ADDALY · THE AI CLASSROOM

Shipping It

Get what you built in front of real people, and keep it alive.

---

8 modules · 74 lessons · 29 figures · 8 case studies · 56,931 words · about 11 hours

Dr Mustafa Mun
Author and editor

ABOUT THIS BOOK

Shipping It, published by Addaly, 2026. Author and editor: **Dr Mustafa Mun**.

The manuscript was drafted with the assistance of a large language model and was edited and published under his name. That is stated plainly here rather than left to be discovered, because a reader is entitled to know how a book they are trusting was made.

This book is the printed form of the course of the same name at addaly.com/learn/cloud-and-shipping. The website is the living version and is corrected first; where the two differ, the website is right.

Free to read, free to download, free to print, and free to teach from. There is no paid edition and there never will be. If you paid for this, somebody has sold you something that was given away.

Every diagram in it is drawn from data by code, not generated as a picture, so the numbers on the page are the numbers in the argument. Answers to every exercise, test and examination question are at the back, with a note on why each wrong option attracts people — those notes are the teaching, not the marking.

Contents

2. Making a deploy boring

Ship a code change and a schema change to a live site without dropping a request, and undo either one in under two minutes

1. The key you must not commit
2. The same build, everywhere
3. Which commit is live right now
4. The checks that run without you
5. The console clicks nobody wrote down
6. Deploying for nearly nothing
7. Ending a process without dropping a request
8. The copy you are allowed to break
9. Changing the schema under a running site
10. Undo, in under two minutes
11. Case study: Three deploys, or one, before the clinic opens
12. Module test

3. Knowing what it is doing

Answer "is it broken, for whom, and since when" in under five minutes using data you are already collecting

1. Reading logs at 2am
2. Where the logs go, and what they cost
3. Four numbers that describe any service
4. One screen that answers the question
5. Errors that come to you
6. The health check that lies
7. Alerts you would get out of bed for
8. Deciding how reliable it has to be
9. Finding the slow bit
10. Case study: Four days of green
11. Module test

4. Load, cost and the traffic that is not customers

Find your first real ceiling with a load test and raise it using caching, pooling or a queue, without raising the bill

1. The bill, and the traffic that runs it up
2. The request you never serve
3. Serving from near the user, and what bandwidth costs
4. Finding your ceiling before your users do
5. The day you run two copies
6. It is almost always the database
7. Work that does not belong in a request
8. The traffic that is not customers
9. Files people send you
10. Why your email went to spam
11. Case study: The bill from an endpoint nobody was told about
12. Module test

5. Staying alive: the bad day and the day after

Run the first thirty minutes of a real outage from a written runbook, and leave behind a postmortem and a handover that change what happens next time

1. The restore you have never tested
2. The short list that actually matters
3. Data you must keep, and data you must delete
4. Everything expires, and nothing tells you
5. Timeouts, retries, and the failure you made worse
6. The outage that is not yours
7. The first thirty minutes of an outage
8. Writing down what happened
9. Breaking it on purpose, on a Tuesday
10. Writing it down for whoever is next
11. Case study: Roll back, and lose twenty-five minutes of bookings
12. Module test

6. Shipping a model, not just a web app

Put a model behind an HTTP endpoint with a measured latency budget and a stated cost per request, ship a new version by shadow then canary rather than by hope, and detect the week its inputs stop resembling the data it was trained on

1. A prediction endpoint is a web service with a large secret
2. Where inference time actually goes
3. What a prediction costs, worked out
4. Which model, which prompt, which version
5. The feature computed two different ways
6. Shadow, then canary, then everybody
7. The model that quietly stopped fitting
8. The model that trains on its own decisions
9. What the page says when the model is not there
10. Case study: Point eight nine offline, a coin toss in the office
11. Module test

7. Security after the short list

Enforce authorisation per object rather than per page, revoke a session or rotate a key without an outage, and run the first day of a breach or an abuse report with evidence preserved and users told the truth

1. Authentication is not authorisation
2. Sessions, and taking one back
3. The code you did not write
4. Changing a key while everything is using it
5. Bots, scrapers and the flood you cannot absorb
6. Taking money without holding card numbers
7. The tools that can do anything
8. The first day of a security incident
9. The email about one of your users
10. Case study: Receipt number 4412
11. Module test

8. Running it for years

Decide what to run yourself and what to buy using a cost per user and a maintenance budget you have actually calculated, keep a system upgraded and legally presentable over several years, and move it to another provider without losing a URL

1. What to run yourself, and what to pay for
2. The upgrade you keep postponing
3. What one user costs you
4. Moving to another provider without losing a URL
5. The hour a week that keeps it alive
6. Removing a feature, and retiring a site
7. The pages every public site needs
8. Keeping it going without burning out
9. Case study: The volunteer left, and the free tier ended
10. Module test

Shipping It — final examination

The paper that opens the next course. Answers to everything follow it.

MODULE 1

How the internet reaches your process

Before you can fix a live site you need a picture of what happens between somebody typing your name and your code running. Six layers, each with its own way of failing. This comes first because every later module assumes you can say which layer you are looking at.

> BY THE END OF THIS MODULE YOU CAN
>
> Trace one request from a typed URL to your running process and back, and name which layer is at fault when it fails

---

1 · 7 min

What a server actually is

> THE ONE THING TO KEEP
>
> A server is just a process holding a port open; the cloud only changes who owns the machine.

A server is a process, not a building

Run this on your laptop:

```
$ node server.js
Listening on http://localhost:3000
```

You are now running a server. A program is holding a TCP port open and answering whatever arrives on it. That is the whole definition. Nothing about a rack, a cooled room in Mumbai, or a company with a blue logo is part of it.

What your laptop lacks is not server-ness. It lacks three things: a public address (your router gave you a private one like 192.168.1.7), a certificate so browsers trust it, and the willingness to stay awake when you close the lid. Deploying is buying those three things from someone else.

What you are actually renting

There is a ladder, and the rungs differ in how much of the machine you are responsible for.

- **Bare metal.** A physical computer in a building. You get all of it, you patch the kernel, you care when a disk dies.
- **Virtual machine.** A slice of that computer, with its own Linux, its own IP, root access. A DigitalOcean droplet, an EC2 instance, a Hetzner box. Small ones run about $4–7 a month.
- **Container platform.** You hand over an image, the platform runs copies of it and restarts them when they die. Fly, Render, Cloud Run.
- **Serverless.** You hand over a function. It runs when a request arrives and stops afterwards. Lambda, Cloudflare Workers, Vercel functions.

"The cloud" means only this: someone else owns the machine, the building, the power and the network, and bills you by time or by request. It is a rental agreement, not a technology.

The axis that actually changes your code

Forget price for a moment. The question that changes how you write things is: **is a process always running?**

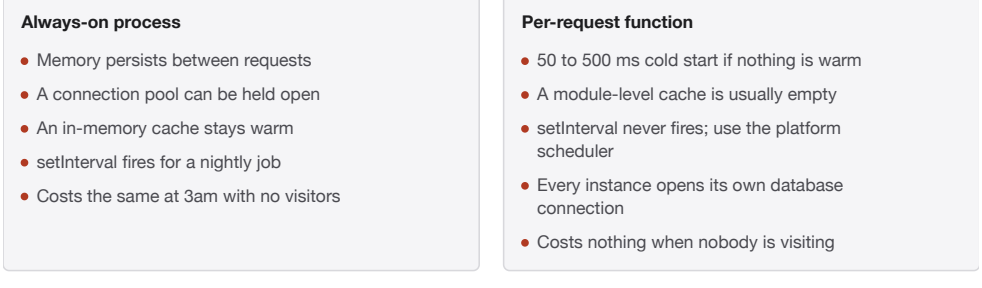
Always-on, you get a process with memory that persists. You can hold a database connection pool. You can keep an in-memory cache. You can start a timer that fires every ten minutes. You pay the same at 3am with zero visitors.

Per-request, a process starts when a request arrives — 50 to 500ms of cold start if none is warm — handles it, and may be gone before the next one. Consequences people discover the hard way:

- A module-level cache is usually empty, because the next request lands on a different instance.
- `setInterval` for a nightly job never fires. You need the platform's scheduler.
- Every instance opens its own database connection. Postgres defaults to about 100 connections. Fifty concurrent instances plus a pool of five each is 250, and you get `sorry, too many clients already`. This is why serverless projects put a connection pooler in front of the database.

Neither shape is better. They fail differently, and the failures are only surprising if you did not know which one you picked.

Always-on process versus per-request function



Neither shape is better. Each one fails in a way that is only a surprise if you did not know which one you picked.

The machine is in a place

Regions are not marketing. A round trip from Lagos to a Virginia region is roughly 110–180ms. Frankfurt might be 60ms. That is per round trip — if your handler runs ten sequential queries against a database on another continent, you have built a two-second page out of fast code.

The rule that matters more than any other: **put your application in the same region as your database.** Users are far away once. Your database is far away on every query.

So what is a deploy

Get the code onto a machine that has a public address, run it, point a name at it, keep it running. Everything else is convenience.

BEFORE YOU MOVE ON

Your handler stores computed results in a module-level object so repeat requests are fast. Locally, the second request is instant. Deployed to a per-request platform, the cache almost never hits — nearly every request recomputes. What is happening?

A. Requests are spread across instances that start fresh, so the object is usually empty for whichever one answers

B. The platform strips global state from deployed functions as a security measure

C. The cache is working, but network latency is large enough to hide the saved computation

D. The production build removed the cache object because nothing appeared to reference it

---

2 · 9 min

The first ten minutes on a new machine

> THE ONE THING TO KEEP
>
> A fresh machine is scanned within minutes, so keys-only login, a non-root user, a closed firewall and automatic security updates are not hardening you do later — they are the machine's first ten minutes.

A rented box is on the public internet within seconds

The moment a provider hands you an IP address, strangers start knocking. Put a fresh machine online with password login enabled and you will see failed SSH attempts within minutes — usually thousands a day, from botnets

working through `root`, `admin`, `ubuntu`, `postgres` and every password in a leaked list. None of this is personal. It is a script.

So the first ten minutes on a new machine are the same ten minutes every time, and they are worth turning into a checklist you keep.

Keys, not passwords

Generate a key pair on your own machine, not on the server:

```
ssh-keygen -t ed25519 -C "laptop-2026"
ssh-copy-id -i ~/.ssh/id_ed25519.pub user@203.0.113.10
```

Ed25519 keys are short, fast, and the current default. RSA still works if you use 4096 bits, and you will meet it on older systems.

The private key never leaves your laptop. What you copy to the server is the `.pub` half, which lands in `~/.ssh/authorized_keys`. Anyone holding that file can let themselves in, which is why the permissions matter: `700` on `~/.ssh`, `600` on the file. OpenSSH silently refuses keys in a world-readable directory, and this is the single most common reason a key that "should work" does not.

Put a passphrase on the private key. Without one, a stolen laptop is a stolen server. `ssh-agent` means you type it once per session rather than once per connection.

Turn off what you are no longer using

In `/etc/ssh/sshd_config`:

```
PasswordAuthentication no
PermitRootLogin no
```

Then `sudo systemctl reload ssh`. Keep your current session open while you test the new one in a second terminal. Locking yourself out of a machine

you have no console access to is a real way to lose a weekend, and it is entirely avoided by not closing the first window.

Changing the SSH port from 22 is popular advice. It removes most of the log noise and stops none of the serious scanning, which enumerates ports anyway. Treat it as tidying, not security.

Do not work as root

Create a normal user, give it `sudo`, and use it. Root has no undo and no audit trail: every process it starts, every file it writes, every `rm -rf` with a typo in it runs with the whole machine's authority. The habit costs nothing and it is the difference between a mistake and an outage.

Your application should run as its own user too, one with no login shell and no write access to its own code directory. If the process is ever made to run attacker-supplied code, that user's permissions are the blast radius.

