assume you know.**

Task 2 — Read open-source code without writing any

Most software in the world lives in a public git repository on GitHub, and you can read all of it without an account. Today you'll browse one of the most-watched repositories on the platform and notice that it is, after all, just files of text.

1. Open github.com. In the top search bar, type `react`. The top result should be **facebook/react** — click it.
2. You're now on the repo's home page. The big file tree on the left is the project. Browse it:
 - Open `README.md` — this is what GitHub renders by default; it's the project's front door.
 - Open the `packages/` folder.
 - Click into `packages/react/`, then `src/`, then a file ending in `.js` (e.g., `React.js`).

Notice: it's just a text file. Long, dense, full of comments — but text.
3. Go back to the repo home page. Look near the top right for the **Contributors** count, and on the file tree for the timestamp on the most recent commit (the "last updated" column).
4. Click the **Insights** tab at the top, then **Pulse** in the left sidebar. Pulse summarises activity over the past week: pull requests opened/merged, issues opened/closed.
5. Take a screenshot of the Pulse page.

✓ **Self-check: you can answer (a) approximately how many people have contributed to this repo (you saw the contributors count), (b) when was the most recent commit, (c) write one sentence describing how active the project has been in the past week based on what Pulse shows you (e.g., "about 20 PRs merged and 30 issues opened — very active"). You do NOT need to understand any of the code — the point is that a repo is a public, browsable, time-stamped record of every change ever made to a piece of software.**

Task 3 — Try an AI coding assistant on a real question

AI assistants are, in 2026, the third tool every engineer keeps open all day (after the editor and the browser). Today you'll use one for the exact thing it's best at: explaining a confusing concept in plain language, then turning the explanation into a concrete example.

1. Open whichever AI chat assistant you can reach: claude.ai, chatgpt.com, gemini.google.com, or the playground at cursor.com. Any of them works.
2. Paste in this exact question:

In one paragraph, explain what `git rebase` does and how it differs from `git merge`.

Read the response carefully.

3. Ask a follow-up:

Show me a 4-line shell example for each.

The assistant should produce two short code blocks. Read them — don't run them.

4. Ask one more follow-up of your own choosing — something like *"When would I prefer one over the other on a team project?"* or *"What's the danger of rebasing a branch that other people are already working on?"*.
5. Take a screenshot of the conversation.

✓ **Self-check:** you can answer (a) does the explanation roughly match what you read in this lesson's theory page (and Lesson 04's reading list, if you've done it), (b) did the assistant give you example commands that you could, in principle, run on a real git repo, (c) did the follow-up answer add useful context or just repeat itself? You've now used an AI assistant the way working engineers use one all day: not to write code *for* you, but to compress an hour of Stack-Overflow-reading into a two-minute conversation. You do NOT need to run any of the commands.

What "done" looks like

You finish this homework when you have, in your notes:

- A screenshot of VS Code with the Command Palette open — plus your own short list naming the five sidebar panels you clicked through.

- A screenshot of the **Insights → Pulse** tab on the `facebook/react` repo — plus your own one-sentence answer on contributor count, last-commit time, and "how active is this project this week".
- A screenshot of your AI-assistant conversation about `git rebase` vs. `git merge` — plus a one-line note on whether the explanation matched the theory page.

You've now *touched* the three tools that defined the theory page: the editor (Task 1), the public repository (Task 2), and the AI assistant (Task 3). The fourth tool — git itself — is the invisible glue that ties them together; you'll see it in action when you start your first project. Next lesson we zoom out one more level — from *how* code gets written to *how* the resulting systems are shaped: monoliths, microservices, serverless, event-driven.

Lesson 06: Software Architecture

Lesson 06 — Software Architecture

Monolith / microservices / serverless / event-driven — what these words actually mean.

IT Landscape

Module 2

What does “architecture” mean?

How you arrange the parts of your software, when it gets bigger than one file.

Same problems buildings have:

- **Load** — how many users at once?
 - **Traffic** — how do requests flow?
 - **Repair** — how do you fix things at 3 AM?
 - **Evolution** — how do you add a feature without breaking what's there?
-

Pattern 1 — Monolith

One big app, one codebase, one deploy.

Pros

- Simple to reason about
- One place to debug
- No network between modules

Cons

- One bug can bring everything down
- Teams step on each other
- Scaling means scaling all of it

How almost every successful product was built first.

Pattern 2 — Microservices

Many small apps, each owning one slice, talking via APIs (or queues).

Pros

- Teams ship independently
- Scale only the hot parts
- Different languages per service

Cons

- Every API call is a network call (can fail!)
 - Operational complexity explodes
 - 10× services to monitor, deploy, secure
-

Pattern 3 — Serverless

You ship a **function**; the cloud provider runs it on demand.

- **AWS Lambda** — the original.
- **Cloudflare Workers** — edge-first.
- **Google Cloud Functions, Azure Functions.**

Pros

- No servers to manage
- Pay-per-invocation
- Scales to zero

Cons

- Cold-start latency
- Vendor lock-in
- Surprising bills at scale

Pattern 4 — Event-driven

Components don't call each other — they **publish events** to a **broker**; other components **subscribe**.

- Events: `OrderPlaced`, `PaymentReceived`.
- Brokers: **Kafka**, **RabbitMQ**, **AWS SQS**, **Kinesis**.

Pros

- Decoupled producers and consumers
- Great audit trail
- Replayable

Cons

- Hard to follow one request across N services
- Eventual consistency by default

Worked example — an e-commerce site, 4 ways

Same product, four shapes.

Monolith

One Rails / Django app. One database. Catalog, cart, checkout, admin — all in one process. One deploy.

Microservices

`catalog`, `cart`, `orders`, `payments`, `users` — each its own service, its own DB, talking over HTTP/gRPC.

Serverless

An API gateway routes each endpoint (`/cart`, `/checkout`) to a Lambda function. DynamoDB holds state. Scales to zero overnight.

Event-driven

Checkout publishes `OrderPlaced`. Payments, inventory, email, and analytics all subscribe and react.

Trade-offs at a glance

Shape	Team size that fits	Ops complexity	Scale story	Bill at low traffic	Best for
Monolith	1–30	Low	Bigger box	Low (one VM)	Startups, most products
Microservices	50+	High	Scale hot services only	High (cluster + ops)	Many teams, big orgs
Serverless	Any	Medium	Auto, to thousands	Near zero	Spiky, low-volume APIs
Event-driven	Any	Medium–high	Consumers scale per topic	Medium (broker)	Pipelines, audit, integrations

Anti-pattern — the distributed monolith

Microservices that all have to deploy together. Worst of both worlds.

This is what happens when teams split a monolith too early or along the wrong lines — e.g., by **technical layer** ("the auth service", "the email service") instead of by **business domain** ("the orders service").

- You pay the ops cost of microservices.
 - You keep the development friction of a monolith.
 - The fix is usually long and political.
-

Cost models

- **Monolith on one VM** — fixed monthly bill, scales by buying bigger boxes.
- **Microservices on Kubernetes** — fixed cluster bill plus ops staff to run it.
- **Serverless** — pay per invocation; cheapest at low traffic, can become expensive at very high traffic.

Architecture is also a **finance decision**. The cheapest shape at 100 users/day is rarely the cheapest at 100M users/day.

How to read an architecture diagram

- **Boxes** = services / processes.
- **Arrows** = “calls” (sync) or “publishes to” (async).
- **Cylinders** = databases.
- **Trapezoids** = queues / brokers.
- **Clouds** = managed services / SaaS.

Tip: always read along the arrows from “user action” forward. If you can't trace a request from the user box to the database and back, the diagram (or the system) is hiding something.

Recap

- **Architecture is shape**, not technology.
- **Monolith first** is fine and often right.
- The four words are **tools, not destinations**.

Coming up: Lesson 07 — where the data lives.

Software Architecture

Reading time: ~10 min · Prereqs: Lessons 01–05 · *Snapshot 2026 — concepts are stable; specific brand names move.*

Why this matters to you

When engineers say "architecture", they don't mean buildings — they mean the shape of a software system. The same building blocks (code, databases, networks) can be arranged dozens of ways, and the arrangement matters enormously. A bad architectural choice early on becomes a multi-year, multi-million-dollar headache. A good choice keeps a small team moving fast for a decade. The vocabulary in this lesson — *monolith*, *microservices*, *serverless*, *event-driven* — shows up in every architecture review, every engineering job ad, and every "why is our cloud bill so high" meeting. Once these four words click, you can read most diagrams, follow most technical debates, and ask sharper questions in the rooms where the bills get paid.

1. What "architecture" actually means in software

Architecture is *how* you arrange a system's parts once it's bigger than one file or one team. The same problems buildings have: **load** (how many users?), **traffic** (how do requests flow?), **repair** (how do you fix things at 3 AM?), **evolution** (how do you add a feature without breaking what's there?). The job of a **software architect** — often a senior engineer, sometimes a dedicated role — is to make those trade-offs explicit and write them down before the team commits to them in code. There are perhaps a dozen named architectural patterns; four of them dominate 2026 conversations, and the rest are usually variations or combinations of these. Get the four words and you can follow the room.

2. Monolith — one big app

A **monolith** is one codebase, one process, one deploy. Almost every famous startup began this way, and many still run this way at very large scale.

Pros: easy to reason about (one place to debug, one place to deploy); no network between modules (every function call is in-process — fast and reliable); one team can own the whole thing end-to-end. New engineers can read the codebase top to bottom and understand the system.

Cons: as the app grows, every change has to be rebuilt and re-tested; a bug in one corner can crash the whole process; many teams editing one codebase step on each other's pull requests. Famously, **Shopify**, **GitHub**, and **Basecamp** ran monoliths for years at extraordinary scale. The modern variation is the **modular monolith**: internally well-bounded modules with clear contracts, externally still one deploy. Many teams now see this as the sweet spot for the first several years of a product.

3. Microservices — many small apps

Microservices split the system into many small services, each owning one slice of the business (the "users" service, the "orders" service, the "payments" service) and talking to each other via APIs.

Pros: each team owns its service end-to-end and ships independently; you can scale only the hot parts of the system; different services can use different languages or databases without forcing the rest of the org to switch.

Cons (and they're big): every API call is now a network call — which can fail, time out, or be slow; operational complexity multiplies, because instead of one deploy, one monitoring target, and one log stream, you have dozens; finding the cause of a slow request now means tracing across services. The honest summary: microservices are an *organizational* solution as much as a technical one — they pay off most when you have many teams that need to ship without coordinating every release. Below maybe 50 engineers, the overhead usually outweighs the benefit.

4. Serverless — functions on demand

Serverless means you ship a function, and the cloud provider runs it on demand. Brand names as of 2026: **AWS Lambda** (the original), **Cloudflare Workers**, **Google Cloud Functions**, **Azure Functions**, **Vercel Functions** (under the hood, often Lambda).

Pros: no servers to maintain, no capacity planning, the bill is per-invocation (you pay nothing while idle), and the platform scales effortlessly to thousands of concurrent calls. **Cons: cold starts** (the first call after idle is slow); vendor lock-in (your code now leans on the platform's primitives); debugging is harder (no SSH'ing into a "server"); and at very high steady volume, serverless is often *more* expensive than just running a fixed VM. The rough rule-of-thumb: serverless is a great fit for spiky, unpredictable workloads and a poor fit for steady, high-throughput ones.

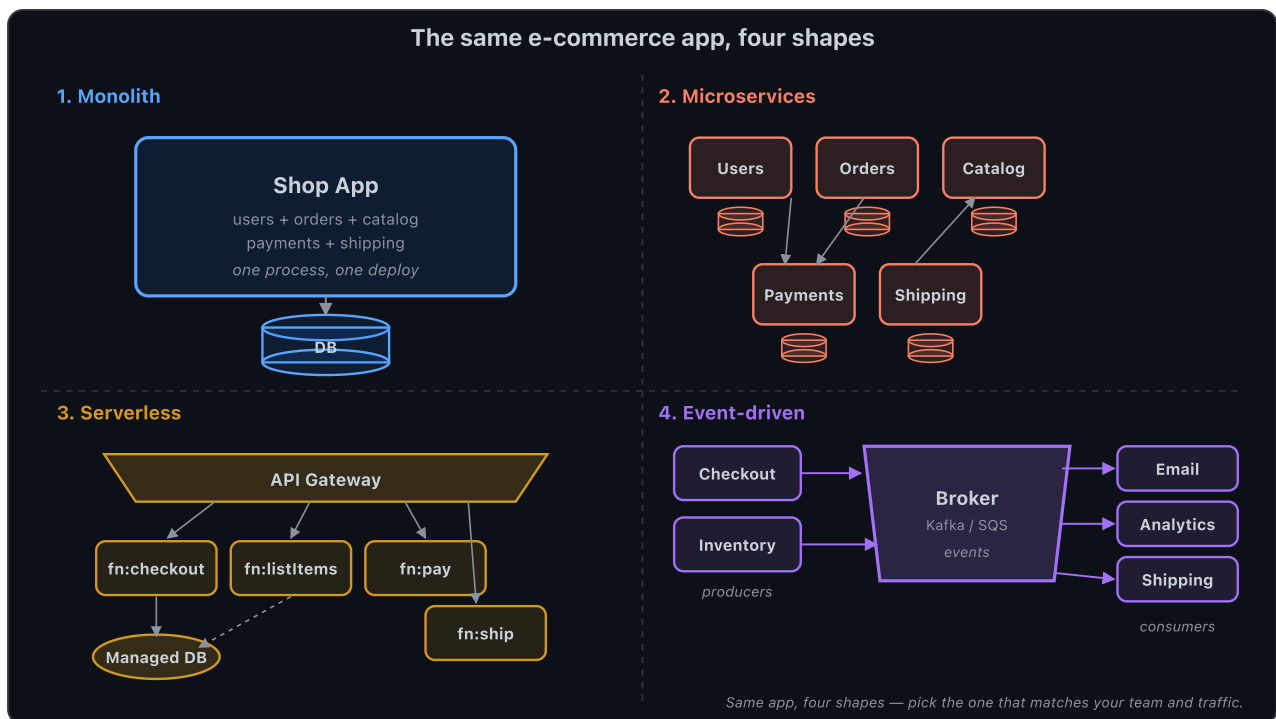
5. Event-driven — publish-subscribe

In an **event-driven** system, components don't call each other directly. Instead, when something happens, the originating component publishes an **event** (e.g., `OrderPlaced`, `PaymentReceived`) to a **broker** — **Kafka, RabbitMQ, AWS SQS / SNS / Kinesis, NATS, Redis Streams**. Other components subscribe to events they care about and react.

Pros: components are decoupled (the publisher doesn't know or care who's listening); you get a natural audit trail in the event log; you can add new consumers without changing the producer. **Cons:** tracing a single user action across N event handlers is hard; *eventual consistency* is the default (the world doesn't update instantly everywhere); ordering and exactly-once delivery are surprisingly tricky to get right. Most large e-commerce, fintech, and logistics systems use events for at least *part* of the workload — rarely the whole thing.

6. The same app, four ways

Here is the same e-commerce application drawn in each of the four shapes. Same boxes, same job, very different arrangements.



7. The anti-pattern: distributed monolith

The most common architectural mistake of the past decade: split the monolith into "microservices" but require all of them to deploy together. This happens when API contracts between services are too brittle, when services share a database, or when services were split by *technical layer* ("the auth microservice", "the email microservice") instead of by *business domain* ("the orders service"). You now have the operational cost of microservices and the development friction of a monolith — the worst of both. The fix is usually long and political. Practical guidance from most senior architects in 2026: **start with a monolith. Split only when team size or scale forces the issue.**

What people argue about

- **"Microservices everywhere" vs "monolith first".** The pendulum has currently swung toward monolith-first, after a decade of microservices fashion. Amazon Prime Video famously moved one service *off* microservices and cut its bill by 90%; the post made the rounds for a reason.
- **Serverless at scale.** Two passionate camps. One: "scales effortlessly, no ops, focus on the product". The other: "your bill scales effortlessly too, and

you can't optimize what you don't run". Both sides have real war stories — the right answer depends on the workload shape.

- **Event-driven by default.** Some teams (especially in finance and logistics) make events the default for everything and treat synchronous calls as the exception. Others say events introduce eventual-consistency bugs that simple synchronous calls would never have. Both are right inside their context.

Reading on

- **Building Microservices** by Sam Newman — the canonical book on the topic; balanced and honest about the costs as well as the benefits.
- **Monolith First** — Martin Fowler's short essay, martinfowler.com/bliki/MonolithFirst.html. Five minutes; the single most-cited argument for starting simple.
- **AWS Well-Architected Framework** — free; one short chapter per architectural concern (reliability, performance, cost, security, operations). A good vocabulary expander even if you're not on AWS.

LAST-REVIEWED: 2026-05-25

Homework — Software Architecture

Estimated time: ~45 min · No code required · Just a browser and a notebook

Three short tool-tour exercises. The goal: turn the four words from the theory page (*monolith*, *microservices*, *serverless*, *event-driven*) into something you can see in real reference architectures, real engineering blog posts, and real cloud pricing. By the end you will have classified a diagram, read a war story, and felt — in dollars — what "pay-per-request" actually means.

Before you start

Keep a one-page notebook open (paper or a text file). For each task, write down the short answers under (a)/(b)/(c). You are not turning this in — you are training the muscle of *reading* architecture diagrams instead of just nodding at them.

The three tasks

Task 1 — Read a real architecture diagram

Open the AWS Architecture Center at aws.amazon.com/architecture. Pick any reference architecture — e.g., *Serverless E-Commerce*, *Web Application Hosting*, or anything from the *Featured* carousel. Look at the diagram for a couple of minutes before reading the description.

Self-check — answer all three:

- **(a)** Classify the architecture into one of the four shapes from this lesson — *monolith*, *microservices*, *serverless*, *event-driven* — or a hybrid (most real ones are). One sentence of reasoning.
- **(b)** Trace one request path with your finger (or cursor) from the user box on the left through to the database / storage on the right. Note the order of

services it passes through.

- **(c)** Name three AWS services in the diagram you already recognise from Lesson 03 (compute / storage / networking basics), and one you have *not* seen before. Look that one up in a single sentence.

✓ **Self-check: you can name the shape, the path, and at least one new service from this single diagram.**

Task 2 — Find a "moved off" or "moved to" microservices story

Search Google or Hacker News for `"monolith to microservices"` OR `"microservices to monolith"`. Pick one engineering-blog post from a real company. The famous Amazon Prime Video *"monolith move"* post is one option, but anything counts as long as it is a first-party story from the team that lived through it.

Self-check — answer all three, two sentences each is enough:

- **(a)** What was the company's *original* architecture? Use the vocabulary from this lesson.
- **(b)** What did they move *to*?
- **(c)** What was the headline reason for the move? Cost, team velocity, reliability, latency, on-call pain — pick the one the blog post actually leads with.

✓ **Self-check: you can explain, in one paragraph, why a real team changed shape — without using the word "scalability" as a hand-wave.**

Task 3 — Feel serverless pricing in dollars

Open `aws.amazon.com/lambda/pricing`. Using either the pricing page itself or the AWS Pricing Calculator linked from it, compute the monthly cost of a function that is invoked **1,000 times per day** with **512 MB** of memory and runs for **200 ms** each. Then re-compute the same function at **1,000,000 invocations per day**.

Self-check — answer all three:

- **(a)** Approximate monthly cost at the low volume (1,000/day). Hint: it will likely be \$0 or a few cents, because of the free tier.
- **(b)** Approximate monthly cost at the high volume (1,000,000/day — roughly 30M invocations/month).
- **(c)** One sentence on what happens to the *cost per call* as volume grows. (Hint: it stays roughly linear — that linearity is the whole point of serverless pricing, and also the thing that bites you when traffic spikes.)

✓ **Self-check: you can quote two numbers (low and high volume) and explain in one sentence why serverless is "cheap until it isn't".**

Wrap-up

If you completed all three tasks, you have done three things that practising engineers do every week: *read* an architecture diagram without panicking, *read* someone else's hard-won lesson about changing shape, and *price out* a cloud workload before committing to it. Those three habits, repeated, are most of what "thinking architecturally" actually looks like in 2026.

Going further (optional)

Repeat Task 1 with a non-AWS diagram — Google Cloud's *Architecture Center* and Azure's *Architecture Center* publish similar reference diagrams. Notice how the shapes (monolith / microservices / serverless / event-driven) are the same even when the service names change. That is the lesson: the vocabulary travels.

Lesson 07: Databases

Lesson 07 — Databases

SQL / NoSQL / vector / search — what to store where, and the cost of getting it wrong.

IT Landscape

Module 2

Why a database, instead of files?

A folder of `.csv` files is a database — for one user. It collapses past that.

- **Many users at once** — concurrent writes corrupt a flat file.
 - **Durable across crashes** — mid-write power loss = broken row.
 - **Queryable** — “give me all orders from yesterday over \$100” — not a grep job.
 - **Enforces structure** — the “email” column is actually an email.
 - **Joins** — combine two datasets by a shared key without writing custom code.
-

SQL — the 50-year incumbent

Invented in the 1970s. Still the most-deployed kind of database in the world.

Brands you'll hear (as of 2026):

- **PostgreSQL** — free, broadly loved, default for new projects.
 - **MySQL / MariaDB** — free, classic LAMP stack.
 - **Microsoft SQL Server** — the enterprise Windows world.
 - **Oracle** — paid, enterprise, deeply entrenched.
 - **SQLite** — a database in a single file. Runs in your phone, your browser, your IDE.
-

Tables, rows, queries

A tiny SQL query:

```
SELECT name, email  
FROM users  
WHERE country = 'UA'  
ORDER BY name;
```

*You don't need to write this — just see the shape:
column names, a filter, an ordering.*

The `users` table:

name	email	country
Olena	olena@example.com	UA
Petro	petro@example.com	UA
Jane	jane@example.com	US

NoSQL — when SQL doesn't fit

In the 2000s, web companies hit limits with single-machine SQL:

- Too much write traffic.
- Too many JSON-shaped objects.
- Too rigid a schema for fast-evolving products.

NoSQL is a catch-all for the alternatives that emerged. There are **four main shapes** — the next four slides.

Document stores — MongoDB & friends

Store **JSON-like documents** instead of rows. Schema can vary per document.

- **Good for:** content (blog posts), product catalogs, user profiles with varying fields.

Names you'll hear (as of 2026):

- **MongoDB** — the icon.
 - **DynamoDB** (AWS), **Firestore** (Google), **Cosmos DB** (Azure).
 - **Couchbase**.
-

Key-value — Redis & friends

Just a giant **hash map**. Lives in memory. Ridiculously fast.

```
SET user:42:session "abc123"  
GET user:42:session    -> "abc123"
```

Used as:

- Cache, session store, leaderboard, rate limiter, pub/sub bus.

Names (as of 2026): **Redis** (king), **Valkey** (fork), **Memcached** (older), **DynamoDB** (also key-value).

Search — Elasticsearch & friends

When you need **full-text search** — typos, synonyms, ranking by relevance — SQL `LIKE` `'%foo%'` is the wrong tool.

Search engines build **inverted indexes** over text and rank results.

Names (as of 2026):

- **Elasticsearch** (and **OpenSearch**, the AWS fork) — the heavyweight.
 - **Algolia** — SaaS, dev-friendly.
 - **MeiliSearch**, **Typesense** — lightweight upstarts.
-

Vector databases — for AI (as of 2026)

LLMs and embeddings need to search by **semantic similarity**, not keyword.

Vector DBs store high-dimensional vectors (~1024 numbers per item) and find **nearest neighbors** fast.

Names (as of 2026):

- **Pinecone** — SaaS pioneer.
- **Weaviate, Qdrant, Chroma.**
- **pgvector** — Postgres extension; the “just use Postgres” choice.

This space is the most volatile — vendors come and go quarterly. Refresh annually.

Picking the right one

If your data looks like ...	Reach for ...
Transactional / accounts / money	SQL (Postgres)
User-shaped data with varying fields	Document store (Mongo)
Session, cache, leaderboard	Key-value (Redis)
Search bar with typo tolerance	Search engine (Elasticsearch / Algolia)
RAG — “find similar to this prompt”	Vector DB (pgvector / Pinecone)

The cost of getting it wrong

- **Data loss** — a NoSQL store dropped writes during a primary failover (sometimes silently).
- **Wrong shape forever** — denormalized so hard that adding a new query takes weeks.
- **Egress cost** — cross-region replication = five-figure monthly cloud bill.
- **Schema rigidity** — changing a column type in a 1 TB SQL table = hours of downtime.
- **Search relevance** — “nobody can find anything in our app” — a 6-month fix.

Database choice is a long-tail commitment. You will live with this one for years.

"Just use Postgres"

A common 2026 take. Also serious advice.

Modern PostgreSQL handles:

- **JSON** — like a document store (`JSONB`).
- **Key-value** — `UNLOGGED` tables or `hstore` .
- **Full-text search** — built-in `tsvector` .
- **Vectors** — `pgvector` extension.
- **Time-series** — TimescaleDB extension.

One database, fewer moving parts. *Caveat: at extreme scale, specialized stores still win.*

Recap

- Databases are **not just “SQL”** — five families, each with a sweet spot.
- The right database is the one that **matches your query patterns**.
- For most teams, **“just use Postgres”** is the right starting point.

Coming up: Plan C / Module 3 — getting software actually running for real users.

Databases

Reading time: ~10 min · Prereqs: Lessons 01–06 · *Snapshot 2026 — concepts are stable; the vector-DB wing especially is fluid.*

Why this matters to you

Databases are the most boring-sounding, most consequential decision in software. Get the database right and the rest of the app stays tractable for a decade — new features slot in, queries stay fast, the on-call rotation stays quiet. Get it wrong and every feature request comes with a multi-week migration tax, the cloud bill creeps up, and "we can't change that because of the database" becomes a recurring meeting note. This lesson is the survey: what kinds of databases exist, what each is good at, and why a meaningful chunk of senior engineers *as of 2026* will roll their eyes and say "just use Postgres" before letting you pick anything else. After this lesson you should be able to read a database name in an architecture diagram and instantly know what kind of workload it's solving for.

1. Why a database, instead of files?

A folder of `.csv` files *is* a database — for exactly one user, on exactly one machine, with no failures. It collapses immediately past that under five forces. **Many users at once**: two concurrent writes to the same CSV corrupt the file. **Durability**: one crash mid-write and your file is half-old, half-new and unreadable. **Query speed**: with no index, every "find all orders over \$100" scans every row, every time — fine at a thousand rows, agony at a billion. **Structure**: nothing stops you from putting `"not-an-email"` in the email column. **Reasoning across data**: joining two CSVs by a foreign key is a multi-hour ad-hoc script. Databases solve all five of these as their day job. The price you pay is a separate piece of infrastructure to install, manage, back up, and (eventually) scale — which is why this whole lesson exists.