Close the ports you are not using

Everything a fresh install starts — a database on 5432, Redis on 6379, an admin panel on 8080 — is reachable from the internet unless something stops it. Databases exposed this way are found and emptied within hours, and there are search engines dedicated to listing them.

```
sudo ufw default deny incoming
sudo ufw allow 22,80,443/tcp
sudo ufw enable
ss -tlnp    # what is actually listening, and on which address
```

Read the `ss` output carefully. `0.0.0.0:5432` means the whole internet; `127.0.0.1:5432` means only this machine. A cloud provider's security group does the same job one layer out, and it is fine to have both.

Patches, applied by something other than your memory

On Debian or Ubuntu:

```
sudo apt install unattended-upgrades
sudo dpkg-reconfigure -plow unattended-upgrades
```

This installs security updates on its own. It will occasionally need a reboot to take effect, which is what `/var/run/reboot-required` is telling you. Unattended upgrades are the highest-value five minutes on this list, because the alternative is a machine that stays on a kernel with a public exploit until you happen to think about it.

Two habits that save you later

- **Write the machine down.** One file in your repository: what it is, who pays for it, what runs on it, which DNS names point at it. A server nobody can name is a server nobody dares turn off.
- **Assume it is disposable.** If rebuilding this box from nothing would take you more than an hour of remembering, the recipe should be in a script, not in your head. That is the whole argument for configuration management, and a shell script committed to the repository already counts.

The free path

All of this is OpenSSH, `ufw` and `systemd`, which ship with every Linux distribution and cost nothing. `mosh` is worth installing if your connection drops on trains. Managed platforms hide this entire lesson from you, which is a genuine advantage — until the day you need to know what a machine is doing, and the machinery underneath is the same either way.

BEFORE YOU MOVE ON

You copy your public key to a new server, then set `PasswordAuthentication no`. Your next SSH attempt is refused with "Permission denied (publickey)", although the same key works on another server. What is the most likely cause?

A. The key is Ed25519 and the server only accepts RSA, so it was never offered
B. The firewall is blocking port 22, so the connection never reaches sshd
C. `PasswordAuthentication no` also disables public-key authentication until `PubkeyAuthentication yes` is added explicitly
D. Permissions on `~/.ssh` or `authorized_keys` are too open, so sshd is ignoring the file

3 · 9 min

What keeps your process running when you close the laptop

THE ONE THING TO KEEP

Something other than your terminal has to own the process: a supervisor that starts it at boot and restarts it on exit, plus the journal that says why it exited — and a process that keeps vanishing without a log line was probably killed by the kernel for using too much memory.

The thing people do first, and why it fails

The first deployment nearly everybody performs is this: SSH in, run `npm start` or `python app.py`, see the site working, close the terminal. An hour later the site is down.

It is down because the process was a child of your login shell. When the connection dropped, it was sent a hangup signal and died. `nohup` and `tmux` fix that particular symptom, and people stop there — which is why so many small

sites are one accidental `Ctrl-C`, one crash, or one reboot away from being offline until somebody notices.

What you actually want is something whose job is to keep your process running: start it at boot, restart it when it exits, and record why it exited. On Linux that is `systemd`, already installed, and a unit file is about twelve lines.

A unit file, in full

`/etc/systemd/system/myapp.service`:

```
[Unit]
Description=myapp
After=network-online.target

[Service]
User=myapp
WorkingDirectory=/srv/myapp
EnvironmentFile=/etc/myapp.env
ExecStart=/usr/bin/node /srv/myapp/server.js
Restart=always
RestartSec=2

[Install]
WantedBy=multi-user.target
```

Then:

```
sudo systemctl daemon-reload
sudo systemctl enable --now myapp
systemctl status myapp
journalctl -u myapp -f
```

`enable` is the part people forget: without it the service does not come back after a reboot. `Restart=always` restarts on any exit, including a clean one. `Restart=on-failure` restarts only on a non-zero exit, which is what you want if the process legitimately finishes.

Anything your program writes to stdout and stderr goes to the journal, time-stamped and searchable, with no logging library involved. `journalctl -u myapp --since "10 min ago"` is the fastest way to answer "what did it say before it died".

The restart loop that hides the bug

`Restart=always` will happily restart a process that dies immediately, forever, several times a second. The site looks broken in a way that keeps changing, and the machine gets busy doing nothing.

systemd has a rate limit for this: by default, five starts within ten seconds and it gives up, leaving the unit `failed`. That is the correct behaviour, and it is worth knowing because "the service will not start and systemctl says failed" often means it started twenty times a minute ago. Tune with `StartLimitIntervalSec` and `StartLimitBurst`.

A restart loop is a symptom, not a fix. The journal has the reason, usually in the last two lines before each restart: a missing environment variable, a port already in use, a database that is not up yet.

The killer nobody warns you about

When a Linux machine runs out of memory, the kernel picks a process and kills it. Your application, being the biggest thing on a small box, is nearly always the one picked. It receives no signal it can catch, writes nothing to its own logs, and simply vanishes.

The evidence lives in the kernel log:

```
dmesg -T | grep -i -E 'killed process|out of memory'
journalctl -k --since today | grep -i oom
```

If you see it, the fix is memory — a smaller heap, fewer workers, a bigger box, or a leak found — not a bigger restart limit. Add swap only as a shock absorber; a machine that is swapping is a machine whose latency has already collapsed.

Containers do the same job with different words

In Docker, `--restart unless-stopped` is `Restart=always` with a different spelling. In Kubernetes, the restart policy plus liveness probes do it, and `CrashLoopBackOff` is the same rate limit as above, expressed as an escalating wait.

One container-specific trap: your process becomes PID 1, and PID 1 does not get the kernel's default signal handling. A process that ignores `SIGTERM` will sit there until the runtime kills it hard, ten seconds later, mid-request. Running with `docker run --init`, or handling `SIGTERM` yourself, avoids it.

The free paths

`systemd` is free and already on the machine. `supervisord` is a good, simple alternative if you find unit files unpleasant. `pm2` is popular in Node projects and does restarts, log rotation and clustering, but note that it is one more thing that must itself be started at boot — usually by systemd, which is a hint about where the real answer lives.

On a managed platform — Fly, Render, Railway, Cloud Run, a Workers runtime — this lesson is done for you, and the platform's restart behaviour is worth reading once so you know what it does with a process that will not start.

BEFORE YOU MOVE ON

Your service disappears every few days. There is nothing in the application log, no stack trace, and no error at all — the log simply stops mid-request and picks up again after a restart. Which explanation fits the evidence best?

A. An unhandled exception is crashing the process before the logger can flush

B. The SSH session that started it was disconnected, sending a hangup signal

C. systemd hit its start rate limit and left the unit in a failed state

D. The kernel's OOM killer terminated the process, which gets no catchable signal and so writes nothing

4 · 8 min

Why your site is not loading

THE ONE THING TO KEEP

Nothing propagates; DNS answers are cached for their TTL, so lower the TTL before you change anything.

The name is a question, not an address

Browsers connect to IP addresses. `addaly.com` is not one. So before any connection happens, a chain of questions runs: your machine asks a resolver (your ISP's, or 1.1.1.1, or 8.8.8.8), which asks a root server, which points at the `.com` servers, which point at the **authoritative nameservers** for your domain, which finally give an answer.

The answer is a record:

- **A** — an IPv4 address. `104.21.53.18`
- **AAAA** — an IPv6 address.
- **CNAME** — "ask about this other name instead".
- **TXT** — arbitrary text, used to prove you own the domain.
- **MX** — where mail goes.
- **NS** — which nameservers are authoritative.

Two things people confuse, constantly

The **registrar** is who you bought the name from — Namecheap, Hostinger, GoDaddy. Roughly $11 or ₹900 a year for a `.com`.

The **nameservers** are who answers questions about it. These are often not the same company. If you moved your domain to Cloudflare, Cloudflare's nameservers answer, and the record editor back at your registrar is a form no-

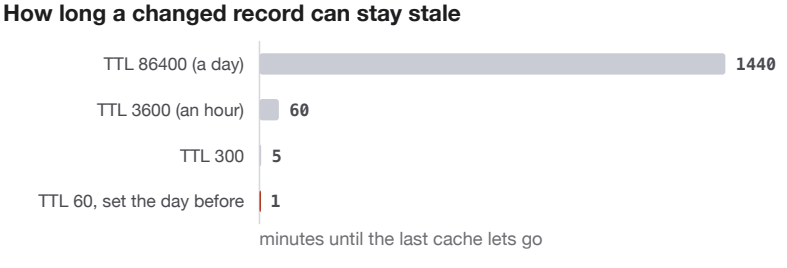
body reads. Editing there and waiting for a change that never comes is the single most common wasted afternoon in this subject.

Nothing propagates

"DNS propagation" is a misleading phrase that has cost people days. Records are not pushed anywhere. They are pulled, and then cached, and every cache holds its copy for the record's **TTL**.

If your A record has a TTL of 3600, a resolver that asked five minutes ago will keep serving the old IP for another 55 minutes, no matter what you do. Your own machine may have asked more recently or less recently than a colleague's, which is why two people see two different sites.

So: **lower the TTL to 60 seconds the day before you plan a change.** Make the change, watch it take a minute instead of a day, then put the TTL back up.



Nothing propagates. A resolver that asked one second before your change keeps the old answer for the whole TTL. Lower it to 60 the day before, change the record, then put it back up.

Ask the question yourself

Stop guessing from a browser. The browser adds its own caching, its own HTTPS upgrade, and its own error pages.

```
$ dig +short addaly.com
$ dig +short addaly.com @1.1.1.1        # what a public re-
solver thinks
$ dig +short addaly.com @ns1.example.com  # ask the authorita-
tive server directly
$ dig NS addaly.com +short              # who is authoritative
at all
```

If authoritative gives the new answer and 1.1.1.1 gives the old one, you are waiting on a TTL. Wait. If authoritative gives the old answer, your change did not save, or you edited a zone nobody consults.

Separate the three failures

When "the site is down", it is one of these, and they need different fixes:

1. **It does not resolve.** `dig` returns nothing. A DNS problem. Check nameservers, then records.
2. **It resolves but does not connect.** You get the IP, the connection hangs or is refused. Firewall, wrong port, nothing listening.
3. **It connects but the certificate is wrong.** The browser shows a security warning, not a 404. Your host has not issued a certificate yet, usually because it is waiting on the DNS to point at it. This is a TLS problem wearing a DNS costume.
4. **Everything works and you get someone else's 404.** DNS is fine, TLS is fine, but the server does not recognise the hostname. You added the domain to DNS and forgot to add it to the application.

The apex quirk

You cannot put a CNAME on the bare domain (`example.com`) next to other records — the specification forbids it. That is why instructions always say "CNAME for `www` , A record for the apex". Providers work around it with ALIAS, ANAME, or CNAME flattening, which look like a CNAME to you and resolve to an A record for everyone else.

BEFORE YOU MOVE ON

You moved hosts and updated the A record an hour ago. You see the new site. A colleague in Manila still sees the old one. From your terminal, `dig @1.1.1.1 yoursite.com` returns the new IP and `dig @8.8.8.8 yoursite.com` returns the old one. What is going on?

A. One resolver still holds the previous answer and will keep serving it until its copy of the TTL expires
B. The update has not reached Asian DNS servers yet; changes spread outward from the registrar over hours
C. Your colleague needs to clear their browser cache — the DNS is already consistent everywhere
D. Two A records exist and traffic is being split between the old and new servers

5 · 7 min

What a status code is telling you

THE ONE THING TO KEEP

The first digit says who has to fix it; a status code that lies makes every layer above you lie too.

A request is just text

Strip away the libraries and an HTTP request is a few lines sent over a socket:

```
GET /rooms/chai HTTP/1.1
Host: addaly.com
Accept: text/html
Cookie: session=abc123
```

And the response:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Cache-Control: public, max-age=60

<!doctype html>...
```

A method, a path, headers, an optional body. A status line, headers, an optional body. Everything else — REST, GraphQL, your framework — is a convention layered on top of that.

The first digit answers "whose problem is this"

This is the part worth internalising.

- **2xx** — it worked.
- **3xx** — look somewhere else.
- **4xx** — the request was wrong. The caller has to change something.
- **5xx** — the request was fine and the server failed. You have to change something.

A graph with two lines, 4xx and 5xx, tells you at a glance whether to open your editor or open a conversation with whoever is calling you. Almost no other single signal is that cheap.

The distinctions that bite

301 versus 302. A 301 is permanent, and browsers cache it aggressively — sometimes until the user clears their profile. Ship a wrong 301 and you cannot take it back from the people who received it. While you are experimenting, use 302 or 307. Switch to 301 when you are certain.

401 versus 403. 401 means "I do not know who you are, send credentials". 403 means "I know exactly who you are, and no". Send 401 to a logged-in user who lacks permission and their client will try to re-authenticate forever, in a loop, against a login they already completed.

429. Too many requests. Pair it with a `Retry-After: 30` header so the caller knows to wait rather than retry immediately and make things worse.

502 versus 500 versus 504. A 500 is your code throwing. A 502 or 504 comes from a proxy in front of your app: 502 means your app answered with garbage or not at all, 504 means it did not answer in time. If you see 502s, your process probably crashed or failed to start. If you see 504s, something in your handler is slower than the proxy's timeout, often 30 or 60 seconds.

Do not lie in the status line

This is common and expensive:

```
HTTP/1.1 200 OK
{ "ok": false, "error": "expired token" }
```

Every layer between you and the user reads the status code, not your JSON. Your uptime monitor reads it. Your CDN reads it and may now cache the failure for a minute. A client library's retry logic reads it and does not retry. Your error dashboard reads it and stays green while people are locked out.

The status code is not decoration on top of the real answer. It is the summary of the outcome, written where the rest of the internet can see it. Say 401 when you mean 401.

GET must not change anything

A `GET /posts/12/delete` link will eventually be followed by a crawler, a link preview bot in a chat app, a browser prefetching what it thinks you will click, or an email scanner opening every URL in a message to check for malware. All of these will delete post 12 with no human involved.

This is not a style preference. GET and HEAD are defined as safe — meaning callers are entitled to assume they cause nothing. Destructive actions go behind POST or DELETE.

BEFORE YOU MOVE ON

An API returns HTTP 200 with `{"ok": false, "error": "expired token"}` when a session expires. During an incident, users were locked out for three hours while the uptime dashboard stayed green. What did that 200 actually cost?

A. Every layer that reads status codes — monitors, caches, retry logic — was told the request succeeded
B. Nothing structural; the monitoring should have been configured to parse the response body
C. The 200 is technically correct, because the HTTP transaction itself succeeded — the failure is at the application layer
D. The real fault is the expired token; the status code choice is a separate style question

6 · 9 min

The padlock, and what it does not prove

THE ONE THING TO KEEP

A certificate proves control of a domain name and nothing else, and the way it kills your site is not being broken but quietly expiring.

What a certificate actually claims

A TLS certificate says one thing: at the moment this was issued, whoever asked for it could prove they controlled this domain name.

That is the whole claim. It does not say the company is real, registered, honest, or the one you meant to visit. A domain-validated certificate is free and takes about thirty seconds, which means anyone who can register `add4ly-login.com` can put a padlock on it. Most phishing pages served today are on HTTPS. The padlock means the connection is private; it says nothing about who is on the other end.

What you get from it is still worth having, and it is not optional. Without TLS, everything is readable and editable in transit — passwords, session cookies, the page itself. Public wifi and hostile ISPs have both injected ads into plain HTTP pages, at scale. Browsers now mark HTTP as not secure, HTTP/2 in practice requires TLS, and a payment provider will not talk to you without it.

Getting one for nothing

Let's Encrypt issues certificates free, through an automated protocol called ACME. Practically every tool speaks it.

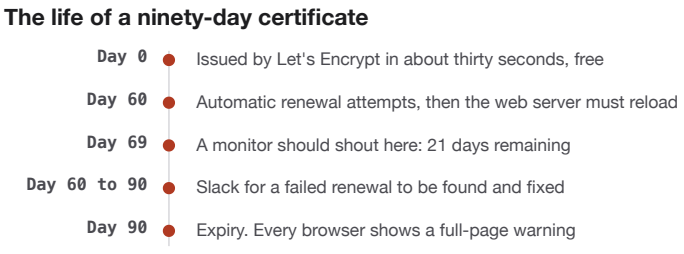
- **Caddy** does it with no configuration at all beyond your domain name. Free.
- **Certbot** does it for nginx and Apache, and writes the config for you. Free.
- Any managed platform — Cloudflare, Vercel, Fly, Netlify, Render — does it for you and you never see it.

Paid DV certificates from a reseller cost roughly $10 to $70 a year and give you the same padlock, plus a warranty that nobody has ever successfully claimed. Extended Validation certificates used to show the company name in the URL bar, which was the one thing you were really buying; Chrome and Firefox stopped displaying it in 2019. There is no remaining reason for a small site to pay.

Ninety days, and the renewal is the outage

Let's Encrypt certificates last 90 days. The convention is to renew at 60, leaving 30 days of slack for a failing cron job to be noticed.

This is where certificates actually take sites down. Not attacks — expiry. And the safety net you may be remembering is gone: Let's Encrypt stopped sending expiry-warning emails in mid-2025. Nothing will tell you. Shorter-lived certificates, measured in days rather than months, are being rolled out across the industry, which makes automatic renewal mandatory rather than merely sensible.



Renewal at day 60 leaves thirty days for a broken cron job to be noticed. Since mid-2025 nothing emails you, so the thing that notices has to be a monitor you set up.

Check any site's dates from a terminal:

```
echo | openssl s_client -connect addaly.com:443 -servername ad-
daly.com 2>/dev/null \
  | openssl x509 -noout -dates -subject
```

Put that in a monitor that shouts at 21 days remaining. It is five minutes of work against the single most common self-inflicted outage in this course.

The two ways of proving you own the name

- **HTTP-01.** The issuer asks for a file at `/.well-known/acme-challenge/<token>` on port 80. Simple, and the default. It needs port 80 reachable from the internet, and it cannot issue wildcards.
- **DNS-01.** You publish a TXT record at `_acme-challenge.example.com`. This works for wildcard certificates, works when the machine is not publicly reachable, and needs API access to your DNS provider.

If your renewal keeps failing, it is nearly always one of these two preconditions quietly changing — a firewall rule that closed port 80, or a DNS API token that expired.

The failures you will actually meet

- **The certificate covers `example.com` but not `www.example.com`.** They are two names. Both must be on the certificate, or the visitor who types `www` gets a full-page security warning.

- **A missing intermediate chain.** Your browser looks fine, because browsers cache intermediate certificates from other sites you have visited. `curl`, Android, and your payment provider's webhook do not. This is why "it works in Chrome" is not a test. Use `openssl s_client` or SSL Labs.
- **The renewal ran but nothing reloaded.** New certificate on disk, old certificate in memory. Your renewal hook must reload the web server, and the cron user must have permission to do it.
- **Rate limits.** Testing your automation against the production issuer will lock you out for a week. Use the staging endpoint while you are getting it working.
- **Clock skew.** A server whose clock is wrong will reject a perfectly good certificate as not yet valid.

HSTS: the header you cannot take back

`Strict-Transport-Security` tells browsers to refuse plain HTTP for your domain for a stated period.

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

That is a year. Every browser that has seen it will refuse to load your site over HTTP, and there is no way to reach those browsers and change your mind. If a subdomain of yours is HTTP-only, `includeSubDomains` takes it offline.

Start at `max-age=300`, confirm nothing breaks, then raise it in steps. The HSTS preload list, which bakes your domain into the browsers themselves, is close to permanent — removal takes months to reach users. Do not submit a domain you are still experimenting with.

Mixed content

One `http://` image URL on an HTTPS page and the browser blocks it silently. Grep your templates and your database content for `http://` before launch, and again after anyone pastes a URL into a CMS field.

BEFORE YOU MOVE ON

Your certificate renews automatically at 60 days and the cron job has been reporting success for months. One morning the site shows an expired-certificate warning to every visitor, and the certificate file on disk has a valid date. What is the most likely cause?

A. The certificate was issued for the apex domain only, so visitors typing www see a name-mismatch warning
B. Let's Encrypt revoked the certificate, which browsers checked before displaying the page
C. The renewal wrote a new certificate but the web server was never reloaded, so it is still serving the old one from memory
D. The clock on the server drifted, so the new certificate is being treated as not yet valid

7 · 9 min

What sits in front of your app

THE ONE THING TO KEEP

A reverse proxy takes the port, the certificate and the client's IP address away from your app, and every one of those has a default that will surprise you.

Your app should not be listening on 443

On Unix, ports below 1024 require root, and you do not want your application code running as root. So the shape almost everybody ends up with is this: your app listens on a high port — 3000, 8000, 8080 — as an ordinary user, and something else listens on 80 and 443 and forwards to it.

That something is a **reverse proxy**. On a virtual machine it is nginx, Caddy or Traefik, all free. On a managed platform it is the platform's own edge, and you never configure it, but it is still there and it still has opinions.

What sits between the browser and your code

Browser	connects to port 443, checks the certificate
Reverse proxy: Caddy, nginx or the platform edge	terminates TLS, serves static files, limits body size, forwards the real IP in a header
Your application on 127.0.0.1:3000	plain HTTP, runs as its own user, trusts X-Forwarded-Proto for redirects
Database on a private address	reachable from the app only, never from the internet

The proxy owns the ports, the certificate and the client's real address. Your app listens on a high port as an ordinary user and only ever hears from the proxy.

What it buys you:

- TLS terminates in one place, so certificates are one component's problem.
- Static files are served without ever waking your runtime.
- Compression, request size limits and connection handling happen before your code.
- Several applications share one machine and one IP.
- You can restart your app without dropping the connection the browser already holds.

`127.0.0.1` versus `0.0.0.0`

`127.0.0.1` means "accept connections that originate on this machine only". `0.0.0.0` means "accept on every network interface".

Bind to `127.0.0.1` when a proxy on the same machine is the only thing that will ever connect. It is a free firewall.

Bind to `0.0.0.0` inside a container. This is the single most common "it works on my laptop and times out in Docker" bug: the container publishes port 3000, the request arrives on the container's network interface, and your server is listening only on the container's own loopback. Nothing is wrong with your code, your networking or the image. One address is wrong.

You have just lost the client's IP address

Every request now arrives from the proxy, so your logs fill with `127.0.0.1`. The proxy is supposed to tell you who it really was, in headers:

```
X-Forwarded-For: 203.0.113.7, 198.51.100.4
X-Forwarded-Proto: https
```

`X-Forwarded-For` is a list, appended to at each hop. Here is the part people get wrong in both directions. It is an ordinary header: anyone can send `X-Forwarded-For: 1.2.3.4` in their first request. Trust only the entry that a proxy **you control** appended — in practice, the rightmost one, not the leftmost. Behind Cloudflare, use `CF-Connecting-IP`, which Cloudflare overwrites rather than appends.

Get this wrong one way and every visitor looks like the same client, so your rate limiter throttles the whole internet as one person. Get it wrong the other way and the limiter is bypassed by typing a fake header.

Most frameworks have a "trust proxy" setting. It is off by default for exactly this reason, and turning it on blindly is how the bypass appears.

The defaults that will bite you

These are nginx's, and every proxy has an equivalent:

- `client_max_body_size` defaults to **1 MB**. A larger upload gets a 413 and your application never sees the request, so there is nothing in your logs to find.
- `proxy_read_timeout` defaults to **60 seconds**. A slow report returns a 504 to the user while your app finishes happily and logs a 200. Two logs disagreeing about the same request is the signature of a proxy timeout.
- **Response buffering is on.** Server-sent events and streamed responses arrive all at once at the end, or never. You need `proxy_buffering off` and often `X-Accel-Buffering: no`.
- **WebSockets need explicit permission** to upgrade — `Upgrade` and `Connection` headers passed through — or the handshake fails with a plain

400.

- **Header size limits** produce a 431 once someone's cookie jar gets fat.

The redirect loop you get exactly once

The proxy terminates TLS and speaks plain HTTP to your app. Your app therefore believes it is running on HTTP. It builds an absolute redirect: `http://example.com/dashboard`. The proxy sees an HTTP request and redirects it to HTTPS. Which reaches your app, which believes it is on HTTP, which redirects to `http://`.

`ERR_TOO_MANY_REDIRECTS`. The fix is to trust `X-Forwarded-Proto` when generating URLs, or to emit relative redirects and let the browser keep the scheme.

Two lines of Caddy