2. SQL — the 50-year incumbent

SQL (Structured Query Language) was invented at IBM in the 1970s and is still the most-deployed shape of database in the world. SQL databases — **PostgreSQL**, **MySQL** / **MariaDB**, **Microsoft SQL Server**, **Oracle**, **SQLite** — store data in **tables** with fixed columns. Each row in a table has the same shape; if you want to add a new field, you alter the table. You query with a declarative language: the famous `SELECT name FROM users WHERE country='UA'` — you describe *what* you want, not *how* to get it, and the engine figures out the fastest plan.

SQL databases have decades of work behind them on three things that matter enormously and are extremely hard to build. First, **ACID transactions** — writes are all-or-nothing, even across crashes, even when multiple users are writing at once. Second, **referential integrity** — foreign keys make it impossible to have an order pointing to a customer that doesn't exist. Third, **query planning** — the engine inspects your query and your indexes and picks the fastest way to answer, often beating what a human would write by hand.

The simple rule: for **anything involving money, accounts, identity, or other "I'd rather lose my hair than lose this record" data**, you want a SQL database.

PostgreSQL is the default choice for new projects *as of 2026*; it's free, broadly loved, and increasingly does everything (see "just use Postgres" below). Think of a SQL database as a *well-organized filing cabinet*: every drawer (table) has a fixed set of labeled folders (columns), every document (row) goes in exactly one place, and you have a librarian (the query planner) who can pull any cross-reference in milliseconds.

3. The NoSQL family

In the 2000s, web companies (Google, Amazon, Facebook) hit limits with single-machine SQL: too much write traffic for one box, too many JSON-shaped objects that didn't want to be tables, too rigid a schema for products that pivoted every quarter.

NoSQL is the catch-all for the alternatives that emerged. The name is a misnomer (it's not "anti-SQL", more "not-only-SQL") but it stuck. Four useful shapes show up in 2026:

- **Document stores** — store JSON-like documents instead of rows; schema can vary per document. Brand names *as of 2026*: **MongoDB** is the icon; **DynamoDB** (AWS), **Firestore** (Google), **Cosmos DB** (Azure), **Couchbase**. Good fit for user profiles, product catalogs, and content where every item is shaped slightly differently. Analogy: a shoebox of receipts instead of a spreadsheet.

- **Key-value stores** — a giant hash map. `SET user:42:session "abc..."` / `GET user:42:session`. Microseconds-fast because they usually live in memory. **Redis** is king as of 2026; **Valkey** is its open-source fork; **Memcached** is the old guard. Used as caches, session stores, leaderboards, rate limiters, and lightweight pub/sub buses.
- **Wide-column stores** — **Cassandra**, **ScyllaDB**, **HBase**, **BigTable**. Built for huge, write-heavy, time-series-ish data spread across many machines. Less common in everyday application work; common in telemetry, IoT, and ad-tech pipelines.
- **Graph databases** — **Neo4j**, **Memgraph**, **Amazon Neptune**. For relationship-heavy data: social networks, fraud detection, knowledge graphs. If your central question is "how is X connected to Y, two hops away?", a graph DB answers it natively where a SQL JOIN would crawl.

4. Search engines

When you need full-text search — typos, synonyms, relevance ranking, faceted filtering — SQL's `LIKE '%foo%'` is the wrong tool (it's slow and dumb about meaning). Search engines build **inverted indexes** over text (for every word, the list of documents it appears in) and rank results by relevance. Brand names as of 2026: **Elasticsearch** (and its AWS-led fork **OpenSearch**) is the heavyweight; **Algolia** is the SaaS-friendly choice with the fastest "drop it in and it works" story; **MeiliSearch** and **Typesense** are lightweight, developer-friendly upstarts. If your app has a search bar that needs to feel like Google's — tolerant of typos, ranked by relevance, fast on millions of records — you want one of these alongside your primary database, not instead of it.

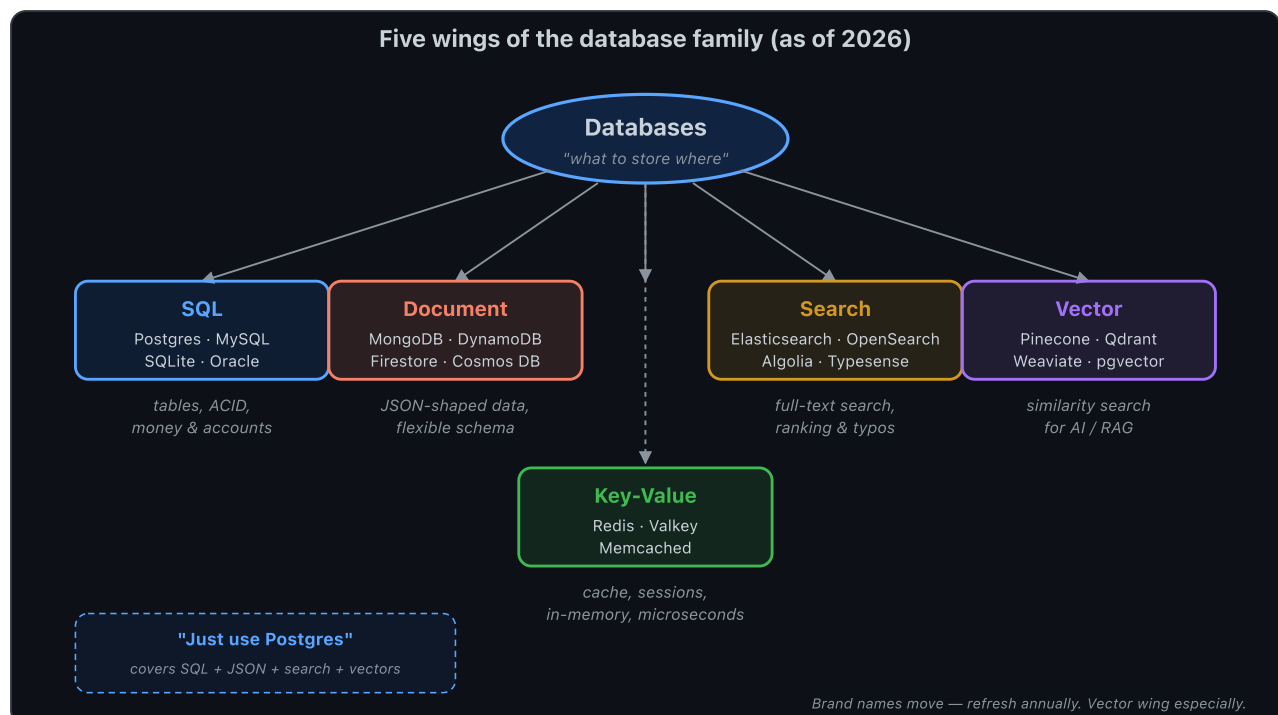
5. Vector databases — the new wing

New as of 2026, born of the LLM era. LLMs and embedding models turn text (or images, or audio) into high-dimensional vectors of numbers — ~1024 dimensions is typical. Searching by **semantic similarity** ("find me documents that mean roughly this") becomes a problem of finding the nearest neighbors of a query vector in that high-dimensional space. Specialized stores have optimized indexes for exactly that. Names as of 2026: **Pinecone** (the SaaS pioneer), **Weaviate**, **Qdrant**, **Chroma** (lightweight, developer-friendly), and **pgvector** (a Postgres extension — the "just use Postgres" path). Every RAG-based AI app (Lesson 13 returns to RAG in depth) uses

one. This whole section is on [refresh-checklist.md](#) as high-churn — vendors come and go quarterly, and the 2027 list will look different.

6. The database family, one diagram

Here is the same idea drawn as a taxonomy: a central *Databases* node branching into five wings, each with a couple of representative brand names *as of 2026* and a one-line use-case caption. When you see one of these names in an architecture diagram, you can place it on this tree.



7. The cost of getting it wrong

Database choice is a long-tail commitment. A handful of failure modes show up over and over in postmortems. **Silent data loss:** some NoSQL stores have, historically, lost writes during a primary failover — sometimes silently, which is the worst kind. **Wrong shape forever:** denormalized so hard, or schema-locked so rigidly, that adding a new query pattern takes weeks of migration. **Egress cost:** cross-region replication of a chatty database runs up a five-figure monthly cloud bill that nobody noticed in the design doc. **Schema rigidity:** changing a column type in a 1 TB SQL table takes hours of downtime if you didn't plan for it. **Search relevance:** "our users say they can't find anything" is sometimes a six-month engineering project to fix. This is why the next section exists.

8. "Just use Postgres"

A 2026 meme that's also serious advice. Modern PostgreSQL handles: relational data (its day job), JSON documents (with `JSONB`), full-text search (built-in `tsvector`), vectors (via `pgvector`), time-series (via **TimescaleDB**), and key-value (cheap `UNLOGGED` tables or `hstore`). For most teams, starting with one Postgres instance and adding specialized stores only when proven necessary keeps the operational surface small, the cost predictable, and the team's mental load manageable. At very large scale, or for genuinely extreme query patterns, specialized stores eventually earn their keep — but most products never reach that point, and the ones that do can migrate later with much better information than they had on day one.

What people argue about

- **SQL vs NoSQL.** The 2010s religious war has largely died. The honest take as of 2026: most apps need a SQL database first; some apps benefit from adding a NoSQL store later for specific shapes of data (a Redis cache, a Mongo store for user-generated documents). "Pick one and use it for everything" is rarely the right answer.
- **Vector DBs vs pgvector.** Specialized vendors say they scale better and offer faster nearest-neighbor indexes; the "just use Postgres" camp says for most teams that's premature optimization. Real workloads in 2026 increasingly favor pgvector at small-to-medium scale and specialized DBs at very large scale — with the crossover point moving each year.
- **NewSQL.** A real-but-niche third category: **CockroachDB**, **Spanner**, **YugabyteDB** — SQL semantics with horizontal scaling across many machines. Worth knowing exists; rarely the right first choice.

Reading on

- **Designing Data-Intensive Applications** by Martin Kleppmann — the canonical modern book on the trade-offs in this lesson. If you read one technical book this year, this one pays back the time many times over.

- **Use The Index, Luke!** — use-the-index-luke.com. Free; teaches the one thing that turns out to matter most about SQL performance (indexes, and when they're not used).
- **PostgreSQL documentation** — postgresql.org/docs. Surprisingly readable and well-organized; a good way to absorb what "modern Postgres" can actually do.

LAST-REVIEWED: 2026-05-25

Homework — Databases

Estimated time: ~50 min · Almost no code · A browser, a free account or two, and a notebook

Three tool-tour exercises. The point is to *touch* three very different shapes of database in your own browser: a classic SQL engine, a document store, and a vector / similarity-search world. By the end you'll have run an actual SQL query, opened a real JSON document inside MongoDB, and seen — without writing a line of code — what makes vector search different from keyword search.

Before you start

Keep a one-page notebook open (paper or a text file). For each task, write the short answers under (a)/(b)/(c). You are not turning this in — this is the muscle of *poking at* a database, not just reading about one. The only "code" in this homework is modifying *one* clause of one pre-written SQL query in Task 1.

The three tasks

Task 1 — Touch a SQL database in the browser

Open sqliteonline.com (or db-fiddle.com if you prefer Postgres/MySQL).

Pick the demo *Chinook* database, or any sample dataset that comes preloaded — most of these sites ship one. Run the example query that's already sitting in the editor; you should see a small table of results appear below it.

Now change **one thing**. You do not need to know SQL. Examples of a single-word change:

- add a `WHERE` clause, e.g. `WHERE Country = 'USA'`
- change `LIMIT 10` to `LIMIT 100`

- change the table name in `FROM` to a different table from the schema panel on the left

Re-run the query. Notice that the result table changes shape.

Self-check — answer all three:

- **(a)** How many rows did the *original* query return?
- **(b)** How many rows did your *modified* query return?
- **(c)** Take a screenshot of one query and its result table for your own notes.

✓ You do NOT need to know SQL — you only need to see that a query produced a table, and that changing one clause changed the table.

Task 2 — Tour MongoDB Atlas (free tier)

Go to `cloud.mongodb.com` and sign up. The **M0** free tier needs no credit card. Create a free cluster — call it `landscape` — and wait until it shows *Active* (usually 1–3 minutes).

Then click **Browse Collections** → **Load Sample Dataset**. Wait another ~2 minutes for it to import. Once it's done, open the `sample_mflix` database → `movies` collection. Click any movie document to expand it — you'll see a tree of fields like `title`, `year`, `cast`, `genres`. Click a few more documents and notice their fields are *not* identical — some have `awards`, some don't; some have `tomatoes`, some don't.

Self-check — answer all three:

- **(a)** One sentence: how does this differ from a SQL table? (Hint: think about the *shape* of each row.)
- **(b)** How many documents are in the `movies` collection? (Atlas shows the count in the collection header.)
- **(c)** Take a screenshot of one document expanded so its JSON shape is visible.

Cleanup

When you're done, you can delete the cluster from **Database** → ... → **Terminate**. The M0 free tier is fine to keep running — it never charges — but if you don't expect to come back, tear it down.

Task 3 — See vector / similarity search in action

Open claude.ai (or chat.openai.com or any chatbot you have access to). Ask:

"Find me five animals that are conceptually similar to an octopus but are not cephalopods."

Read the answer. Notice what just happened: the model didn't grep Wikipedia for the literal string *"octopus"*. It found things that are similar in *meaning* — intelligent, soft-bodied, aquatic, problem-solving. That's the core of how a vector database feels from the outside.

Then open github.com/pgvector/pgvector. Read just the **Getting Started** section of the README — don't try to install anything, just read. Pay attention to the `<->` operator in the example queries.

Self-check — answer all three:

- **(a)** Does the chatbot's answer feel like *keyword* search or like *similarity* search? One sentence on why.
- **(b)** What does the `<->` operator do in pgvector? (One short sentence from the README.)
- **(c)** Write one sentence on why a vector database is a natural fit for retrieval-augmented generation (RAG). Lesson 13 returns to this topic in depth.

What "done" looks like

You're done when, in your notebook, you have:

- three row-count numbers and one screenshot from Task 1 (SQL)
- one shape-difference sentence, one document count, and one screenshot from Task 2 (MongoDB)
- three short sentences from Task 3 (vector / similarity)

The aim was never to make you a DBA in 50 minutes. It was to make sure that the next time someone says *"we'll just shove it in Postgres"* or *"that needs a vector store"*, you have a picture in your head of what they're actually pointing at — because you've now opened all three.

Where this goes next

Module 3 puts data in motion: how it travels over the network (Lesson 10), how systems talk to each other (Lesson 11), and how all of this lives in the cloud (Lesson 12). Module 4 then comes back to databases through the AI lens — embeddings, vector indexes, and RAG — so Task 3 is a foretaste, not a deep dive.

Lesson 08: Networking Infrastructure

Lesson 08 — Networking Infrastructure

Streets, highways, and addressing — the physical + logical layer that everything else rides on.

IT Landscape

Module 2

Networks are roads

A network is just a road system. Data travels in packets — think envelopes in a postal truck.

- **Street** — the Ethernet cable from your laptop to the wall.
 - **Motorway** — the fiber-optic trunk line from your city to the next.
 - **Address** — every device has one (IP address). Every intersection has one too (MAC address).
 - **Post office** — the router. It reads the destination and forwards the packet along the best route.
-

Wired vs wireless

Type	Medium	You see it when...
Ethernet	Copper cable (Cat5/6/7)	Plugging a laptop into an office wall port
Fiber	Glass / light pulses	Your ISP digs up the street to install it
Wi-Fi	Radio waves (2.4 / 5 / 6 GHz)	The router in your living room
Cellular	Radio (4G / 5G towers)	Mobile data on your phone
Satellite	Microwave to orbit	Starlink in remote areas

Two kinds of address

MAC address

Burned into the hardware at the factory.

Like a serial number on a piece of furniture.

Works only on your **local network** (one room / one office floor).

IP address

Assigned by the network. Can change.

Like a postal address — move house, change address.

Private (192.168.x.x / 10.x.x.x) inside your home; **public** (your ISP's IP) facing the internet.

*Your home has **one** public IP but possibly dozens of private IPs. NAT (Network Address Translation) is the bouncer who manages traffic between them.*

Three network worlds

World	Typical size	Who manages it
SOHO	1–20 devices	You — after watching a YouTube tutorial
Enterprise	Hundreds → thousands of devices	A dedicated network team
Data Center	Tens of thousands of servers	NetOps team + custom firmware + 24/7 NOC

The physics are the same. The requirements — speed, redundancy, management — are wildly different.

What changes at scale

- **SOHO:** one router does everything. 1 Gbps is plenty. A 30-minute outage is annoying but survivable.
- **Enterprise:** dedicated switches, VLANs (logical segmentation), redundant uplinks, 24-hour SLA to fix failures.
- **Data center:** 100 Gbps links per rack, N+2 redundancy everywhere, fully meshed spine-leaf topology, no single cable break can drop anything.

At SOHO scale, "the router rebooted" is a root cause. At DC scale, a single reboot in a 10,000-server cluster isn't even a ticket.

VPN — what it actually is

A VPN is an encrypted tunnel punched through a public network so you can act as if you were physically inside a private one.

- **Corporate VPN** (WireGuard, Cisco AnyConnect): you connect from a coffee shop, and your laptop is “inside” the office — same resources, same firewall rules.
 - **Consumer “VPN” services** (NordVPN, ExpressVPN): reroute traffic through their servers. Hides your IP from websites; does not make you anonymous. Your VPN provider sees everything instead.
-

Tor — the real anonymizer

Tor bounces your traffic through **three volunteer relays** worldwide before it exits.

- Each relay strips one layer of encryption — like peeling an onion (hence the logo).
- No single relay knows both who you are *and* where you're going.
- Trade-off: **very slow** (3 relays, often 300 ms round-trip).

A consumer VPN moves trust from your ISP to the VPN vendor. Tor moves trust from any single party to the assumption that at least two of three relays are honest.

SD-WAN — software-defined wide area networks

The problem it solves: corporations used to buy expensive private MPLS lines to link branch offices. SD-WAN uses the public internet instead — with software that intelligently routes traffic.

- Same reliability guarantees, at a fraction of the cost.
 - Branch in a new city? Buy an SD-WAN box, plug it in, configure from a cloud dashboard in minutes.
 - Brands: **Cisco Viptela, VMware VeloCloud, Palo Alto Prisma SD-WAN.**
-

Watching the pipes

Networks are living things. Someone has to watch them.

- **NMS** (Network Management System) — polls every device for its health.
 - **SNMP** — the 1988-era protocol that reports metrics (CPU, port up/down, error counts) to the NMS. Still ubiquitous.
 - **NetFlow / sFlow** — samples actual traffic to show who's talking to whom, in how much volume.
 - **Observability tools** (Grafana + Prometheus, PRTG, SolarWinds) — dashboards, alerts, historical charts.
-

A day in NetOps

- 09:00 — review overnight NMS alerts. One switch port on Floor 3 flapping; open a ticket.
 - 10:30 — help a developer who says “something is slow”. Pull NetFlow data; show it's a misconfigured firewall rule, not the network.
 - 13:00 — push a new SD-WAN config to the new branch in Warsaw.
 - 15:00 — capacity review: two uplinks trending toward 70% utilisation; order an upgrade for next quarter.
 - 17:00 — patch firmware on edge routers in a maintenance window.
-

The mental model

Three network worlds on one canvas — see Theory page for the labeled diagram

Layer	SOHO	Enterprise	Data Center
Link speed	1 Gbps	10 Gbps	100 Gbps+
Redundancy	None (reboot to fix)	Dual uplinks	N+2, fully meshed
Management	Browser UI	CLI + dashboards	API-driven, scripted
Downtime tolerance	Minutes OK	Seconds, SLA-bound	Zero, 99.999% SLA

Common misconceptions

- **"VPN = anonymity."** No. A consumer VPN moves trust from your ISP to the VPN vendor. Tor is the closer approximation to anonymity.
 - **"Wi-Fi = internet."** Wi-Fi is the last-meter radio link. The internet is the vast public network your router connects to.
 - **"Fiber is fast because of magic."** It's fast because glass conducts light, and light doesn't heat up like copper, so you can push much more signal without signal degradation over long distances.
 - **"The network is someone else's problem."** Every application developer eventually hits a network constraint. Knowing the basics saves weeks of blaming the wrong layer.
-

Recap

- Networks are road systems. Packets are envelopes. Routers are post offices.
- Every device has two addresses: a hardware MAC and a logical IP.
- SOHO, enterprise, and data-center networks use the same physics but completely different operational disciplines.
- A VPN is an encrypted tunnel, not an anonymizer. Tor is closer to an anonymizer.
- SD-WAN replaced expensive MPLS leases with software-managed public internet links.
- Network operations means constant monitoring, alerting, and capacity planning — not just “plugging in cables.”

Coming up: Lesson 09 — Security Fundamentals (or the next lesson in Module 2).

Networking Infrastructure

Reading time: ~10 min · Prereqs: Lessons 01–07 · *Snapshot 2026 — physical standards evolve slowly; SD-WAN and observability tooling moves faster.*

Why this matters to you

Every digital thing you have ever used — a web page, a Slack message, a video call, a payment going through — travelled over a network at least once. Networking is the infrastructure underneath all other infrastructure, and yet it's the layer most non-technical people treat as magic. That opacity is a problem. When a product manager says "it's slow," when a cloud bill triples for no obvious reason, when a security team says "we found lateral movement in the network" — understanding what a network actually is turns those phrases from noise into something actionable. This lesson is the mental model: the road analogy, the three very different worlds networks live in, and the mechanisms — VPN, SD-WAN, monitoring — that engineers use to manage them. No configuration, no commands. Just the map.

1. Streets, highways, and addressing

The cleanest analogy for a network is a road system. **Data travels in packets** — small, numbered envelopes each carrying a fragment of the message. The packet knows where it came from and where it's going. **Routers** are post offices: they look at the destination address on each envelope and decide which road leads toward it. No single router needs to know the full route; it only needs to know the next hop, and the next post office takes over from there.

The physical medium underneath can be copper (Ethernet cables, the Cat5/6/7 patch leads you see in offices), glass (**fiber optic**, which carries pulses of light and can run for hundreds of kilometres without signal degradation), radio waves (**Wi-Fi** in the 2.4, 5, and 6 GHz bands inside buildings), cellular radio (**4G/5G** via towers), or microwave beams bounced off satellites (**Starlink** and similar). The physics differ enormously — fiber is dense and fast; satellite adds 500–600 ms of delay because the signal

physically travels to orbit and back — but the addressing system on top is the same everywhere.

Every device has two kinds of address. A **MAC address** is burned into the network card at the factory: a 48-bit serial number that looks like `3C:22:FB:4A:01:0D`. It works only on your local network segment — the equivalent of a furniture serial number that is only meaningful in the building where the furniture lives. An **IP address** is assigned by the network and can change. Your home router gives every device a *private* IP address in the reserved ranges like `192.168.x.x` or `10.x.x.x`. Your internet provider gives your router a single *public* IP that the rest of the internet sees. The mechanism that maps many private addresses to one public IP is called **NAT** (Network Address Translation) — think of it as an apartment building with one street address and an internal mailroom that routes parcels to the right flat.

2. Three network worlds: SOHO, Enterprise, Data Center

The physics of networking are the same at every scale, but the engineering discipline required grows by orders of magnitude. It is useful to separate the world into three contexts.

SOHO (Small Office / Home Office) is the living-room router most people know. One box does everything: it is the router, the Wi-Fi access point, the firewall, and usually the DHCP server that assigns IP addresses. Top speed is typically 1 Gbps on the wired side, maybe 2–6 Gbps on Wi-Fi 6E. If it breaks, you reboot it. The entire network fits in the palm of your hand.

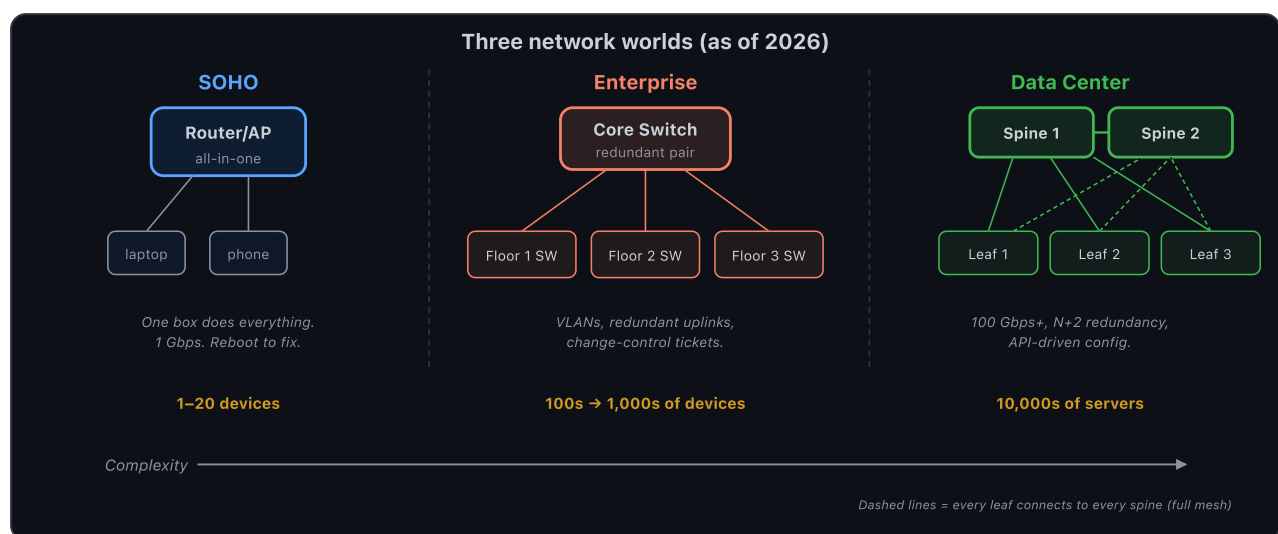
Enterprise networks cover an office campus or a company's national footprint. Here, dedicated switches connect hundreds or thousands of devices. The network is carved into **VLANs** (Virtual Local Area Networks) — logical segments that keep, say, the finance team's traffic separated from the guest Wi-Fi. Redundant uplinks ensure that a single cable cut does not take down a floor. A dedicated team writes change-control tickets before touching anything, because a misconfiguration in a core switch can silence an entire building.

Data-center networks are a different species. A large hyperscaler like Meta or Google runs tens of thousands of servers per building, connected at **100 Gbps or 400 Gbps** per link. The topology is typically *spine-leaf*: every leaf switch (serving a rack of servers) connects to every spine switch, so no path through the fabric is more than two

hops. Redundancy is N+2: two complete failures can happen simultaneously without affecting service. Links, switches, power supplies, and cooling circuits are all doubled or tripled. A data-center network engineer writes code — configurations are pushed via APIs and version-controlled like software — not clicked through a browser UI.