```
addaly.com {
  reverse_proxy 127.0.0.1:3000
}
```

That is a working configuration including a certificate, its renewal, HTTP-to-HTTPS redirection and modern TLS settings. nginx will ask for thirty lines and give you finer control, and far more of the internet's accumulated knowledge is written about nginx. Both are free; start with whichever you can read.

BEFORE YOU MOVE ON

You containerise a Node app that works perfectly on your laptop. The container starts with no errors, `docker ps` shows port 3000 published, and the log line says "listening on 3000" — but every request from the host hangs and then times out. What is the most likely cause?

A. The server is bound to 127.0.0.1 inside the container, so it is not listening on the interface the forwarded traffic arrives on
B. The published port mapping is reversed, so the host port and container port are the wrong way round
C. The container has no TLS certificate, so the connection is dropped during the handshake
D. The app is running as a non-root user and cannot open a port below 1024

8 · 10 min

One request, with a stopwatch

THE ONE THING TO KEEP

Most of a first request is round trips you cannot code your way out of, so measure where the time actually goes before optimising the part you wrote.

Measure first, and it takes one command

`curl` will break a request into its phases. This is the most useful command in this course:

```
curl -o /dev/null -s -w '\
dns      %{time_namelookup}s
tcp      %{time_connect}s
tls      %{time_appconnect}s
ttfb     %{time_starttransfer}s
total    %{time_total}s
' https://addaly.com/
```

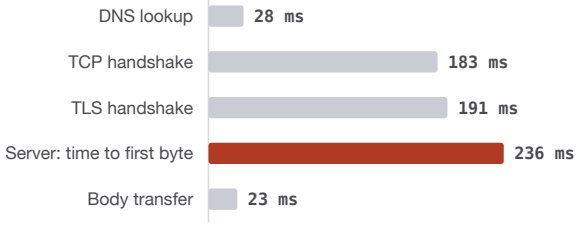
Each number is cumulative from the start, not a duration, which trips people up. Subtract to get the phases. A realistic result from a phone-speed connection in Mumbai hitting a server in Virginia:

```
dns      0.028s
tcp      0.211s
tls      0.402s
ttfb     0.638s
total    0.661s
```

Read that as: DNS took 28ms, the TCP handshake took 183ms, the TLS handshake took another 191ms, your server took 236ms to produce the first byte, and the body took 23ms.

Now notice what you just learned. Your code is responsible for 236ms of a 661ms request. Doubling the size of the server improves the number that is not the problem.

Where 661 ms went: Mumbai to Virginia, cold connection



Your code produced the first byte in 236 ms. The other 425 ms was distance and handshakes. A bigger server improves the number that is not the problem.

The part that is physics

Light in fibre travels at roughly 200,000 km per second, and real routes are not straight. Mumbai to northern Virginia is about 13,000 km, so one way is about 65ms and a round trip about 130ms at the theoretical floor. In practice you see 180 to 220ms.

Now count the round trips before your code runs:

1. DNS — one round trip, if it is not cached.
2. TCP handshake — one round trip.
3. TLS handshake — one round trip on TLS 1.3, two on TLS 1.2.

So a cold connection from India to a US server spends about 400ms doing nothing but agreeing to talk. No amount of optimisation touches that, because it is distance. This is the entire argument for a CDN: not bandwidth, but shortening the round trip. "The server is fast" and "the site is fast" are different claims, and the gap between them is measured in kilometres.

Two consolations. TLS session resumption skips a round trip on repeat visits. And keep-alive means the second request on the same connection skips both handshakes, which is why your first page load feels slow and everything after it feels fine.

The DNS number is lying to you

The first measurement includes a cold resolver lookup. Run it again and DNS will read close to zero, because your machine cached it. Both numbers are true — the first is what a new visitor gets, the second is what a returning one gets. Report which you measured.

Cold starts, and why your monitoring disagrees with your users

Two flavours, both worth knowing:

- **Serverless functions** that have been idle boot on demand: usually 100ms to 1 second, longer for a big dependency tree or a JVM.

- **Free-tier containers that scale to zero** — the default on several free plans — sleep after about 15 minutes of no traffic and take 5 to 30 seconds to wake. The first visitor of the day gets a page that appears broken.

Here is the trap. An uptime monitor pinging every five minutes keeps the container permanently warm. Your dashboard shows a fast, healthy site. Your users, arriving one at a time to a service nobody else is using, see the cold start every time. The monitoring is not wrong; it is participating.

One page is not one request

The HTML is request one. Then come the stylesheets, the fonts, the JavaScript, the images, the analytics tag and the chat widget. A modest page makes 40 to 80 requests, and requests to a new hostname pay their own DNS, TCP and TLS costs — which is why one third-party script can cost more than your entire backend.

Open the network tab, sort by time, and look at what is on the critical path. A render-blocking stylesheet on a third-party domain is a classic: 400ms of blank screen that has nothing to do with your server.

Fix in this order

1. **Shorten the distance.** Static assets on a CDN. If your framework allows it, HTML from the edge too. This is usually the largest single win and it is often free.
2. **Remove round trips.** Keep-alive, HTTP/2 or HTTP/3, fewer distinct origins, no redirect chains — each redirect is a whole extra request including handshakes.
3. **Then make your code faster.**

Almost everyone starts at three, because it is the part they wrote and the part they can see.

BEFORE YOU MOVE ON

Your server-side timing shows a median response time of 40ms, and your five-minute uptime monitor agrees the site is fast. Users in another country report that the site takes several seconds to load, especially the first visit of the day. Which explanation fits all three observations?

A. The server is under-provisioned, and the 40ms figure is measured before request queueing
B. Their internet connections are slow, so there is nothing to be done from your side
C. DNS for your domain has a very long TTL, so their resolvers are answering from a stale cache
D. Round trips over distance plus a container that scales to zero — and the monitor's regular pings are keeping it warm, so the monitor never sees the cold start

9 · 9 min

The handshake you keep paying for

THE ONE THING TO KEEP

Most of a small request is handshake, so the wins are reuse — keep-alive in, a pooled client out, and an app idle timeout longer than the proxy's, or you will serve 502s nobody can explain.

Setting up a connection costs more than using it

Before a single byte of your response moves, a new HTTPS connection costs: one round trip for the TCP handshake, then one or two more for the TLS handshake. On a 40 ms link that is 80–120 ms spent agreeing to talk. Your handler might take 5 ms.

This is why connection reuse is one of the largest, cheapest wins available to a small site, and why the same mistake shows up in three different places.

Keep-alive, and the header that turns it off

HTTP/1.1 connections are persistent by default: the browser sends a request, gets a response, and holds the socket open for the next one. `Connection: close` ends it. Some frameworks, proxies and old tutorials still set that header, and the effect is a fresh handshake per request — which on a page with 30 assets is 30 handshakes.

On the server side there is a timeout for idle keep-alive connections, usually 5 to 75 seconds. Two things about it matter:

- Each held-open connection occupies a slot. A server configured for 200 connections with a 75-second idle timeout can be full of people who left.
- If your load balancer's idle timeout is *longer* than your app's, the app will close a connection at the moment the balancer is handing it a request, and you get a small, steady drizzle of 502s that nothing in your code explains. Make the app's timeout the longer of the two. This is one of the most-reported mystery errors behind a proxy.

HTTP/2 and HTTP/3, honestly

HTTP/2 sends many requests down one connection at once, which removes the old six-connections-per-host limit and makes handshake cost almost irrelevant for a browser. In practice you get it by turning it on at your proxy or CDN; it is one line of configuration and free.

It has one real weakness: everything is on one TCP connection, so a single lost packet stalls every stream on it until the retransmission arrives. That is head-of-line blocking, and it is worst on flaky mobile networks — exactly the networks much of this audience is on.

HTTP/3 moves the whole thing to QUIC over UDP, where each stream is independent, so a lost packet affects one stream. It also merges the transport and TLS handshakes into one round trip. On a good fixed line the difference is small; on a congested mobile network it is not.

One caution repeated from the security literature: the 0-RTT resumption mode, which sends data before the handshake completes, is replayable by an attacker. Use it for GETs, never for anything that changes state.

The bug in your own outbound calls

Everything above is about traffic coming in. The version that actually costs small services money is outgoing.

If your code calls a payment provider, a mail API or your own second service with a fresh client per request, you pay TCP plus TLS every time. Under load you also exhaust ephemeral ports and accumulate sockets in `TIME_WAIT`.

```
# every call: new connection, new TLS handshake
requests.get(url)

# one connection, reused
session = requests.Session()   # create once, at module scope
session.get(url)
```

The same fix in other languages: a single `http.Client` in Go, one `HttpClient` in .NET, Node's `keepAlive` agent (on by default since Node 19, off before it), a connection pool in your HTTP library of choice. Measure it with:

```
ss -s                       # totals, including TIME_WAIT
ss -tn state time-wait | wc -l
curl -w '%{time_connect} %{time_appconnect} %{time_total}\n' -o
/dev/null -s https://example.com
```

Run that `curl` twice in a row against your own API. The second run does not get faster on its own — but comparing `time_connect` and `time_appconnect` tells you exactly how much of your latency is handshake, and therefore how much reuse would remove.

Where this shows up as a bill

Some providers charge per connection or per new TLS session; more commonly, handshakes show up as CPU. TLS termination is not free, and on a small instance a few hundred new handshakes a second is measurable. Reuse and a CDN in front are the two ways out, in that order of effort.

What to do with this

1. Confirm keep-alive is on and no proxy is sending `Connection: close`.
2. Make your app's idle timeout longer than the load balancer's.
3. Turn on HTTP/2 at the edge; add HTTP/3 if your CDN offers it, which is usually a checkbox.
4. Find every place your code makes an outbound HTTP call and give it one shared, pooled client.

The fourth item is where the surprise usually is.

BEFORE YOU MOVE ON

Your app sits behind a load balancer with a 60-second idle timeout; the app's own keep-alive timeout is 5 seconds. Users see occasional 502s with nothing in the application log. What is the mechanism?

A. The load balancer reuses a pooled connection at the moment the app is closing it, so the request is lost in flight

B. The app is running out of file descriptors because the balancer holds connections open for 60 seconds

C. TLS session resumption is failing, so some handshakes are rejected by the balancer

D. The app is too slow and the balancer is timing the request out

CASE STUDY · MODULE 1

The mock test that resolved two ways

A coaching centre in Kota moves its test-series site to a bigger machine the night before four thousand students sit a Sunday mock paper.

Sharma Classes runs a weekly test series for about four thousand students. The papers are taken on phones, in a two-hour window on Sunday morning, on a site the centre's one part-time developer built and keeps running on a rented virtual machine.

The old machine was small and had fallen over twice during a paper. So on Saturday evening the developer built a bigger one, copied the application across, restored the database, checked the site by typing the new IP address into his browser, and at half past ten at night changed the A record for `tests.sharmaclasses.in` from the old address to the new one.

At six on Sunday morning the phone started ringing.

Some students were seeing the new site, with Sunday's paper on it. Others were seeing the old site, which was still running, still showed last week's paper, and had a database nothing was writing to any more. A few were getting a full-page browser security warning. Nobody could work out what separated the three groups, because the same student got a different answer on the hostel wifi and on mobile data.

Three facts were established in ten minutes, with one command each:

- `dig +short tests.sharmaclasses.in @ns1.provider.net` — the authoritative nameserver returned the new address. The change had saved.
- `dig +short tests.sharmaclasses.in @1.1.1.1` — a public resolver returned the old address. It had asked before the change and was still holding the answer.
- `dig tests.sharmaclasses.in` — the record's TTL was 86400. One day. Nobody had touched it, because it was the default the provider set.

Nothing propagates. Records are pulled and then cached, and every cache holds its copy until the TTL runs out. A resolver that asked at nine on

Saturday evening would keep handing out the old address until nine on Sunday evening — the whole of the test window and most of the evening after it. Lowering the TTL at six in the morning changes nothing for a cache that is already holding a copy, because the copy it holds carries the old number with it.

The security warning turned out to be a different fault wearing a DNS costume. The certificate on the new machine had been issued for `tests.sharmaclasses.in` and nothing else. Students who typed `www.tests.sharmaclasses.in`, which the old machine had also answered for two years, were meeting a name the new certificate did not cover.

So at ten past six, with the paper opening at nine, there were two options and a real cost on each.

Turn the old machine off. Then every student still resolving to the old address gets nothing at all — a connection refused, for some of them until that evening. It is clean, it is honest, and it would have stopped a large share of four thousand students from sitting a paper they had paid for.

Leave the old machine running and make it forward. Put a reverse proxy on the old box that passes every request through to the new server, so both addresses serve the same site and the same database. The costs are not zero: an extra network hop for everyone still on the old address, a second machine to pay for and to remember to turn off, and a certificate on the old box that has to be valid for every name it answers to — including the `www` name that was already broken.

The centre's owner asked a third question, which was whether to postpone the paper by a week. Four thousand students had planned a specific Sunday around it, several had travelled, and a postponement is not free either.

> **STOP HERE**
>
> Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

The developer kept the old machine alive and put Caddy in front of it, forwarding to the new server, with a certificate covering both the plain name and the `www` name. That took about twenty minutes and was in place before seven. The paper ran at nine with no further complaints, and nobody was told anything had happened.

The old machine stayed up for six days. It was retired only after `dig +short ... @1.1.1.1` had agreed with the authoritative answer for a full day and its access log had been silent for twenty-four hours.

The centre changed three things afterwards. The TTL on every record is lowered to sixty seconds two days before any planned change and raised again a day after it. Certificates are checked with `openssl s_client` for every name the site answers to, `www` included, and a scheduled job alerts at twenty-one days remaining, because Let's Encrypt no longer sends the warning email. And no infrastructure change is made in the twelve hours before a test — a rule written on the office wall rather than in a document nobody opens.

Why would lowering the TTL at six on Sunday morning not have helped anybody who was already seeing the old site?

A TTL travels with the answer. A resolver that fetched the record on Saturday evening received it stamped with 86400 seconds and will not ask again until that runs out, whatever the authoritative server now says. Lowering the TTL only affects answers handed out after the change, which is exactly why it has to be done days in advance rather than at the moment you need it.

The old machine served last week's paper from a database nothing was writing to. What is the more dangerous version of that situation, and how would you avoid it?

The dangerous version is an old machine that is still writing — pointed at its own copy of the database, taking answers from the students who still resolve to it. You then have two live databases, both correct, both incomplete, and no clean way to merge them. Avoid it by making the old address stop being a destination and start being a forwarder: proxy every request to the new server so there is exactly one database, or take the old machine down entirely and accept the outage.

Students on wifi and on mobile data saw different sites. What does that tell you before you run a single command?

That the two devices are asking different resolvers and getting different answers, so the fault is in name resolution or its caching rather than in the server, the application or the certificate. One machine cannot serve two different sites to two people at the same instant; two addresses can. That single observation narrows the search to DNS before any diagnosis begins.

MODULE 1 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 You changed an A record an hour ago. You see the new server; a colleague still sees the old one. What settles where the problem is, fastest?

A. Open the registrar's control panel and confirm the record was saved correctly

B. Ask the authoritative nameserver and a public resolver the same question

C. Lower the TTL to sixty seconds and wait for the change to spread

D. Clear both browsers and reload the page with the cache disabled

2 A signed-in user requests a page belonging to another team. Your handler answers 401. What does that cause?

A. The CDN caches the refusal and serves it to other users for a minute

B. Their client tries to authenticate again, against a login they already completed

C. The browser retries with the same cookie until the request times out

D. Search engines drop the page from their index as unavailable

3 Your renewal cron ran at day sixty and reported success, but visitors still get the expired certificate. What has most likely happened?

A. HSTS is holding the old certificate in the browser until max-age expires

B. The issuer rate-limited you and silently returned the previous certificate

C. The new certificate is on disk and the web server was never reloaded

D. The server clock is wrong, so the new certificate is not yet valid

4 You turn on your framework's trust-proxy setting and rate-limit on the leftmost entry of X-Forwarded-For. What have you built?

A. A limiter that treats every visitor behind the proxy as one client
B. A limiter that blocks your own CDN's addresses under load
C. Nothing new, because the proxy overwrites the header on every request
D. A limiter anybody can step around by sending the header themselves

5 Curl reports a 661 ms request from Mumbai to a server in Virginia, of which your handler accounts for 236 ms. Which change removes the most time?

A. Serve the response from a location nearer the user
B. Double the CPU and memory on the instance
C. Add an index to the slowest query in the handler
D. Enable compression on the response body

6 Behind a load balancer you get a small, steady drizzle of 502s that nothing in your application logs explains. Which mismatch produces exactly that?

A. The application is listening on 127.0.0.1 rather than on all interfaces
B. The balancer sends Connection: close, so every request opens a socket
C. The application's idle keep-alive timeout is shorter than the balancer's
D. The application's request timeout is shorter than the balancer's read timeout

MODULE 2

Making a deploy boring

A deploy should be a non-event: one command, the same artifact, and an undo you have already practised. This module builds that, including the two things that make deploys frightening — configuration you cannot see and schema changes under live traffic.

> BY THE END OF THIS MODULE YOU CAN
>
> Ship a code change and a schema change to a live site without dropping a request, and undo either one in under two minutes

10 · 8 min

The key you must not commit

> THE ONE THING TO KEEP
>
> A leaked key is only fixed by rotating it; scrubbing git history does not un-leak anything.

Configuration that changes per environment does not belong in the file

Your database URL is different on your laptop and in production. Your Stripe key is different in test and live. So they are not code. They are inputs.

```
const key = process.env.OPENROUTER_API_KEY;
if (!key) throw new Error("OPENROUTER_API_KEY is not set");
```

Check for the value **at startup, not at use**. A missing secret that crashes the process on boot is a failed deploy you notice in thirty seconds. A missing secret discovered by a 500 error on the one route that uses it is a Saturday.

.env is a local convenience, not a deployment mechanism

- `.env` goes in `.gitignore`. Put it there before you create the file, not after.
- `.env.example` gets committed, with the variable **names** and no values. It is documentation.
- On the host, you set the real values through the platform:

```
wrangler secret put BETTER_AUTH_SECRET
fly secrets set DATABASE_URL=postgres://...
```

Those live encrypted on the platform and are injected into the process. Nobody copies a `.env` to a server over SSH.

Anything prefixed for the browser is public

`NEXT_PUBLIC_`, `VITE_`, `REACT_APP_` — these prefixes exist to tell the build tool "substitute this value into the JavaScript bundle". That bundle is downloaded by every visitor. View source, search, done.

This is the most common leak I see, and it has a predictable cause: the build tool said the variable was undefined in the browser, someone added the prefix, the error went away, and the key shipped. If the browser needs to call a paid API, the fix is a route on your own server that holds the key and calls the API on the user's behalf. That route also gives you somewhere to put a rate limit, which you are going to need anyway.

When you commit a key — and you will

Suppose the key is now in a public repository. Someone deletes the line and commits the fix. Nothing has been fixed.

The old commit is still in the history. It is in every clone anyone made. It is in every fork. GitHub keeps unreachable commits addressable for a while. And

most importantly: **automated scanners read the public events stream continuously.** Keys posted to public repositories are routinely used within minutes, without anyone cloning anything.

The only action that ends the exposure is **rotating the credential**. Go to the provider, revoke the old key, issue a new one, set it on the host.

Order matters more than people expect. Rotate first. Clean the history second, if at all. The common mistake is spending ninety minutes on `git filter-repo` while the live key stays live the entire time.

If it is a database password, rotate it. If it is a signing secret, rotate it and accept that every existing session is invalidated — that is the correct outcome, not a reason to delay.

Prevention that costs nothing

- Turn on push protection or secret scanning if your host offers it. It refuses the push.
- Run `gitleaks detect` in CI so a leak fails the build.
- Never `console.log(config)`. Logs are copied into a search index, retained for thirty days, and readable by more people than the database is.
- Give each environment its own key. Then a leak from staging is a staging problem.

The uncomfortable part

Everyone does this eventually — experienced people included. The difference between a small incident and a real one is entirely how fast you rotate, and whether you had the reflex to rotate at all rather than to tidy.

BEFORE YOU MOVE ON

A teammate pushed a database URL containing the password to a public repo. Twenty minutes later they noticed, removed the line, rewrote history with filter-repo, and force-pushed. The secret no longer appears anywhere on GitHub. Is the incident over?

A. No — the credential has to be rotated, because it was readable for twenty minutes and may already be in someone else's hands
B. Yes — the commit no longer exists on the remote, so there is nothing left for anyone to find
C. Yes, as long as the repo has no forks and the traffic graph shows no clones during that window
D. No, but only because the force-push will break collaborators' local branches and needs coordinating

11 · 9 min

The same build, everywhere

THE ONE THING TO KEEP

Build one artifact from a locked dependency tree and move that exact thing between environments, because anything rebuilt per environment is a different program.

The claim you want to be able to make

"The thing running in production is byte-for-byte the thing I tested."

Almost nobody can make that claim, and the gap is where "works on my machine" lives. There are only two disciplines needed: pin everything that can drift, and build the artifact once.

Lockfiles, and the command that respects them

Your `package.json` says `"express": "^4.18.0"`. The caret means "any 4.x above this". Your lockfile records the exact version and every transitive dependency underneath it.

- `npm install` may update the lockfile to satisfy the manifest.
- `npm ci` deletes `node_modules`, installs exactly what the lockfile says, and **fails** if the lockfile and manifest disagree.

That failure is the feature. Use `npm ci` in CI and in your image build; use `npm install` only when you are deliberately changing dependencies. The same distinction exists everywhere: `pip install -r requirements.txt` against a pinned file or `uv sync`, `bundle install --deployment`, `cargo build --locked`.

Pin the runtime too. `.nvmrc`, an `engines` field, a `python` version in `pyproject.toml`, a specific base image tag. "Node 20" is not a version; a minor release changed OpenSSL defaults once and broke a great many builds on a Tuesday.

Containers, briefly and honestly

A container image is a filesystem plus a command. It solves one problem well: the machine's own state — system libraries, locales, that `imagemagick` you installed in 2023 — stops being part of your program.

It is not free. You now maintain a Dockerfile, and image builds have their own failure modes. For a static site or a simple app on a managed platform, you may never need one. For anything with native dependencies, it earns its keep quickly.

Four things worth knowing before you write one:

Layer order decides your build time. Each instruction is a cached layer, invalidated by the first change. Copy your lockfile and install dependencies *before* copying your source, or every one-character edit reinstalls the world.