3. The three worlds, one diagram

Here is a side-by-side sketch of the three network worlds: their physical scale, the devices involved, and the gap in management complexity. The leftmost column is the living-room router. The rightmost is a data-center spine-leaf fabric.



4. Going private through public pipes: VPN and anonymizers

A **VPN** (Virtual Private Network) is an encrypted tunnel punched through a public network so that you can act as if you were physically inside a different, private one. Think of it as a pneumatic tube inside the postal system: your envelope goes into a sealed, opaque tube at one end, travels invisibly through the public mail system, and emerges at the other end as if it had never left the building. Nobody watching the public mail can read the contents — or even tell what building it came from.

Corporate VPNs (WireGuard, Cisco AnyConnect, Palo Alto GlobalProtect) solve a specific, legitimate problem: a remote worker in a coffee shop needs to reach internal company resources that are not exposed to the public internet. The VPN client on their laptop creates an encrypted connection back to the company's VPN gateway; from that point on the laptop looks as if it is sitting on the office LAN. This is also why IT

departments can enforce security policies on remote workers — once inside the tunnel, the traffic flows through the same firewalls and monitoring tools as in the office.

Consumer "VPN" services (NordVPN, ExpressVPN, Mullvad, and dozens of others) use the same technology for a different goal: hiding your IP address from the websites you visit. Your traffic exits not from your home, but from the VPN vendor's server in whichever country you chose. This is useful for bypassing geographic content restrictions, and it does prevent your ISP from seeing your browsing history. But it does not make you anonymous: the VPN vendor now sees everything your ISP used to see. You have moved trust, not eliminated it. The marketing phrase "military-grade encryption" is technically correct (AES-256 is the same cipher used by governments) and practically meaningless for most consumer use-cases where the threat model does not include governments decrypting traffic.

Tor (The Onion Router) is a different architecture designed for stronger anonymity. Your traffic passes through three volunteer-operated *relay nodes* worldwide, each of which strips one layer of encryption — like peeling an onion. The entry node knows who you are but not your destination. The exit node knows your destination but not who you are. No single node sees the full picture. The trade-off is speed: three relays, often on different continents, add 200–600 ms of latency. Tor Browser is a modified Firefox pre-configured to use the Tor network; browsing ipinfo.io from it shows a random exit node's IP in a country you may never have visited.

5. SD-WAN: software-defined networks for enterprises

Until the mid-2010s, companies with multiple office locations connected them with **MPLS** (Multiprotocol Label Switching) circuits — private, carrier-managed links guaranteed to be fast and reliable. The catch: expensive, slow to provision (weeks to months), and inflexible. A new office in a city meant waiting for the carrier to run a circuit.

SD-WAN (Software-Defined Wide Area Network) replaced MPLS with intelligence layered on top of ordinary broadband internet links. An SD-WAN device at each office constantly measures the quality of all available paths (ISP A, ISP B, cellular 5G backup) and routes traffic intelligently: video calls over the lowest-latency path, bulk backups over the cheapest path, financial transactions over the most reliable. A new branch office needs only a box that is shipped pre-configured; it calls home to the SD-WAN

cloud controller, downloads its policy, and is live in minutes. Companies like Cisco (Viptela), VMware (VeloCloud), and Palo Alto (Prisma SD-WAN) dominate the market as of 2026. Cost savings of 60–80% over MPLS are common, which is why the enterprise WAN market flipped quickly.

6. Watching the pipes: monitoring and administration

A network that nobody is watching is a network that fails silently. Network operations (NetOps) is a continuous practice of measurement, alerting, and planned maintenance.

The oldest tool in the kit is **SNMP** (Simple Network Management Protocol), invented in 1988 and still running in almost every enterprise network in 2026. Each network device (switch, router, firewall) runs an SNMP agent that reports metrics — port up/down status, error counts, CPU load, bandwidth utilisation — to a central **NMS** (Network Management System). Think of SNMP as each post office in the road network sending a hourly status report to a head-office dashboard.

NetFlow (Cisco) and its open-source equivalent **sFlow** take a different approach: instead of health metrics, they sample actual traffic. Every five minutes you get a report that says "these 20 IP pairs exchanged these volumes of traffic on these ports." That is how a network engineer discovers that a server is sending 40 GB of data to an IP address in an unexpected country at 2 AM — or that a developer's test job is accidentally saturating the office uplink.

Modern observability stacks layer dashboards (Grafana), alerting (Prometheus / Alertmanager), and log aggregation (Elasticsearch or Loki) on top of these feeds. When an alert fires, the on-call network engineer can open a dashboard, immediately see which link or device is misbehaving, correlate it with a recent change, and escalate or fix. This is not magic; it is years of work building the instrumentation. The networks that have it are dramatically easier to operate than the ones that don't.

What people argue about

- **VPN as privacy vs VPN as security theater.** Security professionals generally agree that consumer VPNs are oversold: they protect one threat model (ISP visibility) while doing little for others (browser fingerprinting, tracking pixels, account-level tracking). The counter-argument is that moving trust away from

your ISP — which may log and sell browsing data — to a vendor whose business model depends on not being caught logging is still a meaningful improvement for many users.

- **SD-WAN vs SASE.** As of 2026 the industry is actively debating whether SD-WAN is a stepping stone or a destination. **SASE** (Secure Access Service Edge) merges SD-WAN networking with cloud-delivered security (firewall-as-a-service, zero-trust access). Some vendors say SASE makes standalone SD-WAN obsolete; others say SASE is SD-WAN with a marketing rebrand and a larger bill.
- **IPv4 exhaustion vs IPv6 adoption.** The internet ran out of IPv4 addresses years ago; the fix is IPv6, which has astronomical address space. Adoption is slow because it requires updating every device and software stack. Most networks as of 2026 run "dual-stack" — both protocols simultaneously — in an awkward middle period that has lasted two decades longer than anyone planned.

Reading on

- **Computer Networking: A Top-Down Approach** by Kurose & Ross — the standard university textbook, surprisingly readable for the first three chapters even without a CS background. Chapters 1–2 cover the material in this lesson at greater depth.
- **Cloudflare Learning Center** — cloudflare.com/learning. Free. Cloudflare publishes extremely clear explainers on every networking topic: DNS, TLS, DDoS, VPNs, SD-WAN. Start with "What is a VPN?" and "What is SD-WAN?" — both are accurate and ad-free.
- **The Tor Project** — torproject.org/about/history. The project's own history page explains the design rationale plainly and is written for non-technical readers.
- **Julia Evans's "How do computers get IP addresses?"** — jvns.ca. Evans is remarkable at making low-level networking concepts feel approachable. Any of her networking zines or blog posts are worth an hour.

LAST-REVIEWED: 2026-06-02

Homework — Networking Infrastructure

Estimated time: ~45 min · No code required · A browser, your home router, and optionally a free download

Three tool-tour exercises. The goal is to turn the abstract picture from the theory page into things you can actually observe: your own connection type, the SOHO world inside your home router, the scale of a data-center campus, and what the Tor anonymizer looks like from the outside. No installs are required for Tasks 1 and 2. Task 3 involves downloading Tor Browser — a free, widely-used application that you can uninstall when you're done.

Before you start

Keep a short notes file (or a piece of paper) open. For each task, jot down the answers to the lettered questions. You are not turning this in — the notes are for you to check that you actually *saw* the things the lesson is about, not just read about them.

The three tasks

Task 1 — Your connection and your router

First, find out what kind of connection you have and poke around the SOHO device that manages it.

1. Open speedtest.net and run a test. When it finishes, note the **ISP name**, the **connection type** if shown (Fiber, Cable, DSL, Cellular), and your **download / upload speeds**.
2. Now open ipinfo.io in a new tab. Write down your **public IP address** and the **org / ISP line**. Does it match what Speedtest reported?

3. Open your **home router's admin page**. The address is almost always `192.168.1.1` or `192.168.0.1` — type it directly into your browser's address bar (not a search engine). Most routers use `admin / admin` or `admin / password` as the default login; check the label on the bottom of the router if that doesn't work.
4. Once inside, explore the settings menus until you find and note down **three SOHO features** mentioned in the lesson. Common examples to look for:
- **DHCP** — usually in a "LAN" or "Local Network" section; shows the range of private IPs being handed out.
 - **NAT** — often visible as "WAN IP" vs your devices' "LAN IPs"; that gap is NAT in action.
 - **Wi-Fi bands** — note whether you see 2.4 GHz, 5 GHz, or 6 GHz listed.
 - **Connected devices** — a list of every device on your network and its private IP.
 - **Firmware version** — note it. Is there an update available?

Self-check — answer all three:

- **(a)** What ISP and connection type does Speedtest report?
- **(b)** What is your public IP, and does the ISP label on ipinfo.io match?
- **(c)** Name three SOHO features you found in the router admin UI (you don't need to change anything, just note what you saw).

✓ You have found where the boundary between “private home network” and “public internet” lives, and you’ve seen that your router is doing more than just “providing Wi-Fi.”

Task 2 — The data-center world from Google Maps

One of the sharpest ways to feel the scale difference between SOHO and data-center networking is to look at the physical footprint.

1. Open [Google Maps](#) and search for "**Google Data Center Council Bluffs Iowa**". Switch to **Satellite view**. Zoom out until you can see the full campus — the grey rectangular buildings with their characteristic cooling equipment on the roof.

2. Use the map's **measure distance** tool (right-click on the map → "Measure distance") to estimate the rough **length and width of the campus** in metres or feet.
3. Compare that to the floor plan of your home or apartment. The data-center campus is, typically, the size of several city blocks.
4. Now open the Wikipedia article: en.wikipedia.org/wiki/Google_data_centers. Read the first two paragraphs only. Note one fact that surprised you.

Self-check — answer all three:

- **(a)** Roughly how large is the Google Council Bluffs campus? (Your best estimate from the satellite view is fine.)
- **(b)** One fact from the Wikipedia article that surprised you.
- **(c)** Can you spot the *cooling towers* on the satellite image? (Hint: they look like large rectangular structures with fan housings on top, often beside the main buildings.) Why do you think a data center needs so much cooling infrastructure?

✓ You have a physical mental image for the “data center” world — it is not a server room in an office basement.

Task 3 — See Tor in action

This task demonstrates what the theory page explains about anonymizers: your apparent IP address changes to an exit-node IP in a country you may not be in.

1. Download **Tor Browser** from torproject.org/download. It is a free, open-source application maintained by a non-profit. No account or sign-up required.
2. Open Tor Browser and wait for it to connect to the Tor network (it shows a progress bar; usually 10–30 seconds). When connected, it opens to the Tor start page.
3. In Tor Browser, navigate to ipinfo.io. Note the **IP address**, the **country**, and the **ISP / org** shown. These are the details of a Tor *exit node* somewhere in the world — not your home IP.
4. Open your normal browser (Chrome, Firefox, Safari, Edge) in a separate window and also navigate to ipinfo.io. Compare the two IP addresses side by

side.

5. Optional: click the **New Circuit for this Site** button in the Tor Browser toolbar (the broom icon or the lock icon in the address bar). Reload ipinfo.io. Notice that the exit-node IP has changed again.

Self-check — answer all three:

- **(a)** What country did your Tor exit node appear to be in?
- **(b)** Does the org / ISP line on the Tor IP look like a commercial VPN provider, a hosting company, or something else? What does that tell you about who runs Tor exit nodes?
- **(c)** Pages loaded noticeably slower in Tor Browser than in your normal browser. In one sentence, why? (Hint: count the hops.)

Uninstall note

Tor Browser is a self-contained application — it does not install anything into your system or modify your other browsers. You can delete it like any other application when you are done.

What “done” looks like

You're done when, in your notes, you have:

- Your ISP name, connection type, public IP, and three SOHO features from Task 1.
- A rough campus-size estimate and one surprising data-center fact from Task 2.
- Your Tor exit-node country, an observation about who runs exit nodes, and a one-sentence explanation of why Tor is slower from Task 3.

The aim was never to make you a network engineer in 45 minutes. It was to make the words “SOHO router,” “data center,” and “anonymizer” refer to things you have actually *seen*, not just read about. The next time someone mentions a “VPN” or asks why a data center needs an entire city block of cooling, you will have a picture.

Where this goes next

Lesson 09 moves from the physical and logical network into security: how attackers move through networks, what firewalls and intrusion detection systems do, and how zero-trust architecture changes the perimeter model. The networking concepts from this lesson — VLANs, VPNs, traffic monitoring — reappear there as the things defenders are protecting and measuring.

Lesson 09: Computing & Virtualization

Lesson 09 — Computing & Virtualization

From the closet under the stairs to AWS us-east-1 — what runs your software, and the trick that made it all possible.

IT Landscape

Module 3

Where does compute live?

A spectrum from the dusty server in the stationery cupboard all the way to a hyperscaler data center the size of several city blocks.

- **On-premises** — hardware you own, sit in your building, managed entirely by your team.
 - **Co-location (colo)** — your hardware, someone else's building, power, and cooling.
 - **Managed hosting** — rented hardware in a provider's data center; they handle physical maintenance.
 - **Public cloud** — AWS, Azure, GCP. You rent capacity by the second; you never touch a physical server.
-

CapEx vs OpEx — the real trade-off

	On-Premises	Public Cloud
How you pay	Buy hardware upfront (CapEx)	Pay-as-you-go monthly bill (OpEx)
Control	Total — every setting, every cable	Limited — you configure, they run the DC
Redundancy	You build it; expensive	Included; multi-AZ is a checkbox
3 AM call	Your team's pager goes off	Cloud provider's team gets the call

What virtualization actually is

One big house chopped into separately-locked apartments. Each tenant has their own space; none of them knows the others exist.

- Before virtualization: one server ran one operating system. If you needed 10 services, you bought 10 servers — typically at **5–15% utilization** each.
 - After virtualization: one beefy server runs many virtual machines. Average utilization jumps to **60–80%**.
 - The magic layer in between is called the **hypervisor**.
-

The hypervisor: landlord of compute

Type 1 — bare metal

Runs *directly on the hardware*. No host OS in between.

VMware ESXi · Microsoft Hyper-V · KVM · Xen

Used in data centers. Maximum performance.

Type 2 — hosted

Runs *inside a host OS* like an app.

VirtualBox · VMware Workstation · Parallels

Used on laptops for testing. Convenient, some overhead.

Rule of thumb: if it runs in a production data center, it's Type 1. If you ran it on your laptop to test something, it's Type 2.

Containers — not VMs, but neighbours

	Virtual Machine	Container (Docker)
What's isolated	Full guest OS + kernel	Process + filesystem only
Shares	Hardware via hypervisor	Host OS kernel
Boot time	Seconds → minutes	Milliseconds
Size on disk	Gigabytes	Megabytes
Typical use	Long-lived server	Microservice, CI job

Both solve “this works on my machine” by bundling the environment alongside the code.

Why virtualization changed everything

- **Utilization** — one physical host replaces a rack of idle servers. Same bill, far more work done.
 - **Isolation** — a runaway process in one VM can't eat the memory of another. Failure is contained.
 - **Mobility** — a VM is just a file. Pause it, copy it to another host, resume. Hardware maintenance stops being an emergency.
 - **Snapshots** — before a risky change, take a snapshot. If it breaks, revert in seconds. This made sysadmins less afraid to experiment.
-

Orchestration — managing thousands at once

*When you have one VM, you log into it. When you have a thousand, you need a conductor. That conductor is an **orchestrator**.*

- **VMware vCenter** — the orchestrator for ESXi hypervisors. Dashboard, migrations, alerts.
 - **OpenStack** — open-source cloud orchestration; what many telcos and universities run to build private clouds.
 - **Kubernetes** — the dominant container orchestrator. Schedules, restarts, scales thousands of containers across a cluster.
 - **Nomad** — lighter-weight alternative to Kubernetes, handles both containers and VMs.
-

Kubernetes in plain English

You describe the *desired state*: "I want three copies of this web server running at all times."

Kubernetes makes it happen and keeps it happening.

- **Pod** — the smallest unit; one or more containers running together.
- **Node** — a worker machine (VM or physical) in the cluster.
- **Control plane** — the brain that schedules pods, watches for failures, and reschedules them elsewhere.
- If a node dies, Kubernetes notices within seconds and re-creates the failed pods on healthy nodes.

You don't say "run this on server 7." You say "run three of these." The scheduler picks the servers.

Watching the herd: metrics, logs, traces

- **Metrics** — numbers sampled over time: CPU at 72%, memory at 4.2 GB, requests per second = 340. *Tool: Prometheus*
- **Logs** — lines of text emitted by running processes: timestamps, errors, warnings. *Tools: Loki, Elasticsearch*
- **Traces** — the path one request took through a system: "this HTTP call touched service A, then B, then the database; step B was slow." *Tool: Jaeger*

Together these three signals are called **observability** — the ability to understand what a system is doing from the outside.

The dashboard layer

Raw metrics mean nothing without a picture. The dashboard layer glues data sources together into charts that humans can act on.

- **Grafana** — open-source, connects to Prometheus and dozens of other sources. The standard self-hosted option.
- **Datadog** — commercial SaaS, one agent install, everything in one place. Common at startups and mid-size companies.
- **New Relic, Dynatrace** — similar commercial offerings, stronger on APM (application performance monitoring).

When an alert fires at 3 AM, the on-call engineer opens a Grafana or Datadog dashboard first — before touching anything.

Who does this work?

Title	Core responsibility	Typical focus
Sysadmin	Keep servers alive	On-prem, legacy systems
SRE	Reliability as an engineering discipline	Large-scale cloud services
Platform Engineer	Build the internal tools devs use to deploy	Kubernetes, CI/CD pipelines
Cloud Architect	Design the infrastructure blueprint	Multi-cloud strategy, cost

Things people get wrong

- **"Cloud is always cheaper."** Not true at scale. Large companies often save millions by repatriating workloads to owned hardware. The break-even is typically 3–5 years of consistent load.
 - **"Containers replace VMs."** They coexist. Kubernetes itself usually runs *on top of* VMs. Containers package apps; VMs provide hardware isolation.
 - **"Kubernetes is for everyone."** Kubernetes is powerful and genuinely complex. A five-person startup often does better with a simpler platform like Railway, Fly.io, or a managed container service.
-

Recap

- Compute lives on a spectrum: closet server → colo → cloud. CapEx vs OpEx is the key trade-off.
- Virtualization is one big house chopped into apartments. The hypervisor is the landlord.
- Type 1 hypervisors run on bare metal (data centers); Type 2 run inside an OS (laptops).
- Containers share the host OS kernel; VMs bring their own. Both isolate "this works on my machine."
- Orchestrators (vCenter, Kubernetes) manage thousands of VMs/containers like a conductor leads an orchestra.
- Observability = metrics + logs + traces. Grafana and Datadog make them visible.

Coming up: Lesson 10 — DevOps & Deployment (CI/CD, pipelines, and how code gets from a laptop to a live server).

Computing & Virtualization

Reading time: ~10 min · Prereqs: Lessons 01–08 · *Snapshot 2026 — hypervisor technology is stable; orchestration tooling, especially Kubernetes, evolves quickly.*

Why this matters to you

Every conversation about “the cloud,” every Jira ticket that says “we need more compute,” every blog post about Kubernetes eventually arrives at the same question: where, physically and logically, does your software actually run? The answer is almost never “on one computer.” It is some combination of rented servers, virtual machines, and lightweight containers managed by software that decides, moment to moment, which hardware to use. If you work anywhere near a product that has users, understanding this layer tells you why deployments take the time they do, why the monthly cloud bill looks the way it does, what a “container” is when a developer mentions it, and what an SRE actually does when they say they’re “on call.” This lesson is the mental model: no configuration, no commands, just the map.

1. The spectrum from cupboard to hyperscaler

Where a company runs its software is a spectrum, not a binary choice. At one extreme: the dusty desktop-class server humming in a locked cupboard, managed by whoever still knows the root password. At the other: a data center the size of several city blocks, owned by Amazon or Google, with its own power substation, fiber entry points, and 24-hour physical security. Most real organizations sit somewhere in between.

On-premises (on-prem) means the hardware is in a building you control — your office, your data-center room, your leased cage in a co-location facility. You bought the servers (capital expenditure, **CapEx**), you rack them, you patch them, and your team gets the pager at 3 AM when a disk fails. The upside is control: every setting, every cable, every security policy is yours. The downside is that the hardware sits idle at 5–15% average utilization on a busy day because you had to buy enough capacity for your peak load.

Public cloud (AWS, Azure, GCP) flips that equation. You pay by the second for exactly as much compute as you use (operating expenditure, **OpEx**). Spin up a thousand servers for a one-hour batch job, pay for one hour, shut them down. No hardware to buy, no data center to cool. The downside: you have no control over the physical layer, costs can surprise you spectacularly if an engineer forgets to turn something off, and at very large sustained loads the bill often exceeds what owned hardware would have cost. Many large companies discovered this in the early 2020s and “repatriated” significant workloads back on-prem after the initial cloud euphoria faded.

Neither answer is universally correct. The industry as of 2026 runs on a **hybrid** mix: regulated data (banking, healthcare) often stays on-prem or in a private cloud for compliance reasons; burst capacity and experimental workloads go to the public cloud; the teams managing both speak the same conceptual language.

2. What virtualization actually is

Until around 2000, the default model was one operating system per physical server. If you needed to run a web server and a database server, you bought two machines. Both sat mostly idle. Then **virtualization** arrived with a deceptively simple idea: what if the hardware could pretend to be multiple independent computers at once?

The analogy that works best is an apartment building. Imagine a huge house with one owner. Now imagine a developer buys it, installs internal walls, separate front doors, and separate meter boxes, and rents each flat independently. Each tenant has their own kitchen, their own bathroom, their own locks. They cannot walk into a neighbour’s flat. From a tenant’s point of view, they live in a complete, self-contained home. They have no idea how many other tenants are in the building, and they do not care. The building owner — the **hypervisor** — is the one who manages the shared structure underneath: the foundations, the electrical wiring, the plumbing.

In computing terms: the physical server is the building. The hypervisor is the landlord software that divides the hardware into isolated slices. Each slice is a **virtual machine** (VM): it has its own virtual CPU, virtual RAM, virtual disk, and its own complete operating system. The guest OS inside the VM has no idea it is sharing a physical host with ten other VMs. From inside, it looks and behaves like a bare-metal server.

Why did this change everything? Four reasons. First, **utilization**: one physical host doing the work of ten idle servers dramatically reduces hardware and electricity costs.

Second, **isolation**: a runaway process in VM 3 cannot eat the memory of VM 7; the hypervisor enforces hard limits. Third, **mobility**: a VM is ultimately just a large file on disk. You can pause a running VM, copy it to another physical host, and resume it — hardware maintenance stops being an emergency that requires a service window. Fourth, **snapshots**: before applying a risky patch, take a snapshot of the VM's current state. If the patch breaks things, revert to the snapshot in seconds. Sysadmins who had previously dreaded change windows started saying yes to upgrades much more readily.

3. Hypervisor types and where containers fit

Not all hypervisors work the same way. **Type 1 hypervisors** run directly on bare metal: the server boots into the hypervisor software, not into a general-purpose operating system. There is no host OS underneath — the hypervisor is the OS. This is the architecture of the big names used in production data centers: **VMware ESXi**, **Microsoft Hyper-V**, **KVM** (the Linux kernel's built-in hypervisor), and **Xen** (which Amazon Web Services used under the hood for a decade). Type 1 has the best performance because there is one fewer software layer between your VM and the hardware.

Type 2 hypervisors run inside a host operating system as a regular application. You install **VirtualBox** on your Windows laptop, click "New Virtual Machine," and a few minutes later you have a Linux VM running in a window. **VMware Workstation** and **Parallels** (popular on macOS for running Windows) work the same way. Type 2 is convenient and perfectly fine for development and testing; the added overhead is noticeable but rarely a problem when you just need to try something quickly.

Containers (popularised by **Docker** starting around 2013) solve an overlapping problem by different means. Instead of virtualizing the hardware and running a full guest OS, containers virtualize only the operating system layer — they share the host OS's kernel and isolate only the process, filesystem, and network. The result is dramatically smaller and faster: a container image is megabytes rather than gigabytes; it starts in milliseconds rather than seconds. The trade-off is weaker isolation — containers share a kernel, so a kernel vulnerability can affect all containers on the host, whereas VMs each have their own kernel and are harder to escape. In practice, containers and VMs coexist: Kubernetes, the dominant container orchestrator, most commonly runs its container workloads on top of virtual machines, not bare metal. The apartment-building analogy extended: VMs are the individually walled flats; containers

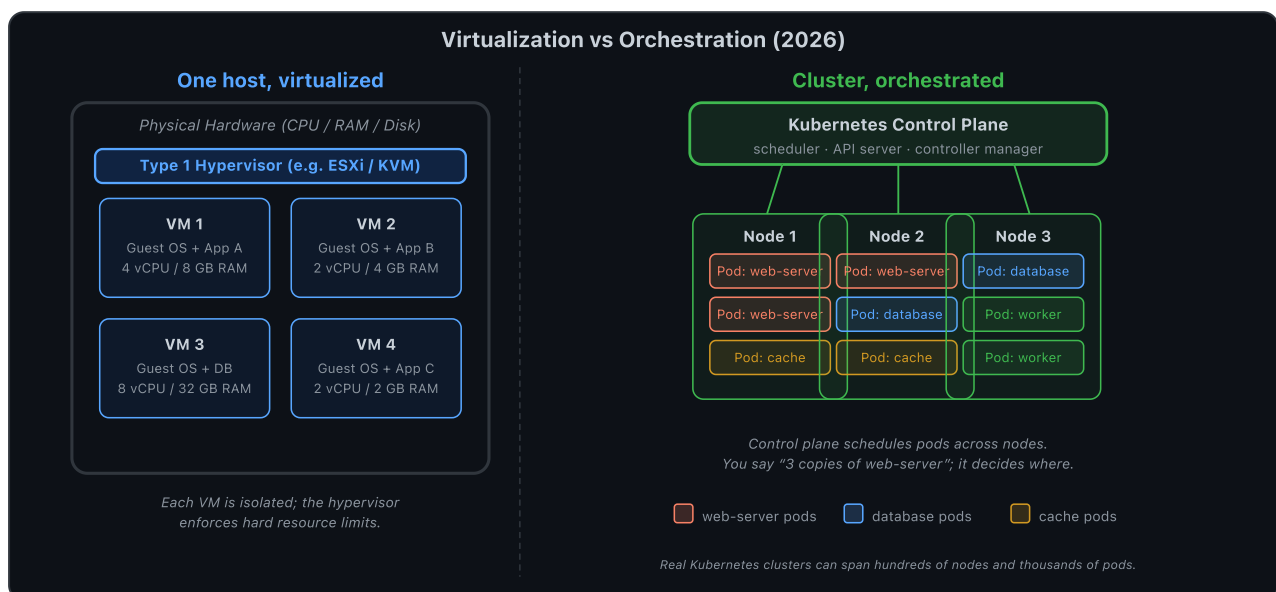
are more like office suites within one flat — shared hallway, but each suite has its own locked door.