```
FROM node:22-slim
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci --omit=dev
COPY . .
CMD ["node", "server.js"]
```

Image size is mostly base image. `node:22` is around 1.1 GB. `node:22-slim` is around 200 MB. `node:22-alpine` is around 130 MB — and Alpine uses musl instead of glibc, so native modules like `sharp` or `bcrypt` may fail to build, or build and then behave differently. Use `-slim` unless you have measured a reason.

Slim removes the compilers and documentation you never use. Alpine goes smaller again by swapping glibc for musl, which is why native modules like sharp or bcrypt sometimes fail to build on it.

A `.dockerignore` is not optional. Without one you copy `node_modules`, `.git` and your `.env` into the image. The last one is a leaked secret sitting in a layer that people can pull.

Architecture is a real trap. Building on Apple Silicon produces an arm64 image. Deploying that to an x86 server gives `exec format error`, which reads like a corrupt binary. Build with `--platform linux/amd64`, or build in CI, which is x86 by default.

Docker Desktop needs a paid licence at larger companies. The free paths are identical in practice: Podman, Colima, Rancher Desktop, or the plain Docker engine on any Linux box.

Build in CI, not on your laptop

A laptop accumulates. Global installs, an old cache, an environment variable exported months ago in a shell profile, an authentication token that only ex-

ists there. CI starts from nothing every time, which is precisely why it catches things you cannot.

The one that stings is case sensitivity. macOS filesystems are case-insensitive by default; Linux is not. `import Button from "./button"` works forever on your Mac and fails in production with a bare module-not-found. There is no warning locally, ever. A Linux CI build finds it the first time.

Promote the artifact, do not rebuild it

The pattern to aim for:

1. Build once from a tagged commit. Produce a container image or a bundle.
2. Test that artifact.
3. Deploy that same artifact to staging.
4. Deploy that same artifact to production.

If staging and production each build from source, you have two builds, two dependency resolutions and two opportunities to differ — and your staging test proved something about a program you are not shipping.

Tag the artifact with the git commit SHA, and print that SHA on a `/version` endpoint. When somebody asks "is the fix live?", the answer becomes a URL rather than an argument.

BEFORE YOU MOVE ON

Your app builds and runs on your Mac. The build in CI, on Linux, fails with "Cannot find module './button'" — and the file `src/Button.tsx` is definitely there and definitely committed. What has happened?

A. The lockfile is out of date, so CI installed a different version of the bundler

B. The import uses different capitalisation from the filename, which the case-insensitive macOS filesystem forgives and the case-sensitive Linux filesystem does not

C. CI checked out a shallow clone that omitted the file

D. The `.dockerignore` or `.gitignore` is excluding the file from the build context

12 · 8 min

Which commit is live right now

THE ONE THING TO KEEP

Stamp the commit SHA into the artifact and serve it from a /version route, because "which code is running on the machine that answered this request" is the first question in every incident and the only one you can make free to answer.

The question you cannot answer at the worst moment

Something is wrong on the site. The first useful question is "what changed", and the second is "is the fix already out". Both need one fact: which commit is serving requests, right now, on the machine the user hit.

Most small projects cannot answer it. The deploy log says a deploy finished. The dashboard says the service is healthy. Neither tells you whether the instance answering this particular request is running the build from twenty minutes ago or the one from Tuesday, and during a rolling deploy the honest answer is "both".

The fix takes about fifteen minutes and pays for itself the first time you use it.

Stamp the build, then serve the stamp

At build time, put the facts into the artifact. In a Dockerfile:

```
ARG GIT_SHA
ARG BUILT_AT
ENV GIT_SHA=$GIT_SHA BUILT_AT=$BUILT_AT
```

Built from CI:

```
docker build \
  --build-arg GIT_SHA=$(git rev-parse --short HEAD) \
  --build-arg BUILT_AT=$(date -u +%FT%TZ) -t myapp:$(git rev-
parse --short HEAD) .
```

Then expose it. One route, no authentication needed if it contains nothing secret:

```
GET /version
{ "sha": "a41f9c2", "built": "2026-09-02T11:04:19Z", "env":
"production", "host": "web-3" }
```

Now "which commit is live" is a `curl`. Hit it repeatedly during a rolling deploy and you will watch the answer flip between two SHAs, which is the most direct demonstration of why two versions of your code run at once — the fact that makes migrations hard, three lessons from here.

Include the host or instance ID. "It works for me but not for my colleague" is very often one bad instance, and without an instance identifier in the response you cannot see it.

Tag the artifact with the commit, never with `latest`

`myapp:latest` is a moving label. Two machines pulling `latest` an hour apart can run different code, and a rollback to `latest` rolls back to nothing in particular. Tag images with the commit SHA, and let `latest` be an extra alias if your tooling needs one.

The same applies to a release tag. `v1.4.0` is for humans; the SHA is the identity. Keep both, and make the deploy record say which SHA a version number referred to, because tags can be moved and SHAs cannot.

Tell your tools which release this is

Every observability tool has a field for the release, and almost nobody fills it in:

- Send the SHA to your error tracker as the release. It will then tell you "this error first appeared in a41f9c2", which converts a mystery into a diff.
- Add it as a label on your metrics, but only as a low-cardinality one — a release label is fine, a per-request label is not.
- Log it once at startup, so the journal for a crashed instance says what it was running.

The payoff is a sentence you can say out loud in an incident: "errors started at 14:02, the deploy of a41f9c2 finished at 14:01, here is the diff, roll it back."

A deploy log you can read

One append-only file or table, one line per deploy: time, SHA, who or what triggered it, and the result. Most platforms keep this for you; if yours does not, a GitHub Actions step that appends to a file is enough.

The reason to have it is that outages correlate with changes, and the changes are not only yours: a dependency updated, an environment variable edited by hand, a DNS record changed, a provider migrating your instance. A shared timeline of everything that touched production is the cheapest incident tool that exists.

Provenance, briefly and honestly

A newer layer of this is the software bill of materials — a machine-readable list of what went into a build — and signed provenance attestations proving that this artifact came from that commit through that pipeline. Tools exist and are free: Syft generates an SBOM, Cosign signs artifacts, and GitHub's build attestations do both with a few lines in a workflow.

For a small site this is not urgent, and pretending otherwise wastes your attention. It matters when somebody else depends on your artifact, or when a compromised build server is in your threat model. Know that it exists, know the words, and reach for it when you have a reason rather than because a vendor page said supply chain.

The test

Ask somebody to run one command and tell you what is live. If they have to open a dashboard, guess from a deploy time, or SSH somewhere, this lesson is not done.

BEFORE YOU MOVE ON

During a rolling deploy you curl `/version` five times and get two different SHAs. What does that tell you?

A. The build is non-deterministic, producing two artifacts from one commit

B. The load balancer is caching the response and serving a stale copy

C. Old and new instances are both serving traffic, which is the normal state during a rolling deploy

D. The deploy has failed and rolled part of the fleet back

---

13 · 9 min

The checks that run without you

> THE ONE THING TO KEEP
>
> A pipeline is worth having only if it is fast enough that people wait for it and honest enough that a red build always means something is broken.

What continuous integration is for

Not ceremony. It is one thing: every push runs the same checks on a clean machine, so nobody has to remember.

A pipeline earns its place when it catches the class of mistake that humans reliably make — the forgotten file, the stale lockfile, the test that only passes because of state on your laptop.

The five checks, in this order

Order them so the fastest failures come first. There is no point spending four minutes on tests when the install was going to fail.

1. **Install from the lockfile.** `npm ci`. Fails on a lockfile nobody regenerated.
2. **Typecheck.** `tsc --noEmit`, `mypy`, whatever your language offers.
3. **Lint.** With `--max-warnings 0`, or the warnings accumulate until they are wallpaper.
4. **Tests.**
5. **Build.** The real production build, because a build error at deploy time is the worst place to find one.

```
name: ci
on: [push, pull_request]
jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version-file: .nvmrc
          cache: npm
      - run: npm ci
      - run: npm run typecheck
      - run: npm run lint
      - run: npm test
      - run: npm run build
```

`cache: npm` is doing real work: dependency installation is usually the longest step, and caching it turns three minutes into twenty seconds.

The ten-minute rule

Past about ten minutes, people stop waiting for the pipeline. They push and walk away, and the feedback arrives after they have moved on — which is the same as no feedback, plus a bill.

If you are over ten minutes: cache dependencies, run independent jobs in parallel, and split the slow end-to-end suite into a separate job that runs on the main branch rather than on every push to every branch.

Add a check for the mistake that has bitten you twice

Generic checks catch generic problems. The ones that hurt are specific to your codebase, and the reason they hurt is that the standard tools stay green.

This project has an exact example. A React component marked `"use client"` must never import a module that reaches the database — do it and the Postgres driver gets pulled into the browser bundle and every page returns a 500. TypeScript compiles it happily. ESLint approves. The site is broken and every check is green.

So there is a script, `verify:client-boundary`, that walks the import graph from every client component and fails if it reaches a server-only module. It runs in CI, before the build.

That is the pattern worth taking: when a bug gets you twice, the fix is not more care, it is a script that runs without you. Write the crude version — thirty lines of grep and import-walking — because a crude check that runs beats a rigorous one you keep meaning to write.

Make the checks actually block

A pipeline that reports failures nobody has to act on is decoration. In GitHub, mark the checks as **required** in branch protection, so a red build cannot merge. Elsewhere the setting has a different name and the same purpose.

Secrets in CI

Two rules and a surprise.

Store secrets in the CI provider's secret store, never in the workflow file. The runner masks them in logs — but only the literal value. Base64 it, print it inside a JSON blob, or pass it to a tool that reformats it, and the mask does not match and your token is in a public build log.

The surprise: **pull requests from forks do not get your secrets.** This looks broken the first time an outside contributor opens a PR and the deploy step fails. It is deliberate — otherwise anyone could open a PR whose workflow prints your production credentials.

Flaky tests are worse than no tests

A suite you re-run until it goes green is not a test suite; it is a slot machine. Once people start re-running, a real failure gets re-run too.

When a test flakes: fix it, or delete it, or quarantine it into a job that does not block. All three are honest. Leaving it in place and re-running is the only option that quietly destroys the value of everything else in the pipeline.

The free paths

GitHub Actions is free and unlimited on public repositories; private repositories on the free plan get a monthly minute allowance that is generous for a small project. GitLab CI has a free tier. If you would rather not depend on any of them, a self-hosted runner on a $5 virtual machine works, and so does Forgejo or Gitea Actions if you host your own git. The pipeline is a shell script that someone else triggers; keep it in a shape you could run by hand.

BEFORE YOU MOVE ON

Your CI pipeline takes fourteen minutes, and one integration test fails roughly one run in five for reasons nobody has traced. The team's habit is to re-run the job until it passes. What is the most serious consequence?

A. The pipeline consumes more billed minutes than it needs to

B. Deploys are delayed by up to fourteen minutes, slowing down the release cadence

C. The flaky test may be hiding a genuine race condition in the code under test

D. Re-running until green trains everyone to dismiss red builds, so a genuine failure in an unrelated check gets re-run and merged too

14 · 9 min

The console clicks nobody wrote down

THE ONE THING TO KEEP

Infrastructure created by clicking is a design nobody wrote down; put it in files so you get a diff, a rebuild and a history, and treat the plan output as the review step where a replace means a new empty database.

How infrastructure actually gets built

You click through a provider's console at eleven at night. A database, a firewall rule, an environment variable, a cron job, a DNS record, a queue, a bucket with the public flag turned off. It works, you go to bed, and the site runs for a year.

Then one of four things happens. You need a second environment and cannot remember what the first one has. Somebody deletes something and nobody knows what it was for. The provider has an outage in that region and you would like the same thing somewhere else. Or you leave, and the person after you inherits a machine nobody can describe.

The clicks were the design. They were never written down.

What infrastructure as code actually buys

Writing the infrastructure as files in your repository gives you four things, and it is worth being precise about them because the marketing is vague:

1. **A diff.** You can see that this change opens port 5432 to the world before it does.
2. **A recreate.** The environment can be built again from nothing, which is also your disaster recovery plan.
3. **A record.** Git history says who added the bucket, when, and in which pull request they explained why.

4. **A second environment for almost nothing.** Staging becomes the same file with different variables.

What it does not buy is safety from mistakes. A typo in a Terraform file can destroy a database far faster than a typo in a console, because the console asks and the tool does not.

The tools, and the free paths

Terraform is the industry standard and the one job ads name. In 2023 its licence changed from open source to the Business Source Licence, which triggered a fork: **OpenTofu**, now under the Linux Foundation, MPL-licensed, and a drop-in replacement for most projects. Both are free to use for the work you are doing here; if you want a licence with no ambiguity, use OpenTofu and the same `.tf` files.

Pulumi lets you write the same thing in TypeScript or Python, which suits people who find a new declarative language annoying. **Ansible** is the neighbouring tool for configuring a machine you already have, rather than creating it. **Docker Compose** counts too: for a single box, one committed `compose.yml` is a real, honest version of this idea and might be all you need for years.

And the smallest version that still works: a shell script called `provision.sh` in the repository, with the exact commands, run against a fresh machine once to prove it does. This is not a lesser answer for a small project. It is the answer that gets written.

The state file, which is the part that bites

Terraform and OpenTofu keep a state file mapping your code to real resources. Three facts about it, all learned the hard way:

- **It contains secrets.** Database passwords end up in plain text. Never commit it to git.
- **Two people applying at once will corrupt it.** Use a remote backend with locking — an S3-compatible bucket plus DynamoDB, Terraform Cloud's free tier, or a Postgres backend.

- **Deleting a resource from the file means deleting it in the world.** Read the plan. Every time.

That last point is why `plan` before `apply` is not a formality. The plan output names each resource and marks it as create, update, or destroy — and the ones that say *replace* are the dangerous ones, because replacing a database means a new empty database.