4. Orchestration: the conductor for a compute orchestra

When you have one VM, you log into it. When you have ten, you probably have a spreadsheet. When you have a thousand, you need software to track where everything is, decide where to place new workloads, restart anything that dies, and distribute traffic evenly. That software is called an **orchestrator**, and the word is exactly right: an orchestrator is the conductor who tells every musician what to play and when, without touching an instrument itself.

5. The mental model in one diagram

Here is a side-by-side picture of the three layers: the physical host, the hypervisor that divides it, and the orchestrator that manages a cluster of hosts. The left side shows a single host sliced into VMs; the right side shows an orchestrator (Kubernetes) scheduling containers across a pool of nodes.



The left panel shows one physical host divided by a hypervisor into four independent virtual machines, each with its own guest OS. The right panel shows a Kubernetes cluster: a control plane telling three nodes where to run which containers (pods). In production, those nodes are often themselves VMs — so the two models stack, rather than replace each other.

The four major orchestrators each occupy a slightly different niche. **VMware vCenter** manages fleets of ESXi hypervisors and their VMs; it is the dashboard that a traditional enterprise VMware shop uses to see all its virtual infrastructure in one place.

OpenStack is the open-source alternative, popular in telecom operators and universities that want to build a private cloud without paying VMware licensing fees.

Kubernetes (originally from Google, now a CNCF project) became the de-facto standard for running containerized applications and commands a huge ecosystem of supporting tools. **Nomad** (from HashiCorp) is lighter-weight and notably handles both containers and plain executable jobs without the complexity of Kubernetes — often the right answer for organizations that find Kubernetes operationally overwhelming.

5. Watching the herd: monitoring and the people who do it

Running thousands of VMs and containers is only possible if you have real-time visibility into what they are doing. The industry has settled on three complementary signals, which together are called **observability**.

Metrics are numbers sampled at regular intervals: CPU utilization is at 72%, memory consumption is 4.2 GB, the web service is handling 340 requests per second.

Prometheus (open-source) and **Datadog** (commercial SaaS) are the dominant tools for collecting and storing metrics. Metrics are cheap to store and excellent for alerting: “if CPU stays above 90% for 5 minutes, page the on-call engineer.” They are less good for understanding *why* something is slow.

Logs are the text lines that running processes emit: timestamps, error messages, warnings, HTTP request records. They are verbose (a busy service might emit millions of lines per minute) but rich in context. Log aggregation tools like **Elasticsearch** (the “E” in the ELK stack) and **Grafana Loki** ingest these streams and make them searchable. When an alert fires, the engineer opens the logs for the offending service to find the specific error that triggered it.

Traces capture the path one request takes through a distributed system. When a user clicks “Buy” on an e-commerce site, that request might touch a frontend service, an inventory service, a payment service, and a database — all in under 200 milliseconds. A trace records each hop, its duration, and any errors. **Jaeger** and **Zipkin** are the common open-source options; Datadog and New Relic include tracing in their commercial platforms. Without traces, diagnosing a slow request in a microservice

architecture feels like trying to solve a murder mystery with only the last page of the novel.

Grafana deserves a special mention as the most popular open-source dashboard layer: it connects to Prometheus, Loki, and many other data sources, and lets teams build visual dashboards showing all three signal types on one screen. When something breaks at 3 AM, the on-call engineer opens Grafana before touching anything else.

The people who operate this stack carry several job titles. A **sysadmin** (systems administrator) is the original title: keeping servers alive, managing OS updates, user accounts. An **SRE** (Site Reliability Engineer, a title Google coined) treats reliability as a software engineering problem: they write code to automate toil, set **SLOs** (service-level objectives like “99.9% of requests complete in under 200 ms”), and use error budgets to decide when it is safe to ship new features versus when to focus on stability. A **platform engineer** builds the internal tools — the deployment pipelines, the Kubernetes clusters, the secrets management systems — that product developers use every day without thinking about them. All three roles exist along a continuum; at a small company, one person does all three.

What people argue about

- **Cloud repatriation: is the cloud overpriced at scale?** Since roughly 2022, a number of high-profile companies (Basecamp/HEY, Dropbox partially, several financial-services firms) published analyses showing that at consistent, predictable loads, owning hardware is substantially cheaper than renting it. The counter-argument from cloud vendors and their advocates is that the analysis ignores the engineering headcount required to manage on-prem infrastructure, the opportunity cost of capital tied up in hardware, and the value of cloud-native services (managed databases, serverless, AI APIs) that have no easy on-prem equivalent. There is no universal answer; the calculus differs by company size, load profile, and engineering capacity.
- **Kubernetes: essential platform or complexity tax?** Kubernetes is genuinely powerful and has become the industry standard for container orchestration. It is also genuinely complex to operate correctly. A vocal contingent argues that most organizations adopting Kubernetes are not large enough or fast-moving enough to justify the operational overhead, and that simpler platforms (Fly.io, Railway, managed container services) deliver 80% of the benefit at 20% of

the complexity. The pro-Kubernetes camp responds that the ecosystem lock-in risk with simpler platforms is real, and that Kubernetes skills are transferable in a way that proprietary platforms are not.

- **eBPF: the next wave in observability?** A technology called **eBPF** (extended Berkeley Packet Filter) allows code to run safely inside the Linux kernel without modifying it. This has enabled a new generation of observability and security tools (Cilium, Tetragon, Pixie) that can trace exactly what every process is doing at kernel level with minimal overhead. Whether this represents a genuine paradigm shift or “just another observability layer” is a live debate among SREs and platform engineers as of 2026.

Reading on

- **The Phoenix Project** by Gene Kim, Kevin Behr, and George Spafford — a novel about an IT organization in crisis, widely credited with making DevOps and SRE thinking accessible to non-technical readers. The compute and virtualization concepts in this lesson are the backdrop for the story.
- **Google’s Site Reliability Engineering book** — sre.google/sre-book/table-of-contents. Free online. The first five chapters cover the philosophy of SRE and why reliability is an engineering discipline, not a job title. Readable without an engineering background.
- **Kubernetes.io “Concepts” overview** — kubernetes.io/docs/concepts/overview. The official documentation is surprisingly approachable. The “What is Kubernetes?” page gives the mental model in plain prose; the “Kubernetes Components” page maps exactly to the right panel of the diagram above.
- **Corey Quinn’s “Last Week in AWS”** — lastweekinaws.com. A newsletter that covers AWS news with healthy scepticism and dry humor. The back-catalogue is an excellent way to understand what cloud providers actually sell, why their pricing is the way it is, and where the genuine trade-offs lie.

LAST-REVIEWED: 2026-06-02

Homework — Computing & Virtualization

Estimated time: ~60 min · No programming · Two guided tours: one virtual machine, one container

Two tool-tour exercises. The goal is to go from reading about VMs and containers to actually running one of each — even briefly — so the mental model from the theory page gets grounded in something you have touched. Neither task requires you to write any code or understand any command in depth. You are just observing: how much resource it asks for, what the UI looks like, what “running” means in practice.

Before you start

Keep a short notes file open (or grab a piece of paper). For each task, jot down answers to the lettered self-check questions. You are not turning these in — the notes are for you to confirm you actually *did* the things, not just read about them.

Both tasks involve free downloads. Neither installs anything into your system beyond the application itself. Both can be uninstalled cleanly when you are done.

The two tasks

Task 1 — Spin up a virtual machine with VirtualBox

This is the Type 2 hypervisor experience: a full virtual machine running inside a window on your current laptop, exactly as described in the theory. You will create a minimal Linux VM, start it, poke around for a moment, and note what resources it asked for.

macOS alternative (Multipass)

If you are on a Mac with Apple Silicon (M1/M2/M3/M4) and VirtualBox does not work for you, use **Multipass** instead — it is Canonical's free lightweight VM manager that runs natively on Apple Silicon. Download from multipass.run/install. The self-check questions below apply to either tool.

1. Download and install **VirtualBox** from virtualbox.org/wiki/Downloads. Choose the installer for your operating system. The install is straightforward — click through the defaults.
2. You also need a disk image (ISO file) for the guest OS. Download the **Ubuntu Server 24.04 LTS ISO** from ubuntu.com/download/server. It is about 2 GB. While it downloads, continue reading.
3. Open VirtualBox. Click **New** to create a virtual machine. Give it a name (e.g. "test-vm"), set the type to **Linux**, version **Ubuntu (64-bit)**. On the next screen, allocate **2048 MB (2 GB) of RAM** and **1 CPU**. Create a new virtual hard disk at the default size (10 GB is fine; it does not immediately use 10 GB on your real disk — it grows lazily).
4. Before starting the VM, attach the ISO: in VirtualBox, select your new VM, click **Settings → Storage → Empty** (under the optical drive icon) → click the disc icon on the right → "Choose a disk file" → select the Ubuntu ISO you downloaded.
5. Click **Start**. The VM boots into the Ubuntu installer. You do not need to complete the installation — watching the boot sequence and seeing the installer menu is enough for this exercise. Notice:
 - The boot takes several seconds. Compare that to how fast a container starts (Task 2).
 - The VM runs in its own window. Your host OS is completely unaffected by what you do inside it.
 - In VirtualBox's main window, look at the **Details** pane for your VM — it shows the allocated RAM, CPU count, and virtual disk size.
6. Close the VM window and choose "Power Off the Machine" when asked.

Self-check — answer all three:

- **(a)** How much RAM and how many CPUs did you allocate to the VM? Does the VM installer see them as if they were real hardware?

- **(b)** Roughly how long did it take from clicking Start to seeing the Ubuntu installer menu? (Time it.)
- **(c)** In the VirtualBox Details pane, can you see where the virtual hard disk file is stored on your real hard drive? Note the path — this is the “VM is just a file” concept from the theory.

✓ You have run a Type 2 hypervisor, created a VM, and seen hardware virtualization from the inside. The numbers — boot time, RAM, disk — are now something you observed, not just read.

Task 2 — Run a container with Docker Desktop

This task shows the container experience: something that starts in milliseconds, shares the host kernel, and disappears without a trace when you stop it. You will run two containers — the obligatory “hello-world” and an interactive Ubuntu shell — and then look at the Docker Desktop dashboard.

1. Download and install **Docker Desktop** from docker.com/products/docker-desktop. It is free for personal use. The installer sets up Docker’s engine and a graphical dashboard. Restart if prompted.
2. Open a terminal (Terminal on macOS/Linux; PowerShell or Command Prompt on Windows). Type the following command and press Enter:

```
docker run hello-world
```

Docker downloads a tiny image (a few kilobytes) and runs it. It prints a “Hello from Docker!” message explaining what just happened, then exits. The whole thing takes a few seconds — mostly download time on first run, fractions of a second on repeat runs.

3. Now run:

```
docker run -it ubuntu bash
```

This pulls the official Ubuntu container image and drops you into a bash shell *inside the container*. Your prompt will change to something like

```
root@a3b7c9d2e1f0:/#
```

. Try a few commands to look around:

- `ls /` — lists the root filesystem (it looks like a Linux install, but it's a container)
- `cat /etc/os-release` — shows the Ubuntu version inside the container
- `ps aux` — shows running processes. Notice there are very few — just bash and ps itself.

Type `exit` to leave the container shell.

4. Open **Docker Desktop**. Click the **Containers** tab. You will see the containers you just ran listed there, with their status (Exited). Click the **Images** tab — you can see the `ubuntu` and `hello-world` images that were downloaded, along with their sizes.

Self-check — answer all three:

- **(a)** How large is the `ubuntu` container image shown in Docker Desktop? Compare that to the 2 GB Ubuntu Server ISO from Task 1. What is the ratio? Why is the container image so much smaller?
- **(b)** When you ran `ps aux` inside the Ubuntu container, how many processes were listed? Now open your host system's task manager or Activity Monitor and note how many processes your real OS is running. What does that gap tell you about container isolation vs VM isolation?
- **(c)** How long did the second run of `docker run hello-world` take (or how fast did the ubuntu container start, after the image was already downloaded)? Compare that to the VM boot time you measured in Task 1.

✓ You have run two containers, looked inside one, and seen the Docker Desktop dashboard. The theory concepts — shared kernel, lightweight images, millisecond start — are now numbers you measured yourself.

What “done” looks like

You are done when, in your notes, you have:

- VM resource allocation (RAM, CPU), boot time, and the virtual disk file path from Task 1.

- Container image size, process count comparison, and container start time from Task 2.

The point of these two exercises is not to teach you to manage VMs or write Dockerfiles. It is to make the words “virtual machine” and “container” refer to things you have held in your hands, not just read in a diagram. The next time an engineer says “we containerized it” or “the VM was snapshotted before the deploy,” you will have a picture — and a number for how fast each one starts.

Uninstall note

Both VirtualBox and Docker Desktop are self-contained applications. To remove them: on macOS, drag them to the Trash; on Windows, uninstall via Settings → Apps. The virtual machine files (the .vdi or .vmdk disk image) are stored in your home folder under `VirtualBox VMs/` and can be deleted separately if you want to free the disk space.

Where this goes next

Lesson 10 moves to **DevOps & Deployment**: how code travels from a developer’s laptop through a CI/CD pipeline and lands on the compute infrastructure you just explored. The containers you ran in Task 2 are exactly what those pipelines package and ship.

Lesson 10: DevOps & Deployment

Lesson 10 — DevOps & Deployment

How code goes from a laptop to running for real users — CI/CD, containers, IaC.

IT Landscape

Module 3

The journey of one line of code

Type → commit → push → test → build → deploy → users.

- **20 years ago** — this took *weeks* of manual handoffs between teams.
 - **In 2026** — it can take **10 minutes**, end to end, fully automated.
 - The cluster of tools and habits that makes that possible is called **DevOps**.
-

What “DevOps” actually means

- Less a tool, more a **culture**.
- *“The people who write the code also own running it in production.”*
- Replaced the old **“developers throw code over a wall to ops”** model.
- The **toolchain** we’ll tour today is what grew around that culture.

SRE (Site Reliability Engineering, coined by Google) is the sibling discipline — for this lesson, treat them as the same job family.

Tool 1 — Version control as the trigger

- Every change starts with a `git push` (see Lesson 05).
- The remote — **GitHub, GitLab, Bitbucket** — **fires events** on every push and every pull request.
- Those events are what **kicks off everything else** in the pipeline.

No commit, no pipeline. The git history is the source of truth.

Tool 2 — CI (Continuous Integration)

Every change is automatically built and tested before it merges.

- On every push, an automated pipeline runs your **test suite**, lints the code, builds the artifacts.
- If anything fails, the change is **blocked**.

Popular CI hosts (as of 2026):

- **GitHub Actions** — dominant.
 - **GitLab CI**, **CircleCI**, **Buildkite**.
 - **Jenkins** — the old guard, still huge in enterprise.
-

What a CI run looks like

- A **green check** or a **red X** next to every pull request.
- Click in → see the log of every step: clone repo, install dependencies, compile, run tests.
- Each step shows its own duration; the whole run usually finishes in **5–20 minutes**.

The green check is the modern code review's best friend.

A real CI config (illustrative)

A minimal **GitHub Actions** workflow, lives at `.github/workflows/ci.yml`:

```
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
      - run: npm test
```

You don't need to run this — just notice it's plain text in your repo, versioned alongside the code.

Tool 3 — CD (Continuous Deployment / Delivery)

- Same machinery, **one step further**: after tests pass, automatically **deploy**.
- **Continuous Delivery** — every successful build is *ready* to deploy (human clicks the button).
- **Continuous Deployment** — every successful build *is* deployed, automatically.

The most mature teams (Amazon, GitHub, Shopify) deploy **thousands of times a day** — fully automated, with zero human pressing a button.

Tool 4 — Containers (Docker)

A container is your app plus everything it needs, sealed in a box that runs the same anywhere.

- Solved the **“works on my machine”** problem once and for all.
 - **Docker** is the brand most people learn first.
 - A container image is a frozen **filesystem + config + run-command**.
 - Often **<100 MB**, starts in **<1 second**, identical across laptop / CI runner / production server.
-

Tool 5 — Orchestrators (Kubernetes & friends, as of 2026)

- When you have **100 containers across 50 machines**, you need something to schedule them, restart them when they crash, and route traffic.
- **Kubernetes** (a.k.a. `k8s`) is the heavyweight standard.
- Lighter alternatives: **Nomad**, **AWS ECS**, **Fly.io**, **Render**, **Railway**, **Cloud Run** (Google).

Most teams in 2026 do NOT run Kubernetes themselves — they pay a cloud to run it for them (EKS / GKE / AKS).

Tool 6 — IaC (Infrastructure as Code)

Your servers, networks, and databases are declared in text files — not clicked into existence.

- **Terraform** (HashiCorp) / **OpenTofu** (the open fork) — cross-cloud standard.
- **Pulumi** — same idea, written in Python / TypeScript.
- **AWS CloudFormation** — the AWS-native one.
- **Ansible** — for configuring existing servers.

Your cloud setup becomes version-controlled, reviewed, and reproducible.

Tool 7 — Monitoring & observability

Once it's running, you need to know if it's **healthy**. Three telemetry families:

- **Logs** — lines of text. *Datadog, Loki, CloudWatch, Splunk.*
- **Metrics** — numbers over time. *Prometheus + Grafana, Datadog.*
- **Traces** — one request's path across services. *Jaeger, Honeycomb, Datadog APM.*

If a tree falls in production and nobody's monitoring — your users tell you on Twitter.

Recap

- The **pipeline**: commit → CI → container → orchestrator → monitoring.
- The **culture**: whoever writes it, owns it.
- The **goal**: deploy small changes, often, safely.

Coming up: Lesson 11 — keeping all of that from getting hacked.

DevOps & Deployment

Reading time: ~10 min · Prereqs: Lessons 01–07 · *Snapshot 2026 — the culture is stable; CI/CD and orchestrator brand-names move every couple of years.*

Why this matters to you

When you open the Spotify app and a fresh playlist appears, when your bank's mobile app gets a new "Send to a friend" button, when Notion ships a new slash command — that change started as a few lines of code on an engineer's laptop and arrived on your phone, probably the same day. Twenty years ago this took weeks of manual handoffs between developers, QA, and operations teams. Today the whole journey is mostly automated. The cluster of tools and habits that automate it is called **DevOps**. Knowing the moving parts makes you a better product manager, recruiter, executive, or curious bystander — you can read a deployment diagram, ask the right questions in a planning meeting, and stop being mystified when someone says "we'll ship it after the pipeline goes green."

1. From "throw it over the wall" to "you build it, you run it"

Old-school IT split labor: developers wrote code, then handed a zipped folder to a separate **operations** team to install on a server. The two teams used different tools, had different incentives ("ship fast" vs "don't get paged at 3 AM"), and blamed each other when things broke. **DevOps**, the cultural movement that emerged in the late 2000s, dissolved that wall: the same people who write the feature also write the deployment scripts, get the pager when it breaks, and own the customer outcome. The tooling chapter of DevOps — CI, containers, IaC, monitoring — is what grew around that culture to make ownership practical at scale. **SRE** (Site Reliability Engineering, a Google-coined sibling discipline) overlaps almost entirely; treat the two as the same job family for the purposes of this lesson. The shorthand "you build it, you run it" was popularized by Werner Vogels at Amazon and is now industry default *as of 2026*.

2. CI — Continuous Integration

Every time a developer pushes a change to a shared git repo, a **CI pipeline** automatically runs: it pulls a fresh copy of the code, installs dependencies, compiles, runs the test suite, runs linters, and builds artifacts. If anything fails, the change is blocked from merging. As of 2026 the popular CI hosts are **GitHub Actions** (now dominant, especially in open source), **GitLab CI**, **CircleCI**, **Buildkite**, and **Jenkins** (the open-source granddaddy, still entrenched in many large enterprises). All work roughly the same way: you commit a small YAML file to your repo that declares "on every push, run these steps in this order on this kind of machine". The result you see is a green check or a red X next to every pull request — the modern code reviewer's first signal. CI also pays a quieter dividend: every test that exists in your repo is guaranteed to run on every change, which catches the regressions humans forget about.

3. CD — Continuous Deployment / Continuous Delivery

Once tests pass, the natural next step is to **deploy** the new code. **Continuous Delivery** means every successful build is *ready* to deploy (a human still clicks the button). **Continuous Deployment** means every successful build *is* deployed, automatically, no human in the loop. The most mature teams (Amazon, GitHub, Shopify) deploy thousands of times a day. The trade-off: full continuous deployment requires excellent tests and excellent monitoring — because there's no human gate, the system itself has to catch regressions. Most teams as of 2026 sit on a spectrum somewhere between manual once-a-week deploys and full continuous deployment, often using **feature flags** (LaunchDarkly, Unleash, or homegrown) so the deploy and the launch can be decoupled.

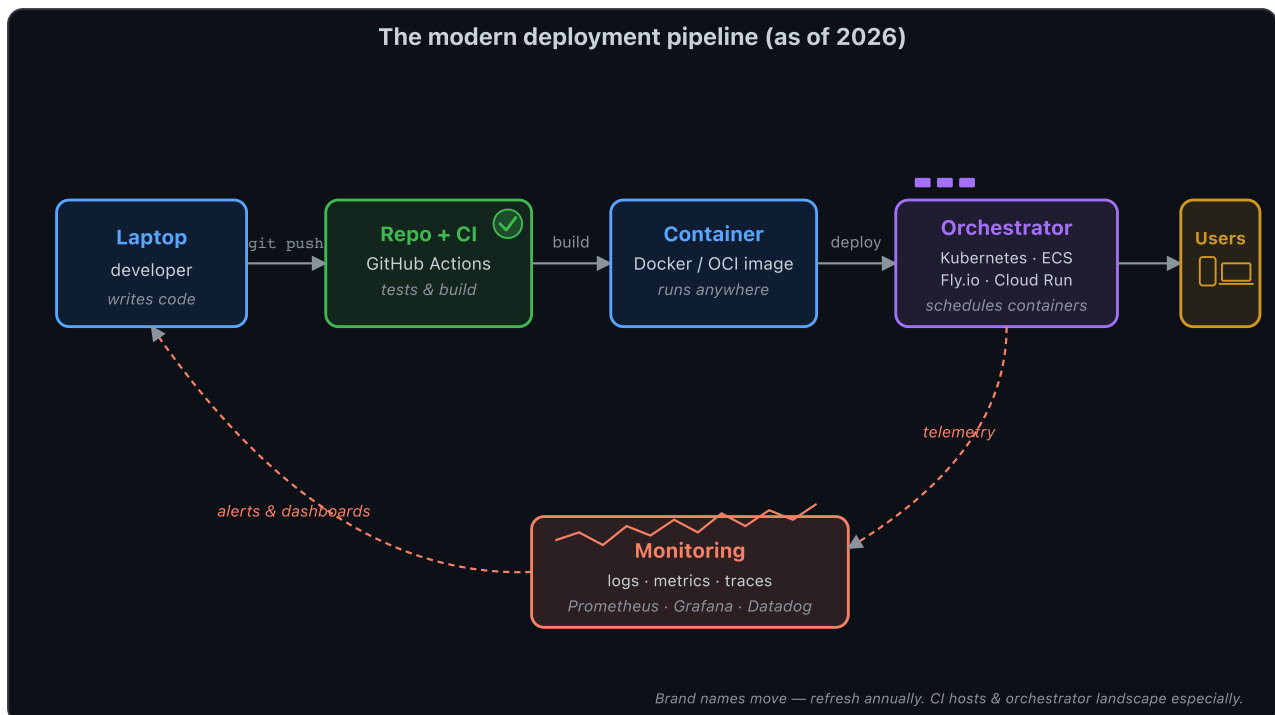
4. Containers — the "works on my machine" cure

A **container** is your application plus *everything it needs to run* (the OS libraries, the language runtime, the dependencies, the config) packaged into a sealed image. The image runs identically on your laptop, on the CI runner, on a colleague's machine, and on the production server. **Docker** is the brand that mainstreamed this approach in 2013 and is still where most developers first learn it. The image format itself is now an open standard (**OCI**). Containers are *not* virtual machines — they share the host's kernel and

start in well under a second, where a VM takes a minute. The shift to containers (~2014–2020) was the single biggest change in how software gets shipped this century — and it's what made the next tool possible.

5. The modern deployment pipeline, one diagram

Here is the full journey of a change from a developer's laptop to a user's phone, with a monitoring loop back to the developer. When you read a deployment diagram in the wild, you can map every box on it onto one of these stages.



6. Orchestrators — running many containers

One container is easy; a hundred containers across fifty machines is not. An **orchestrator** schedules containers onto machines, restarts them when they crash, routes incoming traffic to healthy ones, scales them up or down with demand, and replaces them during deploys without dropping requests. **Kubernetes** (also known as **k8s**, originally from Google) is the heavyweight industry standard — powerful and famously complex. Most teams *as of 2026* do NOT run their own Kubernetes; they pay a cloud to run it for them: **EKS** (AWS), **GKE** (Google), **AKS** (Azure). Lighter, opinionated alternatives have grown popular for smaller teams: **AWS ECS / Fargate**, **HashiCorp Nomad**, **Fly.io**, **Render**, **Railway**, and **Google Cloud Run**. (This space is on [refresh-checklist.md](#) as medium-churn — the vendor list will shift between annual refreshes.)

7. Infrastructure as Code

IaC means you declare your servers, networks, databases, DNS records, and other cloud resources as text files — versioned in git alongside your application code — rather than clicking around a cloud console. **Terraform** (HashiCorp; with the **OpenTofu** open-source fork after the 2023 license change) is the multi-cloud standard *as of 2026*.

Pulumi lets you write the same kind of declarations in Python or TypeScript. **AWS CloudFormation** is the AWS-native version. **Ansible** is more about *configuring* existing servers than provisioning new ones. The upshot: your infrastructure becomes reviewable, reproducible, and disaster-recoverable from a git repo — the same workflow you already use for code.

8. Monitoring & observability

Once your code is running, you need to know if it's healthy. Three telemetry families: **logs** (lines of text — Datadog Logs, AWS CloudWatch, Splunk, Loki); **metrics** (numbers over time — Prometheus + Grafana, Datadog Metrics); **traces** (a single user request's path across services — Jaeger, Honeycomb, Datadog APM). The whole field has rebranded as **observability**; the unifying open standard is **OpenTelemetry**. If a tree falls in production and nobody's monitoring, your users tell you on Twitter — and the post-mortem starts with "why didn't our alerts fire first?"

What people argue about

- **Kubernetes everywhere vs Kubernetes nowhere.** Large enterprises love it; many small-to-mid teams now say "we don't need it" and run on Fly.io, Render, or Cloud Run instead. The pendulum is currently swinging away from default-k8s for teams under ~50 engineers *as of 2026*.
- **Self-hosted CI (Jenkins) vs SaaS CI (GitHub Actions).** SaaS won mindshare, but self-hosted is making a comeback in regulated industries and at very large scale — cost, sovereignty, and custom hardware are the usual drivers.
- **Multi-cloud as a goal.** Every CTO claims to want it; almost nobody achieves it. The practical reality for most teams is "pick one cloud, get really good at it"

— multi-cloud only earns its weight at very large scale or under explicit regulatory pressure.

Reading on

- **The Phoenix Project** by Gene Kim — the classic novelization of how DevOps culture arose. Easier than it sounds; a good first read for non-engineers.
- **Accelerate** by Forsgren, Humble, and Kim — the data behind why fast, automated deployment correlates with business performance. Short and quantitative.
- **The DORA report** — dora.dev. Annual, free; the standard industry benchmark on deployment frequency, lead time, change-failure rate, and recovery time.
- **Lesson 03 of this course** — returns to the cloud platforms where most of this machinery actually runs.

LAST-REVIEWED: 2026-05-25

Homework — DevOps & Deployment

Estimated time: ~45 min · No code to run · A browser, a GitHub account, and a notebook

Three tool-tour exercises. The goal is to see the moving parts of a modern deployment pipeline with your own eyes — a real CI workflow running on a real project, a real container image in a public registry, and a real Infrastructure-as-Code file. By the end you'll have watched an automated build, browsed a Docker image's tags, and read a Terraform file in the wild — without installing anything.

Before you start

Keep a one-page notebook open (paper or a text file). For each task, jot the short answers under (a)/(b)/(c). You are not turning this in — the point is to build the muscle of *poking at* a real deployment pipeline, not just reading about one. None of these tasks require writing code or running commands.

The three tasks

Task 1 — Watch a real CI pipeline run

Open github.com. Pick any well-known open-source project — good candidates are [facebook/react](https://github.com/facebook/react), [microsoft/vscode](https://github.com/microsoft/vscode), or [vercel/next.js](https://github.com/vercel/next.js). Click the **Actions** tab at the top of the repo. (If you'd rather see CI in the context of a code change, open the **Pull requests** tab, click any open PR, and scroll to the bottom — the same workflow runs appear inline under each PR.)

Find a recent workflow run — ideally a green one that finished in the last day or two. Click into it. You'll see a list of **jobs** (often anywhere from 5 to 50+ depending on the project). Click into one job. Watch the *steps* — each one is a

small command (checkout repo, install deps, compile, run tests, upload artifact) with its own log and its own duration.

Self-check — answer all three:

- **(a)** How many jobs ran in this workflow?
- **(b)** How long did the whole workflow take from *queued* to *completed*?
- **(c)** Take a screenshot of one expanded job showing its step-by-step log.

✓ You do NOT need to understand the test output — the goal is to see *the shape of an automated pipeline running on every change*.

Task 2 — Tour a Docker image (no install required)

Open hub.docker.com — Docker's public image registry. Search for [nginx](#) and open the official **nginx** image page (the one with the "Docker Official Image" badge). Read the **Overview** tab — just skim the intro and the "How to use this image" section, don't try to follow it.

Now click the **Tags** tab. Notice that the same image has many different versions side by side: [1.27-alpine](#), [1.27-bookworm](#), [1.27-perl](#), [latest](#), and so on. Each one is a different combination of nginx version + base operating system + extras. Click into one tag and look at its *compressed size* — the [alpine](#) variants are often under 30 MB while the full Debian-based ones can be several times larger.

Self-check — answer all three:

- **(a)** Roughly how many times has this image been pulled in total? (Docker Hub shows the lifetime pull count on the image header — the number is staggering, often in the billions.)
- **(b)** Roughly how many distinct tags / versions are listed?
- **(c)** Why does one logical "image" have versions like [1.27-alpine](#) vs [1.27-bookworm](#)? (Hint: different base operating systems — different size / security trade-offs.)

✓ You do NOT need Docker installed — this is a museum tour of an image registry.

Task 3 — Read an Infrastructure-as-Code file in the wild

Open `github.com` and search for any real-world Terraform file. Two reliable shortcuts:

- In the GitHub search bar, type `language:hcl filename:main.tf` and pick any result that opens a substantial `main.tf`.
- Or browse `github.com/terraform-aws-modules` — open any module repo (e.g. `terraform-aws-vpc`) and click its `main.tf`.

Read 20–40 lines. Don't worry about syntax — just look for the shape: a `provider` block at the top (`aws` / `google` / `azurerm`) followed by a series of `resource "<type>" "<name>" { ... }` blocks. Each resource block is one piece of cloud infrastructure declared in text instead of clicked into existence.

Self-check — answer all three:

- **(a)** Which cloud is this file targeting? (Look for `provider "aws"`, `provider "google"`, or `provider "azurerm"`.)
- **(b)** Name two resources being declared (e.g. `aws_instance`, `aws_s3_bucket`, `aws_vpc`).
- **(c)** Why is declaring infrastructure this way preferable to clicking the same setup into the cloud console? (One sentence — hint: review, version control, reproducibility, from theory §7.)

✓ Take a screenshot of one declared `resource` block for your notes.

What "done" looks like

You're done when, in your notebook, you have:

- two numbers and one screenshot from Task 1 (CI pipeline)

- a pull-count number, a tag count, and a one-sentence answer from Task 2 (Docker Hub)
- a cloud name, two resource types, a one-sentence rationale, and one screenshot from Task 3 (Terraform)

The aim was never to make you a DevOps engineer in 45 minutes. It was to make sure that the next time you hear *"CI is red"* or *"we'll containerize it"* or *"that's all in Terraform"*, you have a concrete picture in your head — because you've now watched a pipeline run, browsed an image registry, and read a real IaC file.

Where this goes next

Lesson 11 closes Module 3 by zooming in on the other half of "runs and stays safe" — the security side. We'll look at the shape of modern attacks (phishing, supply chain, ransomware) and the layered defenses (identity, patching, backups, monitoring) that protect everything you've toured here.

Lesson 11: Security Fundamentals

Lesson 11 — Security Fundamentals

Attacks, network-layer defenses, compute security, and the defense-in-depth model.

IT Landscape

Module 4

Why security at all?

Every system you've learned about in this course can be turned against its users.

- **Data theft** — emails, addresses, passwords, payment info.
- **Ransomware** — your files encrypted, payment demanded.
- **Account takeover** — someone logs in as you.
- **Fraud** — fake invoices, wire transfers, refund scams.
- **Espionage** — state actors and corporate spies.

Global cost of cybercrime *as of 2026*: measured in **trillions of dollars annually** (per Cybersecurity Ventures' running estimate).

Three intuitions to take away

- **1. Attackers go for the cheapest path** — not the most technically impressive one.
- **2. Humans, not algorithms, are the most-attacked surface** — phishing beats zero-days.
- **3. Security is layered** — no single fix is enough. *Defense in depth* wins.

If you remember nothing else, remember these three.

Attack 1 — Phishing (the cheapest path)

Trick a human into giving you credentials or running your code.

Still the **#1 initial-access method** in 2026 across every major incident report (Verizon DBIR, Mandiant M-Trends).

What a phishing email looks like:

```
From: support@arnazon.com    <-- look closely: arn, not ama
Subject: Your account will be locked in 24 hours
Click here to verify: http://arnazon-security.co/verify-now
```

*You've seen one of these. Probably this week. **Spear-phishing** (targeted) is now AI-generated and indistinguishable from a colleague's tone.*

Attack 2 — Credential stuffing & password reuse

- Attackers buy databases of **leaked username/password pairs** — billions exist.
- They try those pairs against **every service** — banks, email, SaaS.
- Because most people **reuse passwords**, a 1% hit rate is enormous.

Mitigation: unique passwords per site (use a **password manager**) plus **MFA** (multi-factor auth) everywhere.

Attack 3 — Supply-chain attacks

Why attack you, when I can attack the library you depend on?

Famous cases:

- **SolarWinds** (2020) — trojaned update reached ~18,000 organizations.
- **Log4Shell** (2021) — one logging library, internet-wide panic.
- **xz-utils backdoor** (2024) — caught hours from landing in every Linux distro.
- Various **npm** / **PyPI** package compromises ongoing in 2026.

A modern app pulls in hundreds of dependencies; any one of them is a foothold.

Attack 4 — Ransomware

Encrypt the victim's data, demand payment for the key.

- As of 2026 still one of the most **disruptive** attack categories — average ransom and average downtime cost both up year-on-year (per IBM X-Force and the annual Sophos *State of Ransomware* report).
 - High-impact targets: **hospitals, municipalities, manufacturing, schools.**
 - Modern ransomware is usually **double-extortion**: encrypt AND leak the data publicly if unpaid.
-

Attack 5 — Web app attacks (the OWASP Top 10)

The **OWASP Top 10** is the long-running canonical list of the most common web-application vulnerabilities.

Selected entries (2021 edition, current heading into 2026's refresh):

- **Broken access control** — #1 for years now.
- **Cryptographic failures** — weak or misused encryption.
- **Injection** (incl. **SQL injection**) — ranked highly for 25 years.
- **Vulnerable & outdated components** — unpatched libraries.
- **Security logging & monitoring failures** — can't see what you're not logging.

You don't need to memorize — you do need to know this list exists.

Defense 1 — Identity & access (the new perimeter)

- **MFA** (multi-factor auth) on everything important — password + a second factor.
- **SSO** (single sign-on) — reduces password proliferation.
- **Least privilege** — give each account only what it strictly needs.
- **Zero trust** — don't trust by network location; verify every request by identity.

Identity is the modern firewall.

Defense 2 — Patching & updates

- Most successful breaches exploit vulnerabilities **with patches already available**.
- **"Patch Tuesday"** (Microsoft's monthly cadence) and the **CVE** registry are how vulnerabilities get tracked publicly.
- Severity is graded by the **CVSS** score (0–10).
- **CISA's KEV** catalog lists vulnerabilities known to be actively exploited — patch those first.

Auto-update where possible; prioritize the rest by severity and known-exploited status.

Defense 3 — Backups, encryption, monitoring

- **Backups** — offline, immutable, *tested*. Most ransomware victims discover their backups didn't work **during** the incident.
- **Encryption at rest and in transit** — HTTPS, TLS 1.3, disk encryption (BitLocker, FileVault, LUKS).
- **Monitoring / SIEM** — detect intrusions in hours, not months.

*Median attacker **dwell time** in 2026 is still measured in **weeks** — per Mandiant M-Trends.*

Defense at the network layer — the first line

- **Firewalls** — perimeter (at the network edge) and host-based (on the machine itself). Both matter.
- **Network segmentation** — split the network into zones (DMZ, internal segments) so a breach in one zone can't freely reach another.
- **NAC** (Network Access Control) — devices must meet health checks before joining the network.
- **IDS / IPS** (Intrusion Detection / Prevention) — watch traffic patterns for known-bad behavior; IDS alerts, IPS blocks in real time. Open-source examples: Snort, Suricata.
- **VPN patch discipline** — unpatched remote-access VPNs are a top ransomware entry point every year.
- **Encrypted DNS (DoH / DoT)** — prevents eavesdropping on and forging DNS lookups.

Network controls don't replace identity — they're the outer ring on the defense onion.

Security in computing & virtualization — shared compute is shared risk

- **Hypervisor isolation** — VMs are meant to be isolated from each other; real cracks existed (VENOM 2015, L1TF/Foreshadow 2018). Hypervisor CVEs are treated as emergencies.
- **Container escapes** — containers share the host kernel; a kernel bug or misconfiguration can let a container break out to the host. Good hardening (seccomp, non-root, SELinux) makes this very hard in practice.
- **Secrets management** — never commit API keys, passwords, or certificates to version control. Bots scan public repos in minutes. Store secrets in Vault / AWS Secrets Manager / Azure Key Vault; inject at runtime.
- **RBAC at the orchestrator** — in Kubernetes, control who can `kubectl exec` into pods, who can read secrets, who can deploy workloads. Wildcard permissions are a common audit finding.

Most compute-layer breaches in 2026 are secrets leaks and misconfigured RBAC — not spectacular VM escapes.

Who does security in a company?

- **Security engineers** — build defenses (IAM, hardening, scanners).
- **SOC analysts** — Security Operations Center; watch the alerts in real time.
- **Incident responders** — handle a breach end-to-end.
- **Penetration testers / red teams** — paid to attack you, before real attackers do.
- **CISO** — Chief Information Security Officer; owns the program.

*And in 2026: **every developer** has security responsibilities — **DevSecOps**.*

Recap

- Attackers take the **cheapest path** — usually a human.
- Humans are the **most-attacked surface** — phishing still wins.
- Defense is **layered**: identity, **network controls** (firewalls, IDS/IPS, segmentation), **compute hygiene** (secrets management, RBAC), patching, encryption, monitoring.
- The **network is the first line of defense** — and shared compute means shared risk.

Coming up: Module 4 — Lesson 12, the AI era begins.

Security Fundamentals

Reading time: ~13 min · Prereqs: Lessons 01–10 · *Snapshot 2026 — concepts are stable; specific breach examples and ransomware stats age fast.*

Why this matters to you

Every system you've learned about in this course — websites, cloud accounts, codebases, databases, deployment pipelines, networks, and virtualized compute environments — can be turned against its users. Cybercrime *as of 2026* is among the largest illegal industries on the planet, measured in the trillions of dollars annually (per Cybersecurity Ventures' running estimate). Even if you'll never write security code, you'll touch security daily: the password manager in your browser, the MFA prompt on your bank app, the breach-notification email that arrives every few months, the suspicious-looking text from "your bank".

This lesson gives you the full shape — broader than "Security Basics." Lessons 08 and 09 built the network and compute layers you now rely on. Security isn't something you bolt on afterward; it runs through those layers from the start. We'll cover how attacks happen, then how the network itself becomes the first line of defense, then how shared virtualized compute introduces its own set of risks. By the end you'll know which questions to ask at every layer.

1. Three intuitions to anchor everything

Three ideas explain most of modern security. First: **attackers take the cheapest path**. They are economically rational. If phishing a human takes ten minutes and exploiting a zero-day vulnerability takes ten months, phishing wins — every time. Second: **humans are the most-attacked surface**. Every major incident report — Verizon's **Data Breach Investigations Report** (DBIR), Mandiant's **M-Trends**, IBM's **X-Force Threat Intelligence** — has put the "human element" in the majority of breaches every year for over a decade (*as of 2026*). Third: **security is layered**. No single fix is enough. The industry phrase is *defense in depth*: assume any one layer will fail, and stack enough layers that the attacker gives up before reaching the prize.

2. The shape of attacks — five archetypes

Phishing — trick a human into giving up credentials or running your code. Still the dominant initial-access vector *as of 2026* (Verizon DBIR). **Spear-phishing** is the targeted version: an attacker studies one specific person and crafts a message tailored to them. AI-generated spear-phishing is now indistinguishable from a colleague's tone, and deepfake voice calls ("vishing") have moved from novelty to routine.

Credential stuffing — attackers buy databases of leaked username/password pairs (billions of them are in circulation), then try each pair against every service. Because most people reuse passwords, a 1% hit rate is enormous.

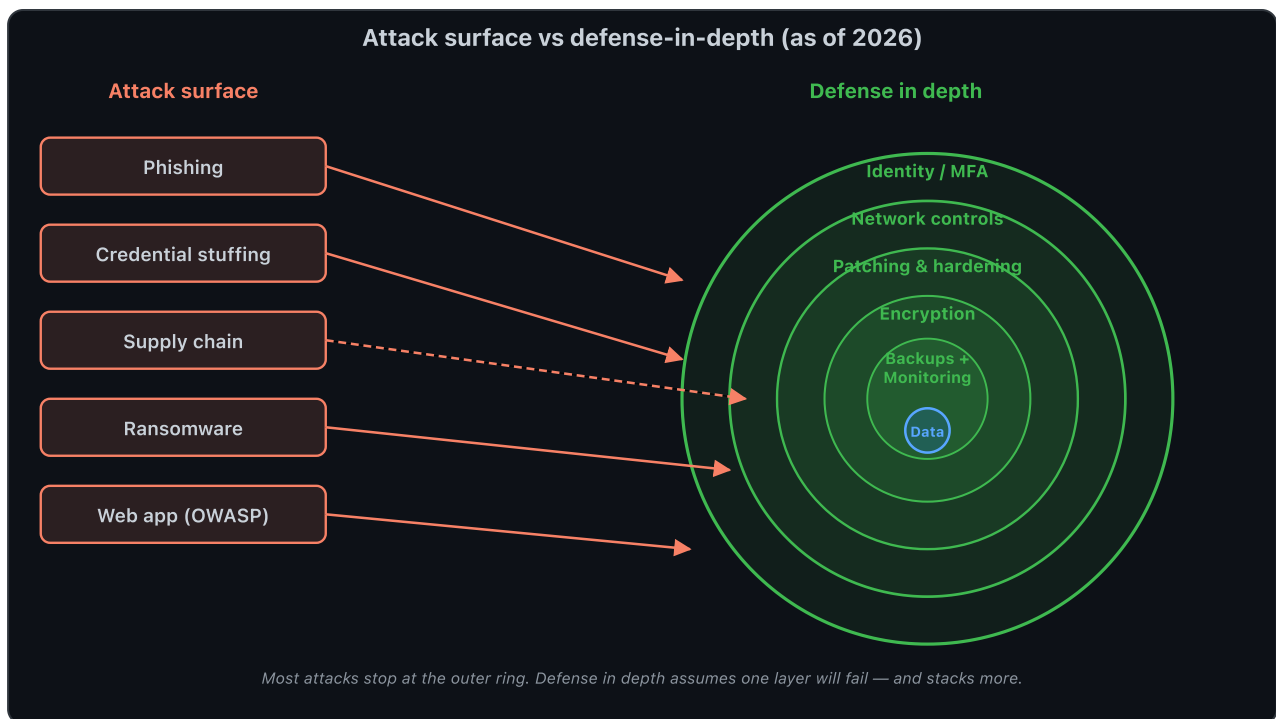
Supply-chain attacks — instead of attacking you, attack a library you depend on. Famous cases: **SolarWinds** (2020), **Log4Shell** (2021), the **xz-utils** backdoor (caught in 2024, cited by Mandiant as the canonical near-miss), and ongoing npm and PyPI compromises. A modern app pulls in hundreds of indirect dependencies; any one of them is a foothold.

Ransomware — encrypt the victim's data, demand payment for the key. *As of 2026* it remains one of the highest-impact attack categories per the annual Sophos **State of Ransomware** and IBM X-Force reports. Hospitals, municipalities, and manufacturers are recurring targets. Modern ransomware is usually **double-extortion**: encrypt AND threaten to leak the data publicly if unpaid.

Web application attacks — the **OWASP Top 10** has been the canonical industry list since 2003, refreshed every few years. As of the 2021 edition (still current heading into the 2026 refresh window), the top three are *broken access control*, *cryptographic failures*, and *injection* (including SQL injection — the same vulnerability that has been ranked highly for 25 years).

3. The attack surface vs the defense onion

Here is the same idea drawn as a diagram. On the left, the five attack archetypes. On the right, the layered defenses an attacker must traverse to reach the protected asset in the middle. Most attackers stop at the first locked layer; the rare persistent ones test every layer in turn — which is exactly what "defense in depth" is designed for.



2026 threat-landscape advisory

The actively hot threat categories as of 2026, per the most-recent annual incident reports: **AI-augmented phishing & vishing** (deepfake voice calls impersonating executives); **ransomware against critical infrastructure** (hospitals, water utilities, ports); **long-tail supply-chain compromises** (small upstream packages with huge transitive blast radius); and **identity-provider attacks** (single-sign-on tenants are a high-value target). Re-check the most recent Verizon DBIR and Mandiant M-Trends for next year's hot list — this paragraph ages fastest.

4. Security in the network layer — the first line of defense

Before an attacker touches your servers, they have to get through the network. This is where Lesson 08's networking infrastructure becomes a security story. The classic tool is the **firewall**: a device (or software) that looks at every packet and decides whether to let it pass based on a rulebook. A **perimeter firewall** sits at the edge of your organization's network and filters traffic coming from the internet. A **host-based firewall** runs on the individual machine itself — so even if an attacker slips past the perimeter, each server still has its own gatekeeper.

Network segmentation is the architectural principle of cutting a network into isolated zones — so that someone who compromises a printer in the guest Wi-Fi zone can't simply walk across to the payroll database. You've seen this in Lesson 08 as VLANs and subnets; the security framing calls it a *DMZ* (demilitarized zone) for public-facing services, and an *internal segment* for sensitive data that never needs to be reached from the internet at all.

NAC — Network Access Control is the policy that says "you can only join this network if your device meets a set of conditions" (up-to-date antivirus, OS patch level, valid certificate). It's the reason a hospital or bank will reject your personal laptop from joining their corporate Wi-Fi even if you know the password.

IDS and IPS — Intrusion Detection and Intrusion Prevention Systems — sit on the network watching traffic patterns, not just headers. An IDS raises an alert when it sees something suspicious (a port scan, an unusual flood of DNS queries, a known malware signature). An IPS goes further and blocks the traffic in real time. Open-source examples include **Snort** and **Suricata**; commercial equivalents are built into most enterprise firewall products.

VPN security matters because remote-access VPNs are now the front door to corporate networks. A VPN that isn't patched, uses weak authentication, or has a publicly disclosed vulnerability is essentially a locked door with the key taped to the outside — a major ransomware entry point every year. **Encrypted DNS (DoH/DoT)** — DNS-over-HTTPS and DNS-over-TLS — closes a different gap: ordinary DNS lookups travel in plain text and can be intercepted or forged to redirect you to a fake site. Encrypted DNS tunnels those lookups so eavesdroppers can't see what domains you're resolving, and so the responses can't be silently altered.

"The perimeter is dead" — and what comes after

A much-repeated observation in security circles is that "the perimeter is dead" — meaning the old castle-and-moat model of one big firewall around a trusted network no longer works when employees work from anywhere and services live in the cloud. This is true as a critique of naive perimeter thinking. But it doesn't mean firewalls and segmentation are useless — it means they aren't *sufficient on their own*. The replacement model is **zero trust**: assume every request is untrusted regardless of where it comes from, and verify by identity and device

posture every time. Zero trust and network controls coexist; one doesn't replace the other.

5. Security in computing and virtualization — shared compute is shared risk

Lesson 09 showed how multiple workloads share physical hardware through hypervisors and containers. That sharing is efficient — and it introduces a category of security questions that don't exist when every workload runs on its own dedicated machine.

Hypervisor isolation is the core guarantee of virtualization: your virtual machine should not be able to see or affect your neighbor's. Most of the time, this holds well. But the guarantee has had real cracks. **VENOM** (2015) was a vulnerability in a widely-used virtual floppy-disk driver that let an attacker escape their virtual machine and run code on the hypervisor host itself. **L1TF / Foreshadow** (2018) was a CPU speculative-execution flaw (in the same family as Spectre and Meltdown) that let a process read memory from another VM sharing the same physical core. These aren't reasons to distrust virtualization — they're why hypervisor vendors treat every CVE as an emergency patch. The lesson: isolation guarantees come from software running on hardware, and both layers can have bugs.

Container escapes are the container equivalent of a VM breakout. Containers share the host OS kernel (unlike VMs, which have their own kernel). That thinner boundary makes them lightweight but also means a kernel vulnerability can let a container process break out to the host. In practice, well-configured containers with seccomp profiles, AppArmor/SELinux policies, and non-root users inside are very hard to escape from — most successful container escapes exploit misconfigurations rather than kernel zero-days.

Secrets management is where most real-world compute security problems live, not in spectacular VM escapes. A *secret* in this context means any credential or key that grants access to something: a database password, an API key, a TLS certificate. The number-one practical failure mode is **committing secrets to version control** — a developer pastes an AWS API key into a configuration file and pushes it to GitHub. Automated bots scan public repositories for credential patterns and find them within

minutes. The right pattern is to store secrets in a dedicated secrets manager (HashiCorp Vault, AWS Secrets Manager, Azure Key Vault) and inject them into the environment at runtime, never baking them into images or code.

RBAC at the orchestrator — Role-Based Access Control in Kubernetes and similar platforms — is the question of who is allowed to do what inside the compute environment, not just who can reach it from outside. In Kubernetes, this means controlling who can run `kubectl exec` to drop a shell into a running container (equivalent to "log in to the running server"), who can read secrets, and who can deploy new workloads. Poor RBAC configuration — such as granting wildcard permissions in a cluster to make things "just work" — is one of the most common security findings in cloud-native infrastructure audits.

6. The shape of defense — five layers

Identity is the new perimeter. The old model (firewall around the corporate network, everything inside is trusted) is dead — your employees are working from coffee shops and your services are spread across clouds. The defining shift of the past decade is **zero trust**: never trust based on network location; verify every request based on identity. Practically that means **MFA** everywhere (multi-factor auth: password + a second factor like an authenticator app or a hardware key — both far stronger than SMS), **SSO** (single sign-on, to reduce password proliferation), and **least privilege** (every account gets only what it strictly needs).

Patch fast. Most successful breaches exploit vulnerabilities with patches already available — sometimes for months. The **CVE** system (cve.org) is the public registry of known vulnerabilities; the **CVSS** score grades severity. Auto-update where you can; prioritize the rest by severity and known-exploited status (CISA's **KEV** catalog — the short-list of vulnerabilities actively being used in attacks *right now*).

Encrypt at rest and in transit. HTTPS / TLS 1.3 for network traffic; disk encryption (BitLocker, FileVault, LUKS) for laptops; encrypted backups in storage. **Back up — and test the restore.** Most ransomware victims discover their backups didn't work *during* the incident (Sophos *State of Ransomware*). Offline / immutable backups beat cloud-only backups when the cloud account itself is what got owned. **Monitor.** A **SIEM** (Security Information & Event Management — Splunk, Microsoft Sentinel, Elastic, Google Chronicle) aggregates logs and alerts on suspicious patterns. **Dwell time** —

how long an attacker is inside before detection — is the most-watched metric; the 2026 median is still measured in weeks per Mandiant M-Trends.

7. Who does security in a company?

Security engineers build defensive systems — they write the IAM rules, harden cloud accounts, integrate scanners. **SOC analysts** sit in a **Security Operations Center** and watch alerts in real time, triaging which are real incidents. **Incident responders** (sometimes the same people, sometimes a specialized **DFIR** team — Digital Forensics & Incident Response) handle a breach end-to-end: contain, eradicate, recover, write the post-mortem. **Penetration testers** (pentesters) and **red teams** are paid to attack the company's own systems to find weaknesses before real attackers do. The **CISO** (Chief Information Security Officer) owns the security program at the executive level. As of 2026 the explicit expectation is that every developer carries security responsibility — the **DevSecOps** label captures the shift (Lesson 10's pipeline, plus security automation baked in: dependency scanning, secret scanning, infrastructure-as-code linting, all running on every pull request).

What people argue about

- **Pay the ransom or not?** Law enforcement says no — payments fund the next attack and there's no guarantee the decryption key works. Insurers and CFOs sometimes say yes — it's cheaper than the downtime. The honest answer depends on whether your backups actually work, which most organizations only find out *during* the incident.
- **Security through obscurity.** Universally derided in textbooks; quietly relied on in many real systems. The mature view: obscurity is fine as one layer in defense in depth — fatal as the only layer.
- **Open source vs proprietary for security tools.** "Many eyes make all bugs shallow" vs "obscurity buys time"; both camps have war stories (and the xz-utils near-miss was cited by both). Most modern security stacks blend the two.
- **Perimeter defense: dead or alive?** "The perimeter is dead" is a useful provocation against naive castle-and-moat thinking — but in practice, most organizations still have firewalls, VPNs, and segmented networks, and they

still catch real attacks. The honest answer: a perimeter alone is not enough; a perimeter plus zero-trust identity plus monitoring is still the dominant model in 2026.

- **Are containers a security sandbox?** Containers are not a security boundary in the same way VMs are — they share the host kernel. Whether that matters in practice depends entirely on how well the rest of the configuration is hardened. The debate is less "containers vs VMs" and more "what's your threat model?" For most workloads, well-configured containers are secure enough; for high-sensitivity multi-tenant environments (e.g., running untrusted user code), you want VM-level isolation or specialized sandboxing (gVisor, Kata Containers).
- **Zero trust — paradigm shift or rebranded segmentation?** Critics point out that "zero trust" often describes what careful practitioners were already doing with strong identity, MFA, and least-privilege — repackaged as a marketing term by vendors. Proponents argue the label matters because it forces a explicit policy conversation about every trust boundary. Both are partly right.

Reading on

- **Verizon Data Breach Investigations Report (DBIR)** — free annual; the canonical "what actually happened last year" report, with a stable methodology that lets you compare year to year.
- **Mandiant M-Trends** — free annual; complements the DBIR with an incident-response perspective (dwell time, attacker tooling, ransomware trends).
- **OWASP Top 10** — owasp.org/Top10. The canonical web-application vulnerability list, updated every few years; the one industry artifact every developer should be able to recognize.
- **Krebs on Security** — krebsonsecurity.com. Independent journalist Brian Krebs; the easiest-to-read running diary of the cybercrime ecosystem, with the human stories behind the headlines.
- **Suricata IDS/IPS** — suricata.io. The leading open-source network intrusion detection and prevention engine. The project website has readable documentation on how rules are written and what network-layer threats look like from a defender's perspective — good context even if you'll never run it yourself.

- **HashiCorp Vault documentation — "What is secrets management?"** — developer.hashicorp.com/vault/docs/what-is-vault. A concise, vendor-neutral framing of the secrets-management problem and the pattern for solving it.
- **CISA Zero Trust Maturity Model** — cisa.gov/zero-trust-maturity-model. The U.S. government's practical guide to zero trust, written for practitioners but readable as a framework overview. Useful for understanding what "zero trust" means operationally, beyond the vendor marketing.

LAST-REVIEWED: 2026-06-02

Homework — Security Fundamentals

Estimated time: ~50 min · No code · A browser, your real email address, and a willingness to look at uncomfortable truths

Three tool-tour exercises. You'll learn four things by doing them, in this order: (1) your data is almost certainly already in a breach somewhere — this is normal, and the question is what to do about it; (2) a password manager plus **MFA** on one important account takes ten minutes and removes most of the risk; (3) what your home network's firewall exposes to the internet — connecting the theory page's "network-layer defense" section to something you can see; (4) how the secrets-management problem looks in practice, through GitHub's automated scanning tools.

Before you start

Keep a one-page notebook open (paper or a text file). For each task, write the short answers under (a)/(b)/(c). You are not turning this in. Nothing here requires you to install software or run a command — every task is a few clicks in a browser on sites that are free to everyone.

The three tasks

Task 1 — Check if your email is in a breach

Open haveibeenpwned.com (HIBP — the industry-standard breach lookup service run by Troy Hunt). Enter your most-used email address in the search box. Read the results.

If you've used the internet for more than ~3 years, expect *multiple* hits — this is normal, not alarming on its own. It just means a site you once signed up for got breached at some point and your email (and sometimes more) ended up in a public dump.

Now click into one of the named breaches and read the "**Compromised data**" list. This is the most useful part: it tells you specifically what about you leaked from that one site — just an email and password hash? Or also phone numbers, addresses, dates of birth, security questions?

Self-check — answer all three:

- **(a)** How many breaches does your email appear in?
- **(b)** What's the date of the most recent breach in the list?
- **(c)** For that most-recent breach, name two or three classes of data that were exposed (e.g. passwords, phone numbers, physical addresses).

Bonus: click **Notify me** and subscribe your address to future-breach alerts — you'll get an email the next time your address shows up in a new dump.

✓ Take a screenshot of the result page for your own notes. (You can redact your email if you'd like to keep it private.)

Task 2 — Set up (or audit) a password manager + MFA on one account

If you don't already have a password manager, sign up for a free tier of any of these:

- **Bitwarden** — free, open source (bitwarden.com)
- **1Password** — free trial (1password.com)
- **Proton Pass** — free (proton.me/pass)
- Your browser's built-in — **Apple iCloud Keychain**, Chrome / Edge / Firefox built-in password manager

Install the browser extension if your choice has one. Then pick *one* important account — your primary email, your bank, or your main work account — and do these three things:

1. Check whether its current password is unique to that account (the password manager will usually flag reused passwords for you).
2. If it isn't unique, change it to a fresh password generated by the password manager.

3. Turn on **MFA** (multi-factor auth, sometimes called 2FA / two-factor authentication). Look in *Account Settings* → *Security* → *Two-Factor Authentication*.

Which MFA method to choose?

Strongly prefer an **authenticator app** (Google Authenticator, Microsoft Authenticator, Authy, 1Password / Bitwarden's built-in TOTP) over SMS. SMS-based MFA is much better than nothing, but is vulnerable to SIM-swap attacks. If the account offers a **hardware key** option (YubiKey, Google Titan), that's the strongest choice of all.

Self-check — answer all three:

- **(a)** Which password manager did you choose, and one sentence on why?
- **(b)** Which MFA method did you turn on (SMS, authenticator app, hardware key)? If you picked SMS, write one sentence on what you'd need to switch to an authenticator app.
- **(c)** Take a screenshot of the account's security settings page showing that MFA is now *on*.

✓ **Ten minutes of work on one important account removes most credential-stuffing risk for that account. The lesson from Attack 2 in the theory page is: do this for your email *first*, because your email is the password-reset path for everything else.**

Task 3 — Tour network and compute security in the real world

Two quick browser visits that connect what you just read (Themes A and B) to tools real security teams use.

Part A — ShieldsUP! (network-layer exposure)

Open grc.com/shieldsup (Gibson Research Corporation's ShieldsUP! tool — free, no account required). Click "**Proceed**", then run the "**All Service Ports**" scan. The tool probes your public IP address from the outside to see which ports are open, closed, or in "stealth" mode.

What you're seeing is exactly what an attacker running an internet-wide port scanner sees when they reach your router or firewall. Each *open* port is a service the internet can try to talk to. Each *stealth* result means the firewall is silently dropping probes rather than replying at all — the preferred configuration, because it reveals less to scanners.

Self-check — answer all three:

- **(a)** How many ports are shown as *open* vs *stealth*? (Write down the rough split.)
- **(b)** Were any ports *open*? If so, look up one of them on speedguide.net/port.php?port=XXXX (replace XXXX with the port number) and write one sentence on what service typically uses that port.
- **(c)** In one or two sentences, describe which defense concept from the "Security in the network layer" section of the theory page this scan is demonstrating.

✓ Most home routers put everything in stealth mode by default — that's the firewall doing its job. If you see open ports you didn't expect, write them down; it's worth investigating what's listening on them.

Part B — GitHub's secrets-scanning alert page (compute/secrets management)

Open docs.github.com/en/code-security/secret-scanning/about-secret-scanning. Read the first two sections (roughly 5 minutes). GitHub scans every public repository push for recognized credential patterns — API keys, OAuth tokens, private keys for dozens of providers — and alerts the repository owner (and sometimes the affected service provider directly) when one is found.

This is the automated response to the problem described in the "Secrets management" part of the theory page: developers accidentally commit credentials to code and push them to version control, where automated bots (both benign like GitHub's scanner and malicious) find them almost immediately.

Self-check — answer all three:

- **(a)** According to the page, what happens when a supported secret pattern is detected in a public repository?

- **(b)** What is "push protection," and in one sentence, why does it matter more than after-the-fact alerts?
- **(c)** Write one or two sentences connecting what you read to the "Secrets management" section of today's theory page — specifically, why a secrets manager (Vault, AWS Secrets Manager) solves the problem at the source rather than at detection.

✓ You don't need a GitHub account to read the documentation page. The takeaway: even the platforms we use to store code have had to build detective controls because committing secrets is genuinely easy to do by accident.

What "done" looks like

You're done when, in your notebook, you have:

- a breach count, a most-recent breach date, and one screenshot from Task 1 (HIBP)
- a password manager chosen, an MFA method chosen, and one screenshot of MFA enabled from Task 2
- a port-scan summary + one open-port service identified + one connecting sentence from Task 3 Part A (ShieldsUP!)
- three answers about GitHub secret scanning from Task 3 Part B

The aim was never to make you a security engineer in 45 minutes. It was to do the two things that actually move the needle for almost everyone — unique passwords and MFA on the accounts that matter — and to leave you with a concrete picture of how network and compute-layer defenses look from the outside.

Where this goes next

Lesson 11 closes Module 3 (How Software Runs & Stays Safe). Module 4 opens the AI era — Lesson 12 covers AI & ML Foundations, and Lesson 13 covers the LLM & agent era. Security comes back as a cross-cutting concern there too:

prompt injection, model abuse, and AI-augmented social engineering are all live topics as of 2026.

Lesson 12: AI & ML Foundations

Lesson 12 — AI & ML Foundations

The world before LLMs — what machine learning is, what models do, training vs inference.

IT Landscape

Module 4

The words “AI” and “ML”

“AI is the dream. ML is the technique that’s actually working.”

- **AI** (Artificial Intelligence) — the broad goal: machines that act intelligently.
 - **ML** (Machine Learning) — one family of techniques where the machine *learns from data* instead of being explicitly programmed.
 - Most of what news outlets call “AI” in 2026 is actually ML.
-

Programming vs learning

Programming

A human writes the rules, step by step.

"If temperature > 30, send alert."

Machine learning

A human gives the machine thousands of past examples (temperature + did-something-bad-happen). The machine works out the rule itself.

ML shines where rules are too complex for humans to write.

What is a “model”?

“A model is a math function shaped by data.”

- **Input** goes in — a photo, a sentence, a row of features.
 - **Output** comes out — a label, a number, a probability.
 - The “shape” of that function — the **parameters** — was learned from training data.
 - Same architecture, different data → **different model**.
-

Training vs inference — the two phases

Training

Slow, expensive, done once (or periodically).

The model sees millions of labelled examples and adjusts its parameters until it predicts well.

Inference

Fast, cheap, done every time a user asks.

The trained model takes new input and produces output.

“Training is months of GPU bills; inference is milliseconds per request.”

Supervised, unsupervised, reinforcement

- **Supervised** — given input + correct answer, learn the mapping (spam classifier, photo tagger, house-price predictor).
 - **Unsupervised** — given input only, find structure (clustering customers, detecting anomalies).
 - **Reinforcement** — an agent acts in an environment, gets rewards, learns a strategy (game-playing AI, robotics, ad bidding).
-

The classical ML toolkit

Techniques that ran the field for ~30 years — and still ship today:

- **Linear regression** — predict a number.
- **Logistic regression** — predict a category.
- **Decision trees, random forests, gradient-boosted trees** — still the workhorses for tabular data.
- **k-means** — clustering.
- **SVMs** — classification.

In 2026, the boring tabular-data winner is still gradient-boosted trees, not deep learning.

Neural networks — what they actually are

- A stack of math operations (multiplications + a nonlinear squash) arranged in **layers**.
 - Each layer transforms its input; the final layer produces the answer.
 - Training nudges the millions of internal numbers (**weights**) so outputs match the training data.
 - The “deep” in deep learning just means **many layers**.
-

Why deep learning took off (~2012)

Three things had to line up:

- **Big labelled datasets** — ImageNet, the web.
- **GPUs** — parallel math, originally for video games, repurposed for neural nets.
- **A few algorithmic tricks** — better optimizers, better activations, better regularization.

Once they did, deep learning crushed classical techniques on **images**, then **speech**, then **language**.

Frameworks people use (as of 2026)

Framework	Owner	Where it fits
PyTorch	Meta	Research + production winner
TensorFlow / Keras	Google	Still huge, especially in industry
JAX	Google	Research darling
scikit-learn	Community	The classical-ML standard
Hugging Face Transformers	Hugging Face	The LLM bridge

Most ML engineers use 2 or 3 of these daily.

Where ML actually ships (none of these are LLMs)

- **Fraud detection** — banks, every credit-card swipe.
- **Recommender systems** — Netflix, Spotify, TikTok feed.
- **Search ranking** — Google, Amazon.
- **Medical imaging** — cancer detection in scans.
- **Speech recognition** — Siri, Alexa, captions.
- **Forecasting** — electricity grid, ride-hailing demand.
- **Autonomous-driving perception.**

The world has been “running on ML” for a decade.

Recap

- **ML = learning from data**, not from rules.
- The workflow is **train once, infer many times**.
- **Deep learning** is one corner of ML — very good at perception.

Coming up: Lesson 13 — what changed in 2022 when LLMs arrived, and where agents are taking us next.

AI & ML Foundations

Reading time: ~10 min · Prereqs: Lessons 01–06 · *Snapshot 2026 — the concepts are stable; brand-name lists drift.*

Why this matters to you

"AI" is the most overloaded word in the 2026 vocabulary. The press uses it for everything from a chatbot answering email, to a self-driving car, to the spam filter on your phone, to the line in a press release that closed a funding round. Underneath the press confusion is a real and old engineering discipline called **machine learning** — and *most* of the so-called "AI" you actually use is ML in a quiet, boring suit. This lesson is the ground truth for everything in Lesson 13. Once you can tell training from inference, supervised from unsupervised, a model from an algorithm, the LLM era stops being magic and starts being a special case of something you already understand.

1. AI vs ML vs deep learning vs LLMs

Think of nested circles. **AI** (artificial intelligence) is the outermost: the broad ambition of building machines that act intelligently — a goal pursued since the 1950s with everything from rule-based expert systems to chess engines to today's statistical approaches. Inside AI sits **ML** (machine learning) — the family of techniques where the machine learns from data rather than from hand-written rules. Inside ML sits **deep learning** — ML done with deep neural networks. And inside deep learning, finally, sits **LLMs** (large language models, the subject of Lesson 13) — a particular flavour of deep learning trained on enormous amounts of text. Most of what newspapers call "AI" today is ML, often deep learning, sometimes specifically LLMs. The distinctions matter when you read a vendor's claim and want to know which kind of "AI" they actually mean.

2. The big shift: rules vs learning

Traditional programming says *"if this, do that."* A programmer writes the rules; the machine follows them. ML inverts it. You give the machine many *examples* of the input

plus the right answer ("here's a photo + 'cat'; here's another photo + 'dog'..."), and the machine works out the rule itself by adjusting an internal mathematical function until the answers come out right. This matters because **rules are easy to write for some problems and impossible to write for others**. Rules for taxes: tedious but doable. Rules for "is this photo a cat?": impossible — nobody can write that down in if-statements. ML lets us solve problems we cannot articulate but can demonstrate by example.

3. What a "model" actually is

A **model** is a mathematical function shaped by data. It has two parts: a **shape** (the architecture — for example, a particular kind of neural network with a certain number of layers and a million, or a billion, parameters) and a **set of numbers** (the parameters — the actual weights inside the function, learned from training data). Inputs flow in (a photo, a sentence, a row of features), the model multiplies and adds and squashes, and an output comes out (a label, a number, a probability). The same architecture trained on different data produces a different model. Once trained, a model is essentially a giant file of numbers — megabytes, gigabytes, occasionally terabytes — that knows how to map inputs to outputs for the task it was trained on, and nothing else.

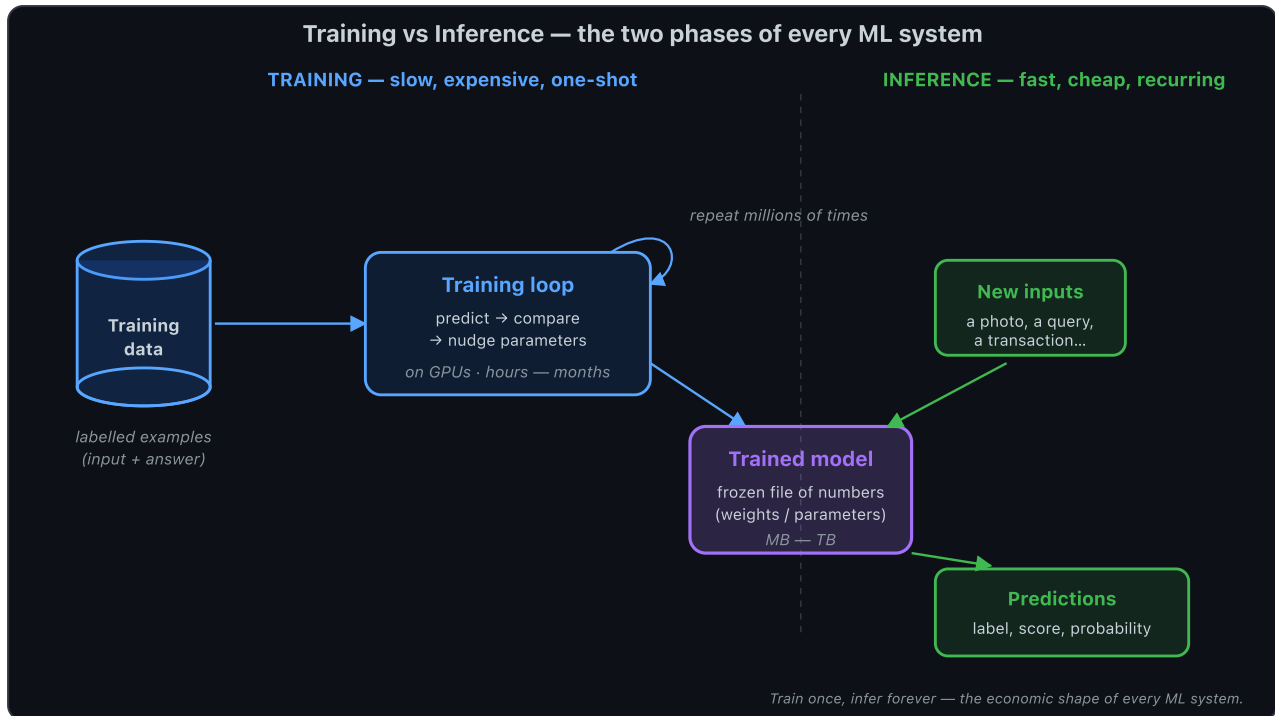
4. Training vs inference — the two phases

Every ML system has two phases. **Training** is the slow, expensive phase that produces the model. The system shows the model millions of labelled examples (or, in unsupervised cases, just lots of data); after each one it adjusts the parameters slightly so the next prediction is a little better. This loop runs on banks of **GPUs** (graphics processing units — the same chips that render video games, repurposed because they multiply matrices very fast) for hours, days, or — for the largest models — months.

Inference is the fast, cheap phase that *uses* the model. New input arrives, the model applies its frozen parameters, and produces an output in milliseconds. The economic shape matters: training is a few enormous, one-shot bills; inference is many small, recurring bills (every search query, every photo classification, every chatbot reply). The "AI is expensive" headlines in 2026 are usually about one or the other; knowing which lets you read the article correctly.

5. SVG diagram — the train-then-infer flow

The same idea drawn as a picture: on the left, a training dataset feeds into a training loop that adjusts model parameters; the loop produces a trained model in the middle; on the right, new inputs enter the trained model and predictions come out.



6. The three flavours of learning

Three top-level paradigms cover most of ML.

- **Supervised learning** — you have input plus the correct answer ("here are 100 000 photos labelled cat / dog / bird; here are 100 000 emails labelled spam / not-spam"). The model learns the mapping. Most production ML — fraud detection, spam filters, photo tagging, price prediction — is supervised.
- **Unsupervised learning** — you have inputs but no labels. The model finds structure in the data: clusters of similar customers, anomalies that don't fit the pattern, useful low-dimensional summaries of high-dimensional data.
- **Reinforcement learning** — an agent acts in an environment, gets rewards for good outcomes, and learns a strategy. Famous for game-playing (AlphaGo) and increasingly used in robotics, recommender systems, and — newly relevant — in the post-training step that makes LLMs follow instructions, called **RLHF** (reinforcement learning from human feedback).

A fourth, increasingly important camp is **self-supervised learning**, which is how LLMs learn: the data labels itself ("given this sentence with one word masked, predict the word"). It looks like supervised learning to the training loop, but no humans had to write any of the labels.

7. The classical ML toolkit, still earning its keep

Long before deep learning, ML had a rich toolkit that still ships today, especially for **tabular data** (rows and columns, like a spreadsheet). **Linear** and **logistic regression** — the simplest things that work, often the right starting point. **Decision trees** and their ensembles — **random forests** and **gradient-boosted trees** (libraries *as of 2026*: **XGBoost**, **LightGBM**, **CatBoost**) — are quietly the winning technique for most tabular problems in 2026, beating deep learning on most non-perceptual data. **k-means** for clustering. **SVMs** (support vector machines) for classification with small datasets. **scikit-learn** is the standard Python library that bundles most of these.

8. Deep learning — and why 2012 mattered

Neural networks had existed since the 1950s, but stayed niche until around 2012, when three things lined up at the same time: huge labelled datasets (ImageNet — a million labelled photos), cheap parallel hardware (NVIDIA's GPUs, originally for video games), and a handful of algorithmic refinements. Once they did, deep learning leapfrogged classical techniques on perceptual tasks: first image recognition, then speech, then language translation, then game-playing. The same recipe — a deep neural network, lots of data, lots of GPU time — kept scaling. By the late 2010s, the recipe had taken over every perceptual ML task. Frameworks that ship this work *as of 2026*: **PyTorch** (Meta — the research and increasingly production winner), **TensorFlow** / **Keras** (Google, still very large in industry), **JAX** (Google, research darling). For LLMs specifically, **Hugging Face Transformers** is the standard library on top.

9. Where ML actually ships today

Long before any chatbot reached your screen, ML was already running underneath things you use every day. A non-exhaustive list *as of 2026*: **fraud detection** on every credit-card swipe; **recommender systems** in Netflix, Spotify, TikTok, YouTube; **search**

ranking in Google and Amazon; **speech recognition** in Siri, Alexa, and your phone's captions; **medical imaging** (cancer detection on scans); **demand forecasting** (electricity grids, ride-hailing); **autonomous-vehicle perception**. The bulk of ML in production today is *not* generative AI — it is these older, less glamorous, deeply embedded systems. Lesson 13 covers the generative wave that joined them.

What people argue about

- **Symbolic AI vs statistical AI.** The field's oldest fight. Statistical / ML approaches dominate today; a vocal minority argues that pure ML cannot reach human-level reasoning without re-introducing symbolic components (knowledge graphs, logic, explicit rules).
- **Is bigger always better?** For the past five years, scaling up data + parameters + compute kept producing better models. Some researchers believe the curve will soon flatten; others believe it will not, and that the next wave of capability is one more order of magnitude away.
- **Do models *understand*?** A model that predicts correctly may or may not be "doing what humans do". Cognitive scientists, ML researchers, and philosophers have not converged — and it matters more than it sounds, because policy and safety arguments rest on the answer.
- **Deep learning vs gradient-boosted trees on tabular data.** A small but real flame war. *As of 2026*, the trees usually still win — despite a decade of deep-learning attempts to dethrone them.

Reading on

- **3Blue1Brown — "Neural Networks" YouTube series.** The clearest visual intro to what a neural network actually computes, no math required to follow along.
- **fast.ai.** Free, practical course on building real models with PyTorch — top-down, code-first, very accessible for a non-academic reader.
- ***Hands-On Machine Learning with Scikit-Learn and PyTorch*** by Aurélien Géron — the standard practitioner book; covers both the classical and the deep-learning halves of this lesson in depth.

- **Lesson 13 of this course** — the LLM era, picking up exactly where this lesson stops.

LAST-REVIEWED: 2026-05-25

Homework — AI & ML Foundations

Estimated time: ~45 min · No code you have to run · A browser, a webcam, and a notebook

Three tool-tour exercises. The point is to put your hands on three very different faces of machine learning: *training* a tiny model yourself in a browser, *using* a deployed ML model that is already shaping what you see every day, and *reading* the raw material a real ML problem actually starts from — a dataset. By the end you'll have felt all three sides of the train / infer / data loop from Lesson 12's theory.

Before you start

Keep a one-page notebook open (paper or a text file). For each task, write the short answers under (a)/(b)/(c). You are not turning this in — the goal is to build the muscle of *poking at* an ML system instead of just reading about one. There is **no code you need to run** in this homework; Task 1 trains a model with mouse clicks and your webcam.

The three tasks

Task 1 — Train a model in your browser (Teachable Machine)

Open teachablemachine.withgoogle.com. Click **Get Started → Image Project → Standard image model**. You'll see two empty class boxes (*Class 1* and *Class 2*).

Rename them to any pair of webcam-friendly opposites — e.g. "*with glasses*" / "*without glasses*", or "*thumbs up*" / "*thumbs down*", or "*pen visible*" / "*no pen*".

For each class, hold down the **Webcam** button to record ~30 sample frames while you act out that class (move slightly, change angle a bit — you want variety, not 30 identical frames).

Click **Train Model**. It takes ~30 seconds — that is a real training run, just on a very small model. When it's done, the **Preview** panel comes alive and shows two prediction bars updating in real time as you move in front of the webcam.

Now play. Hold up each scenario in turn, then deliberately hold up *something the model has never seen* (an unrelated object, your other hand, your coffee mug). Watch what the prediction bars do.

Self-check — answer all three:

- **(a)** Which class scored highest when you held up the "correct" sign for it? Roughly what confidence percentage?
- **(b)** What happened when you held up something the model had never seen? (Hint: it still picked one of the two classes — the model has no concept of "*neither*" unless you teach it one. This is a foundational ML behaviour worth feeling once.)
- **(c)** What would you change to make the model less confused? (Collect more examples per class? More *varied* examples — different lighting, different angles? Add a third class called "*neither*" and train on random background frames?) Take a screenshot of the prediction bars while you're holding up the "unknown" object.

✓ You have now run the full ML loop: collected labelled data, trained a model on it, and used the trained model for inference — all without writing a line of code.

Task 2 — See a deployed ML model recommend things to you

Open **Spotify**, **Netflix**, or **YouTube** — any logged-in account where the home page is personalised to you. (If you don't have a personalised account, ask a friend or housemate to let you peek at theirs for two minutes; the exercise needs *their* recommendations, not a fresh anonymous session.)

Scroll the home page slowly. You are looking at the output of an ML recommender system that has been quietly observing your behaviour for months or years — what you played, what you skipped, what you finished, what other people with similar taste enjoyed. Every row title is a hint at what signal it is using.

Self-check — answer all three:

- **(a)** Name *three* sections of the home page whose content was clearly picked for you personally (examples — Spotify: *Made For You*, *Daily Mix*, *Discover Weekly*, *Your Top Mixes*; Netflix: *Top Picks for [Your Name]*, *Because You Watched X*, *Continue Watching*; YouTube: the main feed, *Recommended*, *Mixes*).
- **(b)** Pick one specific item shown and write one sentence on why you think the recommender chose it for you. (A song by an artist you played last week? A film in a genre you finished three of? A video from a channel you subscribed to?)
- **(c)** Find one section on the same page that is *not* personalised — usually a globally trending or editorial row (e.g. *Top 10 in your country today*, *Trending Now*, *New Releases*). Write one sentence on the difference: what is the recommender doing in the personalised row that it deliberately is *not* doing in this global row?

What you just saw

This is one of the biggest ML deployments in the world by sheer number of inference calls per second — and it has been running long before "AI" was a press headline. Recommender systems were arguably the first ML technology most people on Earth used daily without noticing.

Task 3 — Read a Kaggle dataset description (don't write any code)

Open kaggle.com/datasets. You will see a feed of public datasets uploaded by the community. Browse for a minute, then pick a **tabular** dataset (rows and columns — not images, not text). Three that work especially well for this exercise:

- *Titanic — Machine Learning from Disaster*
- *House Prices — Advanced Regression Techniques*
- *Credit Card Fraud Detection*

On the dataset's page, open the **Data** tab (or scroll to the data card). You'll see a list of columns with their names and types — this is what a real ML problem looks like before any model touches it. Read the dataset description above the column list; one column will usually be flagged as the thing everyone tries to predict.

Self-check — answer all four:

- **(a)** What is the *target* — the single column a model would predict? (Titanic: `Survived` . House Prices: `SalePrice` . Credit Card Fraud: `Class` .)
- **(b)** What are the *inputs*? Name three of the other columns and, in plain language, what they describe.
- **(c)** Is this a *supervised* or *unsupervised* problem? (Hint from Lesson 12: if you can point at a target column, it's supervised.)
- **(d)** Click the **Code** tab at the top of the dataset and skim the count of public notebooks people have uploaded for this dataset. You do **not** need to read any of them — just notice that hundreds or thousands of people have already attempted this exact problem, and that this is what a healthy ML community looks like.

Don't get pulled in

Kaggle is fascinating and competitive and there are leaderboards and prize money. For *this* exercise the only goal is to see the **shape** of an ML problem — columns, a target, examples of solutions existing. Resist the urge to open ten notebooks; close the tab when you can answer (a)–(d).

What "done" looks like

You're done when, in your notebook, you have:

- two prediction-bar observations and one screenshot from Task 1 (training)
- three section names, one why-it-picked-this sentence, and one personalised-vs-global sentence from Task 2 (inference at scale)

- one target column, three input columns, one supervised/unsupervised verdict, and one observation of the Code tab from Task 3 (data)

The aim was never to make you an ML engineer in 45 minutes. It was to make sure that next time you read "we trained a model on user data and deployed it to production", you have a concrete picture in your head of *all three* nouns in that sentence — because you've now touched each of them with your own hands.

Where this goes next

Lesson 13 takes the same train / infer / data picture and zooms into the corner of ML that exploded in 2022: **LLMs** (Large Language Models), the agents built on top of them, and the product landscape they created. Everything you just felt — a model learning a pattern from data, then using that pattern on new inputs — still applies; the model just gets very, very much bigger, and the data is the entire internet.

Lesson 13: The LLM & Agent Era

Lesson 13 — The LLM & Agent Era

What changed 2022–2026 — LLMs, RAG, agents, MCP, the new product landscape.

IT Landscape

Module 4

High-volatility — as of 2026

Where we are in 2026

The most disruptive software shift since the smartphone — happening over three years.

- **Nov 2022** — ChatGPT released to the public.
- **By 2026** — AI assistants are embedded in IDEs, browsers, the OS, customer-service stacks, and a growing share of search.
- **Workflows are being rewritten** — coding, writing, support, research, design.

Snapshot — every number, every brand, and every pricing figure in this lesson has a “best before” date.

What is an LLM?

*A **Large Language Model** is a giant neural network trained to **predict the next word**.*

- **That's it.** One simple objective — applied at extreme scale.
 - **Scale** — trillions of words of training data, hundreds of billions of parameters.
 - **Out of that** emerges reasoning, translation, code generation, summarization, conversation.
 - It is exactly Lesson 12's "model" idea — with a particular shape (**transformer**) and a particular signal (next-word prediction).
-

The big players (as of 2026)

Lab	Family	Known for
Anthropic	Claude (Sonnet, Opus, Haiku)	Frontier reasoning, safety focus, long context
OpenAI	GPT family	Broadest consumer reach (ChatGPT)
Google DeepMind	Gemini family	Multi-modal, integrated with Search & Workspace
Meta	Llama family	Open-weights leader
Others	DeepSeek, Mistral, Alibaba Qwen, xAI Grok	Frontier or open-weights challengers

Names and rankings move quarterly — as of 2026.

Closed vs open weights (as of 2026)

Closed-weight

Claude, GPT-4-class, Gemini Pro.

- Best frontier capability.
- Accessed via the vendor's API.
- You do *not* run the model yourself.

Open-weight

Llama, Qwen, Mistral, DeepSeek.

- You can download and run yourself.
 - Great for privacy, cost, customization.
 - Capability gap to frontier is much narrower than in 2023.
-

What a prompt looks like

"You are a helpful assistant. Summarize the following email in two sentences and suggest one reply."

- A **prompt** is just text.
 - **Prompt engineering** — the craft of writing prompts that get good answers reliably.
 - **System prompt** (instructions, persona) vs **user prompt** (the actual question).
 - The model sees them concatenated, as one stream of text.
-

The context window

The context window is the model's RAM.

- Everything in one request — system prompt + history + documents + the new question — must fit.
 - As of 2026, common windows are **200K–2M tokens** (~150–1500 pages).
 - Anything beyond the window is invisible — the model “forgets” it.
 - **Pricing** happens here too — vendors charge per million input and output tokens (ballpark single dollars per million as of 2026; verify before quoting).
-

RAG — Retrieval-Augmented Generation

LLMs only know what they were trained on — their training has a **cutoff date**.

To answer questions about your private documents (or anything newer):

- **Retrieve** — pull the most relevant chunks from a database.
- **Augment** — paste them into the prompt.
- **Generate** — let the LLM answer using both.

The retrieval step typically uses a **vector database** (Lesson 07).

Most enterprise “ask our docs” chatbots are RAG under the hood.

Tools and function calling

A modern LLM can be given a list of **tools** — functions it may request:

- `search_web(query)`
- `send_email(to, subject, body)`
- `query_database(sql)`

The model decides one would help, asks the host program to call it, the result comes back into the conversation, and the model continues.

*This is how LLMs reach beyond their training data and **take action** in the world.*

Agents — LLMs in a loop

*An **agent** is an LLM that calls tools, looks at the result, and decides what to do next — in a loop.*

Not magic — a loop. Examples (as of 2026):

- **Cursor, Windsurf** — IDE agents that edit your code.
- **Claude Code** — terminal agent that runs commands and reviews their output.
- **Devin** — brands itself as an autonomous software engineer.
- **Operator** — web-browsing agent.
- **Salesforce AgentForce, Microsoft Copilot Studio** — enterprise platforms.

Each one lets the LLM iterate until the task is done — or it gives up.

MCP — the agent's USB-C (as of 2026)

Every agent needs to plug into your tools — Slack, Gmail, the filesystem, a database, the internal wiki.

MCP (Model Context Protocol) — introduced by Anthropic in late 2024 — is the open standard for how agents talk to those tools.

Clients that speak MCP (as of 2026):

- **Claude Desktop, Claude Code, Cursor, Windsurf, Cline.**
- VS Code (via extension) and a growing list of others.

*Write one tool as an **MCP server** — plug it into any compatible client. This platform has a whole **MCP Servers** course.*

The new product landscape (as of 2026)

Category	Names you'll hear
Chat assistants	Claude, ChatGPT, Gemini, Copilot, Grok
Coding	Cursor, Claude Code, GitHub Copilot, Windsurf, Cody
Search	Perplexity, Google AI Overviews, Brave Leo
Browsing & action	Operator, Comet, MS Copilot
Voice	Advanced voice models from each major lab
Image / video	Midjourney, DALL·E 3+, Stable Diffusion, Sora, Veo
Open-source orchestration	LangChain, LlamaIndex, LiteLLM

Every name on this slide is on `refresh-checklist.md`.

Recap

- An **LLM** is just a giant next-word predictor — at extreme scale.
- **RAG** and **tools** let it touch your data and act in the world.
- An **agent** is that loop, with permissions.

Coming up: Lesson 14 — who does what in IT, the open-source ecosystem, and which course on this platform to take next based on what you've enjoyed.

The LLM & Agent Era

Reading time: ~10 min · Prereqs: Lesson 12 (AI & ML Foundations) · *Snapshot 2026 — the highest-volatility lesson in the course. Brands, versions and prices move quarterly; concepts move slowly.*

Why this matters to you

The most public IT story of the early 2020s is the one happening in this lesson. ChatGPT released to the public in November 2022; three and a half years later, **Large Language Models** and the **agents** built on top of them are in your IDE, your browser, your customer-service queue, your search results, and — if you work in IT — almost every meeting. None of it is magic; all of it is a particular kind of machine learning (Lesson 12) applied at unprecedented scale, with a few clever wrappers (retrieval, tool use, loops). This lesson is the snapshot — *as of 2026* — of what changed, what the vocabulary means, and what new products you'll keep hearing about. Expect this page to be the most-edited file in the course a year from now.

1. What an LLM actually is

A **Large Language Model** is a giant neural network — usually a particular architecture called a **transformer** — trained on a huge corpus of text (the open web, books, code, some private data) with one deceptively simple objective: **predict the next word**. Show the model "The cat sat on the...", ask it what comes next; nudge its weights so it gets closer to "mat" (or whatever the training text actually said). Repeat trillions of times across hundreds of billions of parameters. Out of that one objective, applied at extreme scale, emerges the ability to translate, summarize, write code, hold a conversation, answer questions, and — controversially — *reason* about novel problems. The same recipe that produced Lesson 12's image classifiers, scaled to text and to a much bigger model, produced this.

2. The 2026 player landscape

High-volatility section — refresh annually. As of 2026 the **frontier closed-weight models** are roughly: **Anthropic's Claude** family (Opus, Sonnet, Haiku — known for reasoning, safety focus, and long context); **OpenAI's GPT** family (broadest consumer reach via ChatGPT, strong reasoning models); **Google DeepMind's Gemini** family (deeply integrated with Google Search, Workspace, and Android); and **xAI's Grok** in a smaller but visible role. The **open-weight** side is led by **Meta's Llama** family, with **Mistral**, **Alibaba's Qwen**, and **DeepSeek** as serious frontier-class open challengers — the open-weight gap to frontier closed models is much narrower *as of 2026* than it was in 2023. Closed-weight models you typically use via an API call to the vendor's cloud; open-weight models you can download and run on your own hardware (or via a hosting provider like **Together AI**, **Fireworks**, **Groq**, or **OpenRouter**). The right choice depends on cost, privacy, latency, and how much capability you actually need.

3. Prompts, context windows, and tokens

A **prompt** is just text — typically a **system prompt** (instructions, tone, persona — "You are a helpful assistant...") plus a **user prompt** (the actual request). The model sees them concatenated and produces a response. Everything the model knows about your current task lives in the **context window** — its working memory for one request. Modern context windows *as of 2026* range from ~128K tokens at the small end to 1M+ tokens at the large end (roughly 100 000 to 1 000 000 *words*, depending on language). Anything beyond the window simply isn't visible. Pricing also happens here: vendors charge per million tokens of input and per million tokens of output, with input usually cheaper. Real prices *as of 2026* are in the small-dollars-per-million-tokens range and have been falling roughly an order of magnitude every 18 months — verify before quoting.

4. RAG — giving the model your data

An LLM only knows what it saw during training, which has a **knowledge cutoff** date. To answer questions about your private documents — or anything newer than the cutoff — the standard pattern *as of 2026* is **RAG (Retrieval-Augmented Generation)**: when a question comes in, first **retrieve** the most relevant chunks of your documents (using **embeddings** stored in a **vector database** — see Lesson 07), then **augment** the prompt with those chunks ("Here are five relevant excerpts. Now answer the user's question."), then **generate** the answer. Most enterprise "ask our internal docs"

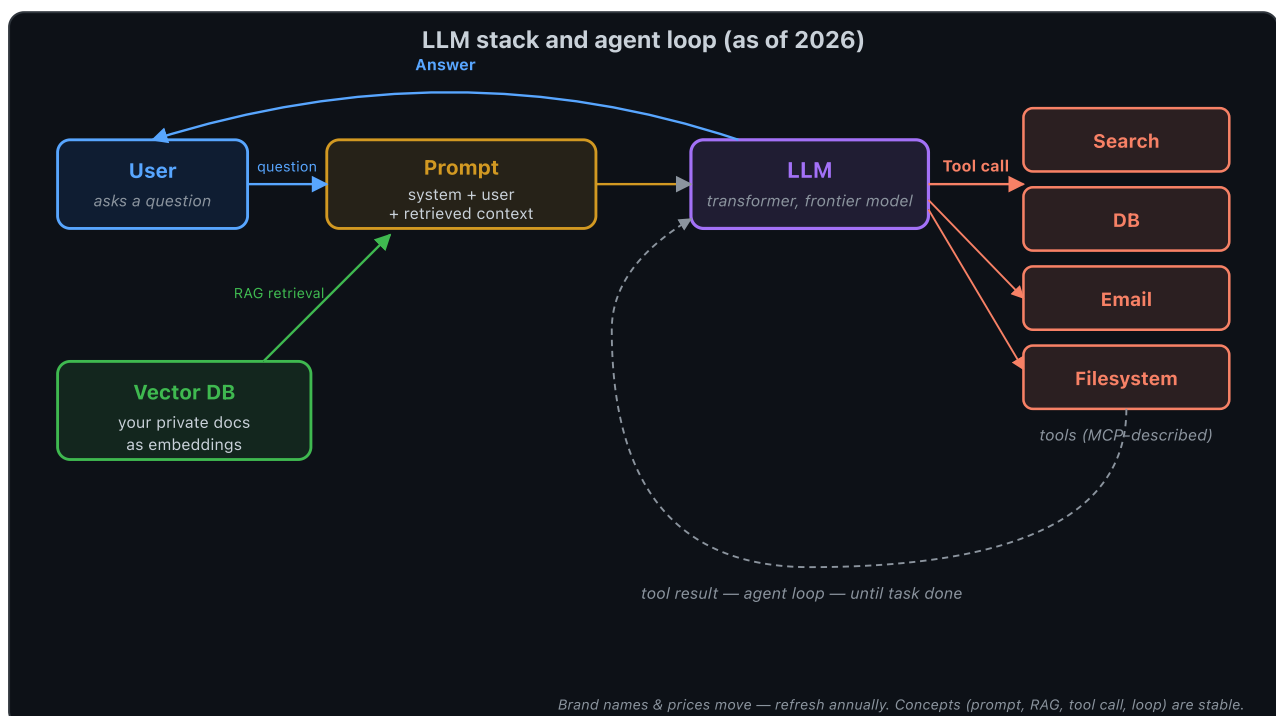
chatbots, most documentation chatbots, and a meaningful slice of customer-support bots in 2026 are RAG under the hood. RAG also keeps the answer grounded — if the retrieved chunks contradict the model's prior, the prompt usually wins.

5. Tools, function calling, and reaching the outside world

A modern LLM can be given a list of **tools** — described as functions with names, parameters, and what each does (`search_web(query)`, `send_email(to, subject, body)`, `query_database(sql)`). The LLM doesn't run the tool itself; it *requests* a call ("call `search_web` with `query='latest GDPR ruling'`"), the host program executes the call, the result is fed back into the conversation, and the model continues. This is how LLMs reach beyond their training data and *take action* in the world. It is also the building block of the next section.

6. The LLM stack and agent loop, one diagram

Here is the same idea drawn as a flow: a **User** asks a question; relevant chunks come out of a **Vector DB** via *RAG retrieval* and are folded into the **Prompt**; the **LLM** either returns a final **Answer** or emits a **Tool call** to one of several labelled tools. Tool results flow back into the model and the loop repeats until the task is done. That outer loop is what makes a chatbot into an **agent**.



7. Agents — LLMs in a loop

High-volatility — refresh brand names annually. An **agent** is, simply, an LLM that calls tools, looks at the result, decides what to do next, and *loops* — until the task is finished or it has decided it can't finish. Not magic; a loop. As of 2026 the agent landscape is busy: **Cursor** and **Windsurf** are IDE-embedded agents that edit your codebase on instruction; **Claude Code** is a terminal-first agent that runs commands and reviews their output; **Devin** brands itself as an autonomous software engineer; **Operator** is OpenAI's browser-driving agent; **Comet** (Perplexity's browser) is in the same family. Enterprise platforms like **Salesforce AgentForce** and **Microsoft Copilot Studio** package the same pattern. Open-source toolkits — **LangChain**, **LlamaIndex**, **AutoGen**, **CrewAI**, **PydanticAI**, **smolagents**, the **OpenAI Agents SDK**, the **Anthropic Claude Agent SDK** — let you build your own. The pattern is consistent across all of them; only the wrappers differ.

8. MCP — the agent integration standard

Every agent needs to plug into the user's tools — Slack, Gmail, GitHub, the filesystem, the database, the internal wiki. Without a standard, every agent vendor had to write a custom integration for every tool. **MCP (Model Context Protocol)** — introduced by **Anthropic** in late 2024 and rapidly adopted — is the open standard for how an agent talks to its tools. As of 2026, **Claude Desktop**, **Claude Code**, **Cursor**, **Windsurf**, **Cline**, **VS Code** (via extension), and a growing list of other clients all speak MCP. The payoff: write your integration once as an **MCP server** and it plugs into any compatible agent. This platform has a dedicated **MCP Servers** course covering it in depth.

What people argue about

- **Will scaling keep working?** For five years, bigger models on more data have kept producing better capabilities. As of 2026, some labs believe the curve will keep going; others believe we'll hit a wall and need new techniques (more data efficiency, neuro-symbolic hybrids, new architectures). Nobody knows yet.
- **Open weights vs closed weights.** Open-weight advocates argue safety, sovereignty, and innovation; closed-weight advocates argue frontier capability

and responsible release. Both camps publish strong work, and the gap between the two has narrowed *as of 2026*.

- **Confabulation (a.k.a. hallucination).** LLMs sometimes confidently produce wrong answers. RAG, tool use, and post-training reduce but do not eliminate this. How much to trust LLM output in high-stakes domains (medical, legal, financial) is an active debate.
- **Prompt injection and security.** When an LLM reads untrusted text (a web page, an email, a PDF), that text can contain instructions that the model may follow. *As of 2026* this is still unsolved — mitigated, not fixed.
- **Agent autonomy.** How much should an agent do without asking? The "ask permission" vs "ask forgiveness" trade-off is the live UX argument of 2026 — especially once an agent can spend money, send email, or push code.

Reading on

- **Anthropic's "Building Effective Agents"** — anthropic.com/research/building-effective-agents. Short, practical, written by the lab that built Claude. The best single starting point for the agent pattern.
- **Simon Willison's blog** — simonwillison.net. The most-read LLM blog in the field *as of 2026*; updated weekly with new model releases, prompt-injection findings, and tool experiments.
- **The Claude & LLMs course on this platform** — claude/claude-course.html. The deep dive: 33 lessons covering API usage, prompt engineering, RAG, fine-tuning, evals, and deployment.
- **The MCP Servers course on this platform** — if the agent-integration story above interested you, the dedicated course walks through writing your own MCP server end-to-end.

LAST-REVIEWED: 2026-05-26

Homework — The LLM & Agent Era

Estimated time: ~45 min · No code you have to run · A browser, a free account on a couple of AI products, and a notebook

Three tool-tour exercises that put you in front of the three shapes of LLM product you'll keep meeting in 2026: a *chat assistant* (and the differences between the big three), a *retrieval-augmented* reader that grounds answers in your own documents, and an *agent* that actually carries out a multi-step task. By the end you'll have felt — not just read about — the knowledge-cutoff problem, what citations are for, and the gap between a chatbot reply and an agent run.

Before you start

Keep a one-page notebook open (paper or a text file). For each task, write the short answers under (a)/(b)/(c). You are not turning this in — the goal is to build the muscle of *poking at* an LLM product instead of just reading about one. There is **no code you need to run** in this homework. Free accounts on the products mentioned are sufficient as of 2026; this is a fast-moving field, so if a UI element has moved by the time you read this, pick the nearest equivalent.

The three tasks

Task 1 — Side-by-side: ask three chatbots the same questions

Open three browser tabs: claude.ai, chatgpt.com, and gemini.google.com. Sign in with free accounts — all three vendors offer a free tier as of 2026 (a paid tier exists too, but the free tier is enough for this exercise).

Ask **the same three questions** to each of the three chatbots. Use the same wording in all three tabs so the only variable is the model:

1. *"In two paragraphs, explain what an LLM is to a non-technical friend."*
2. *"What were the most important AI news stories in the last 30 days? Be specific."*
3. *"What is your knowledge cutoff date?"*

Take a screenshot of each Q1 answer so you can compare them afterwards.
(You're not grading them — you're feeling the style differences.)

Self-check — answer all three:

- **(a)** Whose explanation in Q1 felt clearest to *you*? There is no wrong answer — note the difference in style (length, use of analogy, presence of headings, tone). Three frontier LLMs reading the same prompt will produce three recognisably different "voices".
- **(b)** For Q2 — did each model claim to know recent news, or did one or more say *"I don't have access to current information"*? This is the **knowledge cutoff** problem from the theory: a static LLM only knows what it was trained on. Note which models have a web-search tool turned on by default (recent answers with citation links) versus which are answering from the static weights only.
- **(c)** For Q3 — write down each model's stated cutoff date. Compare with the actual date today. The gap is the window the model literally cannot see — and it is why, for time-sensitive questions, you either need a model with tools (web, search) or a RAG system on top.

✓ You have now run a controlled comparison of the three leading consumer chat assistants on the same prompts — the most common "which one should I use" question, answered with evidence instead of vibes.

Task 2 — See RAG in action (NotebookLM)

Open notebooklm.google.com (a free Google account is enough). Click **New notebook**. Upload 2–3 PDFs you have lying around — a product manual, a research paper, a long article, a textbook chapter, a company handbook. If you have nothing handy, grab any free PDF from arxiv.org or a public-domain book from gutenberg.org.

Wait ~30 seconds for NotebookLM to ingest the documents. Then ask the chat panel two questions:

1. *"What are the three most important points in these documents?"*
2. A second, more specific question that **requires reading the documents** — e.g. *"What does section 3 of document 2 say about X?", "Which document mentions Y, and what does it claim?", or "Summarise the methodology paragraph on page 5 of the paper."*

Look closely at each answer. You'll see small numbered **citations** next to most claims, each linking back to the specific source passage. Click one and watch it jump you to the exact paragraph the model is drawing from.

Self-check — answer all three:

- **(a)** Where did the answer come from? Click two or three of the citations and note that each one points to a specific page in a specific document. *This is RAG made visible* — the model didn't pull the answer from its training weights; it pulled it from the chunks of your PDFs that the retriever fetched in response to your question.
- **(b)** Now ask a question whose answer is **not** in your uploaded documents (e.g. *"What will the FTSE close at next Friday?"* or *"What is the capital of a country none of these documents mention?"*). Notice what happens — does the model decline, hedge ("the sources don't say"), or hallucinate confidently? Compare this to how a plain chatbot would have handled the same question.
- **(c)** Write one sentence on why citations matter when an LLM is reading your private documents. (Hint: the answer is not "they look nice". It's about auditability — the ability for you, or a colleague, or a regulator, to check exactly which sentence in which document the model based its claim on.)

What you just saw

This is the same pattern almost every "chat with your docs" enterprise product is built on: chunk the documents, embed the chunks, retrieve the

most relevant ones at query time, and ask the LLM to answer *using those chunks as context*. The product changes; the recipe doesn't.

Task 3 — Watch an agent do a task

Open claude.ai (free or Pro account). The point of this task is to feel the difference between *a chatbot replies to you* and *an agent works on a task on your behalf*. Pick one of the two variants below:

Variant A — Lisbon trip planner (no engineering setup needed). Ask Claude:

"Plan a 3-day trip to Lisbon for a couple in October 2026, with a budget of €1500 (excluding flights). Include morning / afternoon / evening for each day, mix of culture and food, with one specific restaurant recommendation per evening. Return the result as a markdown table."

Watch how it works: it doesn't just dump an answer — on a multi-part request like this you'll typically see it think through the structure, list the days, then fill in the cells.

Variant B — coding agent (if you already have Cursor / Claude Code / Copilot installed). Open any small public GitHub repo locally and ask the assistant: "Add a [README.md](#) summarising what this repo does in 5 bullets." Watch the assistant read multiple files, propose a draft, and offer to write the file — that's an agent loop in miniature: read, reason, act.

Self-check — answer all three:

- **(a)** Did the model make any factual claims you'd want to verify before acting on them — a specific restaurant name, an opening time, a price, a quoted fact from the repo's source code? This is the **confabulation** point from the theory: even capable agents still need a human reviewing the output before it touches reality.
- **(b)** How many distinct steps did the model take to complete the request? In agentic tools you can often see the step list (Variant B will literally show file

reads and edits; Variant A will show a reasoning outline). Compare that to a single-shot chatbot reply — an agent decomposes the problem and works through it.

- **(c)** Write one sentence on the difference between *"a chatbot replies"* and *"an agent works on a task"*. The shorthand: a chatbot answers your message; an agent runs a small plan against an environment (files, web, APIs, your screen) and reports back.

Don't actually fly to Lisbon

For the purposes of this exercise you do **not** need to book a trip or merge any code. The point is the *experience* of watching a model decompose and execute a multi-step task, not the artefact it produces. Resist the urge to spend the next hour iterating the itinerary — close the tab when you can answer (a)–(c).

What "done" looks like

You're done when, in your notebook, you have:

- three screenshots of Q1 answers and short notes on style / cutoff / recent-news behaviour from Task 1 (chat assistants)
- two citation observations and one sentence on why citations matter from Task 2 (RAG)
- one verification-worthy claim, a rough step count, and one sentence distinguishing chatbot from agent from Task 3 (agents)

The aim was never to make you pick a favourite chatbot or to ship an agent in 45 minutes. It was to make sure that next time you read the words *"LLM"*, *"RAG"*, or *"agent"* in a product announcement or a job description, you have a concrete picture in your head of each — because you've now sat in front of all three and watched them behave.

Where this goes next

Module 4 ends here. The skills you just exercised — comparing model outputs, noticing what's grounded versus hallucinated, watching an agent step through a task — are the daily skills of working with LLMs in 2026. The technology will keep moving fast; the habit of *poking at* a system instead of trusting it is what stays useful.

Lesson 14: IT Roles & Open Source

Lesson 14 — IT Roles & Open Source

Who does what; the open-source ecosystem; how to enter; which course to take next.

IT Landscape

Module 5

Course Finale

Roles are clouds, not boxes

"In real teams, titles overlap. A 'Full-stack Engineer' at a 5-person startup does what a 'Frontend Engineer' + 'Backend Engineer' + 'DevOps Engineer' + 'QA Engineer' do at a 500-person company."

- Same person, **different title** depending on company size and culture.
 - Hold the titles loosely — *use them to know what to search for*, not to box yourself in.
-

The engineering job-family map

Build the product

- **Frontend**
- **Backend**
- **Full-stack**
- **Mobile** (iOS / Android)
- **Embedded** (firmware)
- **Game** (Unity, Unreal)

Keep it running

- **DevOps Engineer**
- **Site Reliability Engineer (SRE)**
- **Platform Engineer**
- **Cloud Engineer**
- **Security Engineer**

As of 2026 — titles drift; the work is what matters.

The data and AI job-family map (as of 2026)

- **Data Engineer** — pipes data around (Airflow, dbt, Spark, Kafka).
- **Data Analyst** — asks questions of data (SQL, dashboards).
- **Data Scientist** — builds statistical / ML models in notebooks.
- **ML Engineer** — puts models into production.
- **AI Engineer** — *new* — builds with LLMs, RAG, and agents.
- **Research Scientist** — academia-adjacent.

The boundaries between these five shifted noticeably in 2024–2026 as LLMs landed. Refresh annually.

The “around engineering” roles

- **Product Manager** — what to build, for whom, why.
- **Designer** — UX, UI, research.
- **Engineering Manager** — people + delivery.
- **Tech Lead** — people-light, technical-direction-heavy.
- **Solutions / Sales Engineer** — technical, customer-facing.
- **Developer Advocate** — community-and-content-facing.
- **Technical Writer · QA / Test Engineer.**

Many people land in IT through these doors without ever writing production code.

Security as its own world

- **Defensive** (Blue Team) — SOC analyst, threat hunter, incident responder.
- **Offensive** (Red Team) — penetration tester, security researcher.
- **Governance, Risk & Compliance (GRC)** — risk, compliance, audit.
- **AppSec / Cloud Security / OT Security** — specializations.

Increasingly its **own career ladder**, not a stop on the engineering one.

*See sibling courses on this platform: **CTF Blue Team** and **CTF Red Team**.*

Open source — what it actually is

"Code anyone can read, anyone can copy, anyone can change. Usually for free."

- Most of the software you use is **built on it**.
 - Every cloud, every browser, every database you've met in this course sits on a tower of open-source pieces.
-

Why open source exists (and why it works)

- **Dogfooding** — maintainers volunteer because they need the tool too.
- **Corporate backing** — many are paid by their employers to work on tools their employers use: *Linux, Kubernetes, Postgres, Python, the LLVM compiler* — all overwhelmingly maintained by paid engineers at large companies.
- **Foundations** — Linux Foundation, Apache, CNCF, Python Software Foundation, Mozilla.

Money + identity + meritocracy + accumulated trust.

Open-source licenses (the 30-second version)

- **MIT / Apache** — very permissive; use however you want, even closed-source.
- **GPL** — you must share changes back if you redistribute.
- **AGPL** — same, even if you only use it as a network service.
- **BSL / SSPL** — newer “source-available”; *not strictly open source*; vendor protection.

~95% of new projects in 2026 use MIT or Apache.

How to “enter” IT without a CS degree

1. Pick **one role** from slides 3–6 that looks interesting.
2. Read its **job-ad language** until it stops feeling foreign.
3. Ship **something tiny in public** — a GitHub project, a blog, a tool, a Substack.
4. Contribute a **one-line fix** to a small open-source project.

The “ship in public” step matters more than any certification.

Which course on *this* platform next?

If you liked...

- Module 2 / building things
- Module 2 + building tools FOR AI
- Module 4 / agents specifically
- Module 3 / security, defending
- Module 3 / security, attacking
- Module 1 / networking, infrastructure
- Module 1 / cloud and ops
- AI ops & safety
- Password cracking / hashing

...take next:

- **Claude & LLMs** (33 lessons)
 - **MCP Servers**
 - **Building Your Own Agents**
 - **CTF Blue Team**
 - **CTF Red Team**
 - **Linux Networking**
 - **Helm Charts** (20 lessons)
 - **AI OpSec + AI Prompt Security**
 - **Hashcat** (10 lessons)
-

The course, in three bullets

- You now have a **map of IT** (*L01*) — frontend / backend / data / infra / security all on one page in your head.
 - You've seen what the **tools actually look like** (*L05*) — IDEs, GitHub, AI assistants — and know the words on a job description aren't magic.
 - You've watched **AI become a real seat at the table** (*L11*) — and you know which sibling course to take to actually build with it.
-

The end — and the start.

"This was a survey course. The map is in your head now. The next step is to go deep on one square of it. Pick a course from slide 11. Or go talk to someone who works in IT — you can keep up now."

Course complete. Continue with the recommended sibling course of your choice — all live in this same repo.

IT Roles & Open Source

Reading time: ~10 min · Prereqs: Lessons 01–11 · *Snapshot 2026 — role titles drift, especially around AI engineering; the open-source half is much more stable.*

Why this matters to you

You've now seen most of the surface area of modern IT. The last question is the one this course was secretly building toward all along: *where in IT do you actually want to land?* This lesson is two halves that answer that. The first half is **roles** — who actually does what, in what kind of team, with what kind of background; the map of jobs that hangs over the map of technology you've been building since **Lesson 01 (the IT map)**. The second half is **open source** — the cultural and economic substrate underneath almost every job, and the most common on-ramp into a paying IT career. And then, at the very end of this lesson and of this whole course, an explicit recommendation: *which course on this same learning platform you should take next*, based on which lessons in this course made your eyebrows go up.

1. Roles are clouds, not boxes

The job titles you see on LinkedIn — "Frontend Engineer", "Backend Engineer", "DevOps Engineer", "Site Reliability Engineer", "ML Engineer", "Platform Engineer" — sound like crisp categories. Inside any real team, they aren't. At a five-person startup, one engineer does the work that, at a 5,000-person company, would be split across fifteen specialized titles. The same person carries different job titles at different points in their career without actually changing what they do day to day. The technologies you saw in **Lesson 05 (the tools)** — IDEs, GitHub, AI assistants, terminals — sit underneath all of these titles equally. So when you read this lesson's role names, hold them loosely. **Use them to know what to search for and what conversations to have**, not to box yourself in. The cluster matters more than the exact noun on the business card.

2. The engineering job families

The biggest cluster is **software engineering: frontend** (what runs in the browser — React, Vue, TypeScript), **backend** (what runs in the data center — Python/Django, Node, Go, Java/Spring), **full-stack** (both), **mobile** (iOS / Android), **embedded** (firmware in your car, your headphones, your fridge), and **game** (Unity, Unreal). This is the cluster most people picture when they hear "programmer".

The second cluster is **infrastructure and operations: DevOps Engineer, Site Reliability Engineer (SRE), Platform Engineer, Cloud Engineer** — the people who keep the software actually running, deploy new code safely, own the on-call rotation, and build the internal developer-tooling that the first cluster ships on top of. The boundary between "DevOps" and "SRE" and "Platform" shifted noticeably in the late 2020s; *as of 2026*, treat them as three names for overlapping work, with SRE being the most reliability-and-error-budget focused, and Platform being the most internal-developer-tooling focused. The third cluster — security — is now substantial enough to deserve its own section.

3. Data and AI roles (as of 2026)

Five roles, with shifting boundaries:

- **Data Engineer** — builds the pipes (Airflow, dbt, Spark, Kafka) that move data from where it's produced to where it's queried.
- **Data Analyst** — asks questions of the data (SQL, dashboards in Looker / Tableau / Metabase), often embedded in a business team.
- **Data Scientist** — builds statistical and ML models on top of the cleaned data, usually in notebooks.
- **ML Engineer** — takes a data scientist's notebook and turns it into a production service.
- **AI Engineer** — a newer title (broadly adopted starting in 2023) for someone who builds *with* LLMs, RAG, and agents — closer to a software engineer with deep prompt and orchestration craft than to a classical ML researcher. This is the role most directly born of **Lesson 13 (the LLM & agent era)**.

Boundaries here moved meaningfully between 2022 and 2026 as LLMs reframed what "doing ML" means in industry. Treat any role-list this specific as a 12-month snapshot, not a permanent taxonomy.

4. Security as its own world

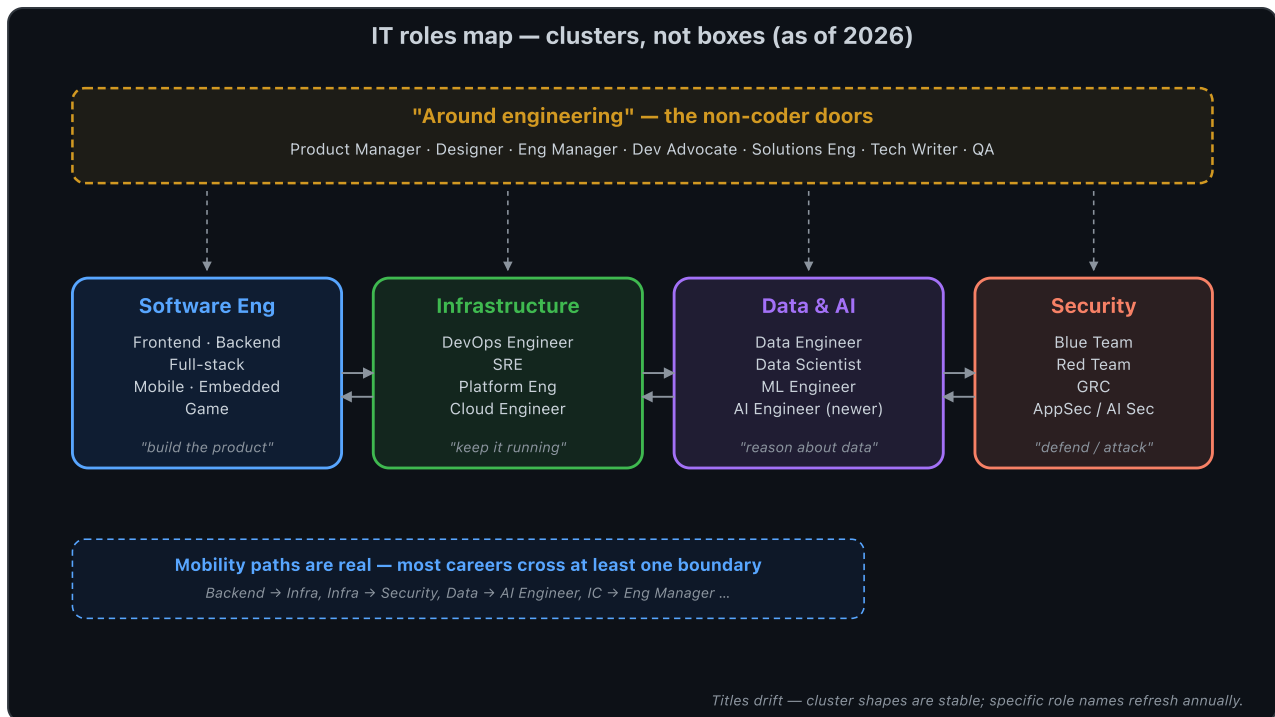
Defensive (Blue Team) — SOC analysts, threat hunters, incident responders, detection engineers; the people watching the logs at 3 AM and writing the playbooks. **Offensive (Red Team)** — penetration testers, security researchers, exploit developers; the people *being* the simulated attacker so the defenders can practice. **Governance, Risk, Compliance (GRC)** — the audits, the policies, the SOC 2 reports, the regulator-facing work; often the door into security from a non-engineering background. And then specializations: **AppSec** (security of the application code), **Cloud Security**, **OT / ICS Security** (industrial systems), **AI Security** (model, prompt, supply-chain — born of the world introduced in Lesson 13). Security is now its own career ladder, not just a senior-engineer rotation.

5. The non-coder doors into IT

Many of the highest-paid and most-influential people in IT do not write production code. **Product Manager** — what to build, for whom, why; sits between engineering and the rest of the business. **Designer** — UX, UI, research; the human face of the product. **Engineering Manager** — people + delivery; "1:1s and unblock people" by the hour. **Solutions / Sales Engineer** — technical, customer-facing, helps prospects evaluate the product. **Developer Advocate** — community, content, conferences; reaches developers where they are. **Technical Writer** — the documentation, the API reference, the changelog; rare and very valued. **QA / Test Engineer** — designs the way the product gets stressed. These doors don't require a computer-science degree, but they do require the vocabulary you've built across this course.

6. The roles map, one diagram

Here is the same idea drawn as clusters arranged spatially. The thin lines suggest common career-mobility paths — people move between adjacent clusters more often than the org chart implies.



7. Open source — what it actually is, and why it works

Open-source software is code anyone can read, copy, and change — usually for free. Almost every program you run, every website you visit, every cloud service you use, sits on a tower of open-source pieces underneath. Linux runs most of the internet. Python, JavaScript, and Go are open. The web browser you're reading this in is largely open. Postgres, Redis, Kubernetes, Git itself — all open.

Why does it work, when nobody is paid by you for it? Three reasons. **Dogfooding** — the maintainers needed the tool too, and built it for themselves; you benefit as a side effect. **Corporate backing** — most "open source" you depend on (the Linux kernel, Kubernetes, Postgres, the V8 JavaScript engine, the LLVM compiler, Python itself) is overwhelmingly maintained by engineers paid by large companies who depend on those projects. **Foundations** — the Linux Foundation, Apache, CNCF, the Python Software Foundation, Mozilla act as legal-and-fundraising homes for projects too important to belong to any one vendor. Licenses matter too: **MIT** and **Apache 2.0** let you do almost anything (including build a closed-source product on top); **GPL** requires you to share your changes back if you redistribute. Most new projects *as of 2026* choose MIT or Apache.

What people argue about

- **Is "open source" being captured by VC-backed companies?** Several influential projects (HashiCorp's stack, Redis, MongoDB, Elasticsearch) shifted their licenses from true open source to "source-available" (BSL, SSPL) in the past five years, to prevent cloud providers from monetizing the project without contributing back. Critics call this a betrayal of the original promise; defenders call it pragmatic survival. The fight is far from settled.
- **Will AI assistants reduce the number of engineers needed?** Three years into the LLM era, the evidence is "engineers are faster, but headcount is not falling and ambition per engineer is rising". Predictions for the next 3–5 years range across the full spectrum. Pick a camp at your peril.
- **CS degree vs bootcamp vs self-taught?** All three still produce hireable engineers in 2026. Increasingly the deciding factor is what you've shipped in public, not which door you came through.

Reading on — the course you take next, on this same platform

This section is unlike any other "Reading on" in this course. Instead of pointing outward, it points sideways — at the other courses sitting in this same repo. Pick one based on which lesson in this course made your eyebrows go up.

- **If Module 4 (the AI era) is the part that made you lean in:** take the **Claude & LLMs** course (33 lessons — the longest course on this platform; the depth answer to Lesson 13). Then, if you want to build tools that LLMs use, take **MCP Servers** (focused on the Model Context Protocol — how Claude and other LLMs talk to your tools). If you want to build LLM-driven agents end-to-end, take **Building Your Own Agents**.
- **If Module 3 (security) was your favorite:** take **CTF Blue Team** (defender exercises in a Dockerized lab) or **CTF Red Team** (offensive techniques in the same kind of lab). For AI-specific security — model, prompt, supply-chain — take **AI OpSec** and **AI Prompt Security**. For password-cracking and crypto practice, take **Hashcat** (10 lessons on practical password cracking).
- **If Module 1 + Module 3 (infrastructure, networking, deployment) felt at home:** take **Linux Networking** (deep dive on routing, namespaces, policy-based routing)

and **Helm Charts** (20 lessons on Kubernetes packaging — the next step beyond the cloud and DevOps lessons of this course).

- **External, for context, not next-step: Roadmap.sh** publishes role-specific learning paths (frontend, backend, DevOps, AI engineer) that go alongside whichever sibling course you choose; **The Pragmatic Engineer** newsletter (Gergely Orosz) is the best read on the industry-trend side — salaries, layoffs, hiring climate, what the senior engineers are actually arguing about.

The course in three sentences

You now have a **map of IT** (Lesson 01) — frontend, backend, data, infra, security all on one page in your head. You've seen **what the tools actually look like** (Lesson 05) — IDEs, GitHub, AI assistants — and you know the words on a job description aren't magic. And you've watched **AI become a real seat at the table** (Lesson 13) — and you know which sibling course on this platform to take to actually build with it. This was a survey course; the map is in your head now. The next step is to go deep on one square of it. Pick a course from the list above. Or go talk to someone who works in IT — you can keep up now.

LAST-REVIEWED: 2026-05-26

Homework — IT Roles & Open Source

Estimated time: ~60 min · No code you have to run · A browser, a job-board account, and a text editor for the finale

Three exercises to close the course. The first two are *tool tours* — you go look at the actual open-source world and the actual IT job market, with the vocabulary you've built over the last eleven lessons. The third is the one and only **reflection task** of the whole course: a 200-word IT autobiography that asks you to name what surprised you, where you want to land, and what you'll learn next. Save that one. Re-read it in six months.

Before you start

Keep a one-page notebook open (paper or a text file). For the two tool tours, write the short answers under (a)/(b)/(c)/(d) per listing — you're not turning this in, the goal is to *look at the real thing* with the course's vocabulary in your head. For the autobiography, open whatever you actually write in (Notes, Obsidian, Google Doc, paper). There is **no code you need to run**. Free accounts on GitHub and one job board are sufficient as of 2026; the URLs and UI elements named below were correct as of the course's last review, so if a button has moved by the time you read this, pick the nearest equivalent.

The three tasks

Task 1 — Tour the open-source ecosystem (OSS shopping)

Open github.com/trending in your browser. At the top of the page is a **language filter**. Try it three times in a row — switch to **Python**, then **TypeScript**, then **Rust**. For each language, scroll the top 5 trending repos and read each repo's one-line description (the small grey text just under the repo name).

Then open opensource.guide/how-to-contribute/ in a second tab and read the section titled "*Finding a project to contribute to*". It's short — under five minutes.

Finally, pick any one of the trending repos you just looked at, click into it, and find the **Insights → Contributors** page (top-right tab on the repo, then *Contributors* in the left side-bar). Take a screenshot of the chart. Notice the **long-tail distribution**: a few people commit a lot, many people commit a little. That's the shape of almost every successful open-source project — and one of the points the theory page makes about how OSS actually works.

Self-check — answer all three:

- **(a)** Of the three languages (Python, TypeScript, Rust), which one has the most "*AI / LLM / agent*" projects in its top 5 today? (Likely Python — but check. The answer can shift week to week, and noticing the shift is the point.)
- **(b)** Pick one trending repo whose purpose you *completely understood* from the one-line description alone, and one whose purpose you *didn't*. Write down both names. The first one is evidence that the vocabulary from this course is now in your head. The second one is a pointer at where to look next.
- **(c)** For the repo whose purpose you didn't understand, click in and read the README for five minutes. Does it become clear? If not, that's fine — not every project explains itself well, and reading READMEs critically is itself a skill. Write one sentence on what the project does, in your own words.

What you just saw

The trending page is a daily snapshot of what the open-source world is currently excited about. The contributor distribution is the same shape regardless of project size — a small core, a wider ring of regulars, and a long tail of one-time contributors. If you ever want to contribute to OSS yourself, the one-time-contribution slot is exactly where you'd start.

✓ You have now seen what the open-source ecosystem looks like from the inside — not as a slogan, but as a live page of real projects with real contributor graphs.

Task 2 — Tour the IT job market through the lens of this course

Open linkedin.com/jobs (or indeed.com, or welcometothejungle.com if you're in Europe). For each of the following **5 job titles**, search and open *one* real listing — whichever one happens to look representative, you're not job-hunting, you're sampling:

1. **Frontend Engineer**
2. **Site Reliability Engineer (SRE)**
3. **Data Engineer**
4. **AI Engineer**
5. **Security Engineer**

For each of the 5 listings, work through the four short prompts below. You will end up with one short table of notes — that table is your second map of the IT landscape, overlaying the one you started this course with in Lesson 01.

Self-check — answer all four per listing:

- **(a)** Write down the **3 most-named technologies** in the listing's *Requirements* or *Must-haves* section. Names only — no commentary.
- **(b)** For each of those three, write down **which lesson of this course** you saw it in. Examples to calibrate: "React" → L04; "Kubernetes" → L08; "Postgres" → L07; "LLM / RAG / vector database" → L11; "SIEM / EDR / SOC" → L09. If a named technology didn't appear anywhere in this course, write "*new*" next to it — that's normal and useful information.
- **(c)** Find **one bullet point in the listing that you didn't recognize at all**. Copy it verbatim into your notes.
- **(d)** Paste that bullet into Claude or ChatGPT with the prompt: "*Explain this to someone who's just finished a 12-lesson IT survey course.*" Read the explanation. Notice that you can now *understand* the explanation — even though you didn't recognize the bullet cold — because the vocabulary that the explanation uses is the vocabulary you spent twelve lessons building. That compounding is the actual point of this exercise.

One listing per title — resist the rabbit hole

It's tempting to open ten listings per role to "*compare*". Don't. One representative listing per title is enough for the mapping exercise; ten will swamp you without changing the conclusion. If you find yourself wanting to bookmark roles for later, open a separate *future_me* note — but finish the homework first.

✓ You have now read 5 real job listings spanning 5 distinct IT clusters, mapped their requirements back to specific lessons of this course, and used an LLM to translate one unfamiliar bullet per listing. The map of IT in your head is now anchored to actual employer language, not just curriculum.

Task 3 — Write your 200-word IT autobiography (the course finale)

This is the course's only reflection task, and it sits here at the very end on purpose. Open a plain text editor, a Notes app, a Google Doc, or paper — whatever you actually write in. Write approximately **200 words** answering, in this order, the three short paragraphs below.

1. **One paragraph — which 2–3 lessons in this course surprised you?** For example: maybe you had no idea the cloud was that *physical* (L06); maybe you had no idea databases mattered as much as they do (L07); maybe you didn't realize *AI Engineer* and *ML Engineer* are actually different roles (L11 / this lesson); maybe you found yourself reading the security lesson three times (L09). The word "*surprised*" is doing the work here — the goal isn't "*which was your favourite*", it's "*which made you sit up*". Surprise is what your interest looks like in real time.
2. **One paragraph — based on those surprises, where in IT do you most want to land?** Use the role clusters from this lesson's theory: *engineering*, *infrastructure*, *data-and-AI*, *security*, or one of the *non-coder doors* (Product Manager, Designer, QA / SDET, Tech Writer, DevRel, IT Support, Project Manager, Tech Recruiter). Be honest — "*I want to land in a non-coder door*" is a perfectly normal and well-paid answer, and it's the one this course works hardest to make visible.

3. **One paragraph — which sibling course on this platform will you take next, and *when*?** Pick one from slide 11 of this lesson, or from the theory page's "Reading on" list — the candidates are *Claude & LLMs*, *Building MCP Servers*, *Building Your Own Agents*, *hashcat*, *helm*, and *linux-net*. Give it a date. Put the date in your calendar before you close this page.

Self-check: the 200 words are written; they name at least **2 specific lessons** of this course; they pick at least **1 sibling course** by name; and there is a **concrete date** for when you'll start it (and that date is in your actual calendar, not just on the page). This is the autobiography. *There is no answer key* — reflection tasks deliberately don't have one. Save the file somewhere you'll find it again, and re-read it in 6 months to see which parts came true.

Why we end the course with this

Every other task in every other lesson had a tool tour or a lab to anchor it. This one doesn't, on purpose. Twelve lessons of vocabulary only become *your* vocabulary once you've said, out loud and in writing, which parts of it grabbed you. The 200 words you write today are a small artefact that converts "*a course you took*" into "*a direction you chose*" — which is the whole reason this course exists.

What "done" looks like

You're done with this homework — and with the course — when, in your notebook, you have:

- a screenshot of one trending repo's *Insights* → *Contributors* graph and short notes (a)–(c) per language sampled from Task 1 (OSS tour)
- a small table of 5 job listings with (a)–(d) filled in per listing from Task 2 (role tour)
- a saved file of approximately 200 words from Task 3 — the autobiography — with at least 2 lesson references, 1 named sibling course, and a date on your calendar

Where this goes next

Module 5 ends here, and so does the IT Landscape 2026 course. The skills you just exercised — reading real OSS projects with the right vocabulary, reading real job listings with the right vocabulary, and naming out loud what you actually want next — are the daily skills of *navigating* IT rather than being intimidated by it. The technology will keep moving fast; the habit of looking at the real thing and the habit of choosing your own next step are what stay useful. See you on whichever sibling course you put on your calendar.