```
tofu plan -out=tfplan     # read this output line by line
tofu apply tfplan
```

Wire `plan` into CI on every pull request so the diff is reviewed by a person before it exists.

Drift, and the honest compromise

Somebody will fix something in the console at three in the morning during an outage. That is correct behaviour — the site matters more than the process. What must not happen is that the change stays only in the console, so the next `apply` silently reverts it.

The habit is: fix it by hand, then write it into the file within a day, then run `plan` and confirm it reports no changes. `terraform plan` with an empty diff is the check that your code and your world still agree, and running it on a schedule turns drift into a notification rather than a surprise.

What to put under it first

Do not try to codify everything. Start with the parts that are hard to remember and expensive to lose: DNS records, firewall and security-group rules, the database and its backup settings, buckets and their access policies, and the environment variables' *names* (not their values, which stay in a secret manager).

Leave the rest clicked for now, and write a one-line note in the repository saying so. A partially codified system with an honest map beats a fully codified system that nobody maintains.

BEFORE YOU MOVE ON

During an outage a colleague opens the console and widens a firewall rule to restore service. The rule is not added to the Terraform files. What happens on the next `apply`, and why?

A. The rule is preserved, because Terraform only manages resources it created

B. The apply fails with a conflict, forcing somebody to reconcile the difference

C. The rule is reverted, because the files describe the desired state and the world is corrected to match

D. The state file is updated to include the manual change, since refresh reads reality first

15 · 8 min

Deploying for nearly nothing

THE ONE THING TO KEEP

Both versions run during a deploy, so schema changes take three deploys, not one.

A deploy is four steps

Get the code onto a machine. Build it. Start it. Send traffic to it. Every platform, from a shell script to a Kubernetes cluster, is a wrapper around those four steps. When a deploy fails, ask which of the four.

Pick the shape before the vendor

What you are building determines what it costs, far more than which logo you choose.

- **Static site** — HTML, CSS, JavaScript, no server code. A bucket behind a CDN. Cloudflare Pages, Netlify, GitHub Pages. Free at any traffic level you

will see this year.

- **Static front end plus a few server routes** — same, plus functions that run per request. Free tiers cover a genuinely useful amount: hundreds of thousands of requests a month.
- **Always-on process** — you need WebSockets, background jobs, or long-running work. A small virtual machine or a container platform. The honest floor is about $4–7 a month; Hetzner's smallest is around €4.
- **A database** — managed Postgres free tiers exist (Neon, Supabase) and typically suspend when idle, which means a cold query takes a second. Fine for a side project, not fine for a checkout page.

A small real application with a few thousand visits a day sits between $0 and $10 a month. Anyone telling you otherwise is selling something.

Build once, run the same artifact

"It works on my machine" almost always means one of a short list:

- Different runtime version. Pin it. A `.nvmrc`, an `engines` field, a version in the Dockerfile.
- Different environment variables. See the previous lesson.
- **Case sensitivity.** macOS filesystems are case-insensitive by default. Linux is not. `import Button from "./button"` works perfectly on your Mac and fails in production. This one catches people repeatedly, and the error message is a bare module-not-found.
- A dependency you installed globally months ago and forgot.

Building in CI rather than on your laptop finds all four, because CI starts from nothing every time.

Boring and reversible

Four properties are worth insisting on, even for a hobby project.

1. **One command or one push.** If your deploy is eleven manual steps, someone will get step seven wrong at 1am, and that someone is you.

2. **The previous version stays online.** Rollback should be selecting an earlier build, not rebuilding from an older commit and hoping. Most platforms give you this; make sure you know where the button is *before* you need it.
3. **A health check before traffic switches.** A process that starts and immediately crashes should never receive requests.
4. **Migrations are separate from code.**

The one that surprises people: rolling deploys

During a deploy, both versions of your code are running at once. Not metaphorically — for thirty to ninety seconds, some requests hit old instances and some hit new ones.

So this breaks:

```
ALTER TABLE users RENAME COLUMN name TO display_name;
```

shipped together with the code that uses `display_name`. The migration runs, and every old instance still asking for `name` starts failing until it is replaced.

The pattern is **expand, migrate, contract**, and it takes three deploys:

1. Add `display_name`. Write to both columns. Old code keeps working.
2. Backfill. Switch reads to the new column. Deploy.
3. Once nothing reads `name`, drop it. Deploy.

Tedious. Also the difference between a rename and an outage.

Deploy on day one

Before you have an app, deploy a page that says the project's name. Point the domain at it. Get the certificate issued. Then every subsequent deploy is a small diff against something that already works, and you never face the situation where the first deploy and the first launch are the same stressful evening.

BEFORE YOU MOVE ON

You rename a database column and ship the migration in the same deploy as the code that uses the new name. The platform does a rolling replacement, so old and new instances both serve traffic for about ninety seconds. What happens?

A. Old instances keep querying a column that no longer exists, so a fraction of requests fail during the overlap
B. Nothing — the migration completes before the new code starts, so the schema and the code always agree
C. Only requests that arrive mid-restart fail, and the health check keeps those instances out of rotation
D. The database blocks the conflicting queries until the deploy finishes, adding latency but no errors

16 · 8 min

Ending a process without dropping a request

THE ONE THING TO KEEP

A deploy stops a process as well as starting one, so catch SIGTERM, fail readiness first, drain with a deadline shorter than the platform's grace period, and check that your container is not swallowing the signal at PID 1.

Every deploy kills something

A deploy is two events: a new process starts, and an old process stops. Almost all the attention goes to the first. The dropped requests come from the second.

When the old process is told to stop, it is usually in the middle of something: three HTTP requests half-answered, a database transaction open, a background job two seconds into a five-second task, a websocket holding some-

body's live page. What it does in the next few seconds decides whether your users see anything at all.

The two signals

`SIGTERM` is a polite request to stop. It can be caught, and that is the point: your code gets to decide what happens.

`SIGKILL` cannot be caught. The process ends immediately, mid-write, mid-request.

Every deploy system does the same thing: send `SIGTERM`, wait a grace period, then send `SIGKILL`. The grace period is 10 seconds in Docker and Kubernetes by default, 90 in systemd. If your requests take longer than the grace period, some of them are being cut off on every deploy, and the only sign is a small bump in 502s that looks like noise.

What graceful shutdown actually means

Four steps, in this order:

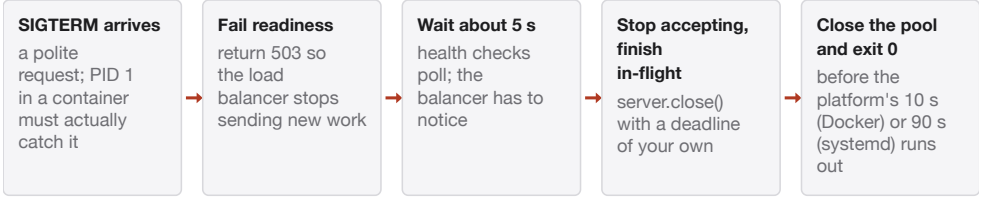
1. **Fail the readiness check first.** Start reporting not-ready so the load balancer stops sending new requests. This must happen *before* you stop accepting, and you have to wait — health checks poll every few seconds, so there is a window where the balancer still believes in you. Sleeping 5 seconds here removes more dropped requests than anything else in this lesson.
2. **Stop accepting new connections**, keep serving the ones in flight.
3. **Finish what is open**, with a deadline.
4. **Close the database pool and exit.**

In Node:

```
process.on('SIGTERM', async () => {
  await setTimeout(5000);            // let the LB notice we
are unready
  server.close(async () => {          // stop accepting, drain
in-flight
    await pool.end();
    process.exit(0);
  });
  setTimeout(() => process.exit(1), 25_000).unref();  // hard
deadline
});
```

The last line matters. A drain with no deadline is a process that never exits, and the platform will `SIGKILL` it anyway — later, and less predictably. Make your own timeout shorter than the platform's grace period, and set the platform's grace period longer than your slowest normal request.

Graceful shutdown, in the order that drops nothing



The five seconds of doing nothing after failing readiness are the most valuable five seconds in the deploy. Your own deadline must be shorter than the platform's grace period, or SIGKILL arrives mid-request anyway.

The trap in containers

In a container your process is PID 1, and PID 1 does not get default signal handling from the kernel. If it has no `SIGTERM` handler, the signal is ignored entirely and the container always dies by `SIGKILL` after the grace period.

Worse, if your image starts the app through a shell — `CMD npm start`, or an `ENTRYPOINT` written as a shell string — the shell is PID 1 and does not forward signals to its child. Your handler is there and never runs.

Two fixes: use the exec form (`CMD ["node", "server.js"]`), and run with `docker run --init` or `tini` so a real init process sits at PID 1 and forwards signals.

This is worth testing directly, because it is invisible otherwise:

```
docker run --rm --name t myapp &
time docker stop t      # ~0.2s means it handled SIGTERM; ~10s
means it did not
```

That single command has caught this bug in more projects than any amount of reading.

Background workers need the same thing, differently

A web request lasts milliseconds; a job might last minutes. Draining a worker means: stop pulling new jobs, finish the current one, then exit. If the current job is too long for the grace period, the right design is a job that checks a stop flag between steps and re-queues its remainder — which is only safe if the job is idempotent, which is the argument made in the queues lesson.

Schedulers and cron-like runners need one more: on a hard kill, the lock a job was holding must expire on its own, or the job never runs again.

What to check on your own service

- Does `SIGTERM` reach your code at all? Log a line in the handler and stop the process to see it.
- Is your drain deadline shorter than the platform grace period?
- Do you fail readiness before you stop accepting, with a pause in between?
- Does a deploy under a small load test produce zero failed requests? Run one with `oha` or `hey` — both free — and deploy while it runs. That is the only proof.

BEFORE YOU MOVE ON

Your app has a tested SIGTERM handler, yet `docker stop` always takes the full ten seconds and requests are still cut off. The Dockerfile ends with `CMD npm start`. What is happening?

A. The shell started by `CMD` is PID 1 and does not forward SIGTERM to the Node process, so the handler never runs
B. Ten seconds is the drain deadline in the handler, so it is working as written
C. Docker sends SIGKILL first for containers whose main process is a package manager
D. The load balancer is still routing traffic because readiness has not failed

17 · 8 min

The copy you are allowed to break

THE ONE THING TO KEEP

A staging environment is worth having and will still lie to you, so make it cheap, make it disposable, and put a wall between it and anything that can reach a real person.

Why a second environment at all

So that the first time code meets a real deployment is not the first time real users meet it. That is the whole argument, and it is a good one — but be clear about what a second environment can and cannot tell you.

It can tell you: the build works, the migration applies, the config is complete, the pages render, the third-party integration is wired up.

It cannot tell you: how it behaves under real traffic, with real data volumes, with real cache states, with real users doing things nobody designed for. Staging has ten rows where production has ten million, so the query that takes

3ms there takes nine seconds live. Staging has one user, so you never see the lock contention.

Knowing which list you are in stops you from over-trusting a green staging run.

Preview environments beat one shared staging

The older pattern is a single long-lived staging box that everybody shares. It has a predictable failure: two people's changes are on it at once, something breaks, and nobody knows whose.

The better pattern is **a fresh environment per branch or pull request**, created automatically and destroyed on merge. Every managed platform does this now, on free tiers. Each pull request gets a URL you can send to somebody.

The database is the awkward part, and there are three honest options:

- Seed a small fixture dataset on creation. Fast, safe, unrealistic.
- Use a database that supports cheap branching — Neon's branches are copy-on-write and take about a second.
- Restore an anonymised production dump nightly. Most realistic, most work, and the anonymisation must be real.

The rule that matters more than any of this

Nothing in a non-production environment may reach a real person.

The classic incident is not hypothetical and it happens somewhere every month: a test run in staging pointed at the production mail credentials, and forty thousand customers get an email that says `Hello {{first_name}}`. It goes out in under a minute. There is no recall.

The controls are boring and they work:

- Non-production uses a mail catcher — **Mailpit** or **MailHog**, both free, both a single binary. Every email is captured and viewable in a web UI,

and nothing leaves the machine.

- If you must use a real provider, use its sandbox mode, or enforce an allowlist of recipient domains at the sending layer, refusing anything else.
- The same rule for SMS, push notifications and webhooks out. A staging webhook hitting a partner's live endpoint is the same incident wearing different clothes.

And the sharper version: **staging must never hold production credentials for anything that writes.** Not the mail provider, not the payment processor, not the production database. If staging can write to production, staging is production with a friendlier name and worse discipline.

Anonymise properly or do not copy at all

If you restore a production dump, transform it: replace emails with `user+<id>@example.invalid`, blank phone numbers, drop payment tokens, blank uploaded files. Do it as part of the restore, in one script, so there is no window in which a raw copy is sitting on a less-guarded machine.

Be honest about what anonymisation does not achieve. A dataset with real behaviour in it can often be re-identified. If your data is sensitive, seeded fixtures are the safer answer even though they are less realistic.

Keep it cheap or it will be deleted

An always-on staging environment is a second bill for something used a few hours a week. Scale it to zero, use the smallest instance, use free tiers, and delete preview environments on merge. A staging environment that costs nothing when idle survives the first cost review; one that costs as much as production does not.

Feature flags are the third option

Sometimes the right answer is neither a branch nor a staging box: merge the code to main, ship it dark behind a flag, and turn it on for yourself, then for 1% of users. The code is in production, exercised by production traffic, with production data — and off.

This avoids the long-lived branch, which is its own tax: a branch alive for three weeks accumulates a merge conflict per day and gets tested against a world that has moved. Flags cost you cleanup — every flag is a branch in the code that must eventually be deleted — but a stale flag is a smaller problem than a stale branch.

BEFORE YOU MOVE ON

A nightly job in your staging environment runs the "re-engagement email" task against a restored copy of the production database, using credentials that happen to be the live mail provider's. What is the first control that would have prevented the incident, rather than merely detected it?

A. Point every non-production environment at a mail catcher so no message can leave the machine at all

B. Anonymise the restored dump so the addresses are not real customers

C. Add an alert on outbound email volume so the spike is noticed quickly

D. Run the nightly job only on weekdays during working hours so somebody is watching

---

18 · 11 min

Changing the schema under a running site

THE ONE THING TO KEEP

A schema change is safe when it never takes a long lock and never assumes only one version of the code is running, and both properties are things you arrange rather than hope for.

Two facts that make schema changes hard

The first you have already met: during a deploy, old and new code run at the same time. The second is that a schema change takes a lock, and locks in a

busy database queue.

Every technique below follows from those two.

The lock that turns a 1ms change into an outage

In Postgres, most `ALTER TABLE` statements take an `ACCESS EXCLUSIVE` lock, which conflicts with everything, including plain `SELECT`s.

Here is the sequence that gets people:

1. A long-running query — an analytics report, an open transaction from a stuck worker — is reading `users`. It has a shared lock.
2. Your `ALTER TABLE users ADD COLUMN` asks for an exclusive lock. It waits.
3. **Every subsequent query on `users` queues behind your waiting lock request**, because Postgres does not let new shared locks jump the queue.

The change itself takes a millisecond. The site is down for as long as that report runs. And the cause looks nothing like the symptom — your monitoring shows every query on one table timing out and a migration that has not finished.

The guard is one line before the DDL:

```
SET lock_timeout = '3s';
ALTER TABLE users ADD COLUMN display_name text;
```

Now, if the lock is not available in three seconds, your migration fails cleanly and nothing else is disturbed. Fail, retry in a minute, and let a blocked migration be a five-second annoyance instead of an incident.

Which changes are cheap and which rewrite the table

- **Adding a nullable column** is metadata only. Instant.
- **Adding a column with a constant default** has been metadata-only since Postgres 11. Before that it rewrote the whole table. A volatile default

like `random()` still rewrites.

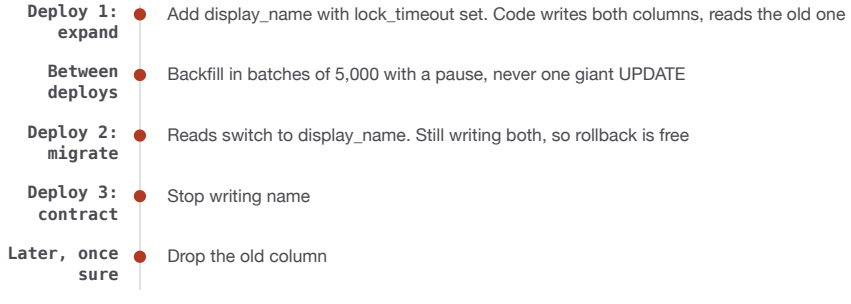
- **Adding `NOT NULL` to an existing column** scans the whole table under an exclusive lock. Instead, add a `CHECK (col IS NOT NULL) NOT VALID`, then `VALIDATE CONSTRAINT`, which takes a weaker lock.
- **Changing a column type** usually rewrites. `varchar(50)` to `text` is free; `int` to `bigint` is not.
- **Creating an index** locks out writes — unless you use `CREATE INDEX CONCURRENTLY`, which does not. Two caveats: it cannot run inside a transaction, so your migration tool needs to be told, and if it fails it leaves an `INVALID` index behind that you must drop by hand before retrying.

MySQL has its own version of this table, with different entries. The habit is the same: before you run a migration on a live database, know whether the statement rewrites the table.

Expand, migrate, contract, in detail

Renaming `name` to `display_name` is three deploys, and it is worth walking through what each one guarantees.

Renaming a column: expand, migrate, contract



Old and new code run side by side during every deploy, so no single step may assume only one version exists. Each step is reversible on its own, and code can roll back at any point because the old column stays current until the last one.

Deploy 1 — expand. Add the new column. Change the code to write to both columns and read from the old one. Old instances write only `name`, which is still the source of truth, so both versions are correct at once.

```
SET lock_timeout = '3s';
ALTER TABLE users ADD COLUMN display_name text;
```

Backfill, separately from any deploy. In batches, with a pause between them:

```
UPDATE users SET display_name = name
WHERE display_name IS NULL AND id IN (
  SELECT id FROM users WHERE display_name IS NULL LIMIT 5000
);
```

Loop that until it affects zero rows. Do not write it as a single `UPDATE users SET display_name = name`. Ten million rows in one statement holds a transaction open for minutes, blocks autovacuum on that table, and bloats it — and if it fails at 90% you get all of it rolled back and none of the work.

Deploy 2 — migrate. Switch reads to `display_name`. Keep writing both. Now roll back is still free, because `name` is still current.

Deploy 3 — contract. Stop writing `name`. Then, in a later deploy once you are sure, drop it.

Tedious. It is also the difference between a rename and an outage, and each step is individually reversible.

Where migrations run, and the race nobody expects

If your container's start command is `npm run migrate && npm start`, then deploying five containers runs the migration five times, simultaneously. Some migration tools handle this with their own lock; many produce a partial application and a wedged ledger.

Run migrations as **one separate step, before the new code rolls out** — a release job, a one-off task, a step in the deploy script. If you cannot, take an advisory lock so exactly one process proceeds:

```
SELECT pg_advisory_lock(72401);
```

Test against production-sized data

A migration on your development database with 200 rows tells you the SQL is valid. It tells you nothing about duration. The same statement against ten million rows can be forty minutes.

Restore a recent backup to a scratch instance and time it there. This is the only way to know, and it costs an hour of a Saturday against an outage you would otherwise discover live.

Forward-only, in practice

Write the `down` migration if your tool demands one, but do not plan to use it. Rolling a schema backwards after the new code has written data in the new shape loses that data. In an incident, the reliable move is to roll the **code** back — which the expand/migrate/contract sequence has kept safe — and then fix forward with a new migration in daylight.

BEFORE YOU MOVE ON

You run `ALTER TABLE orders ADD COLUMN note text` on a busy production database. It normally takes milliseconds. Instead, every query touching `orders` starts timing out, and the site is effectively down until you cancel the migration. What happened?

A. Adding a column rewrites the entire table, and `orders` is large

B. The new column was added without an index, so the query planner fell back to sequential scans

C. A long-running query already held a lock on the table, the ALTER queued behind it, and every later query queued behind the ALTER

D. The migration ran inside a transaction that was never committed, holding the lock open indefinitely

19 · 9 min

Undo, in under two minutes

THE ONE THING TO KEEP

Code rolls back, data does not, so keep them separate and make sure the undo you rely on is one you have actually performed while nothing was wrong.

The only question that matters at minute five

Something is broken and you deployed twenty minutes ago. The question is not "what is wrong". It is "how do I make it stop while I find out".

Teams that answer that question in two minutes have incidents measured in minutes. Teams that start debugging first have incidents measured in hours, because the fire keeps burning while they read.

Decide the rule in advance and write it down: **if the cause is not clear within ten minutes, roll back and diagnose from the artifact and the logs.** The evidence does not disappear when you roll back. Your users' patience does.

What is actually reversible

- **Code: yes.** A previous build artifact still exists. This is why the last module insisted on building once and promoting the artifact — rollback becomes "run the old image", not "rebuild an old commit and hope the dependency tree resolves the same way".
- **Schema: mostly no.** Covered in the last lesson. Expand/migrate/contract exists precisely so that the *code* rollback stays safe while the schema only moves forward.
- **Data: no.** Anything users wrote, or your job wrote, in the twenty minutes the bad version was live is now there. If the bad version wrote wrong data, rollback stops the bleeding and you still have a repair job to write.

Keeping these three separate is most of the skill.

What a rollback undoes, and what it does not

Reversible

- Code: redeploy the previous build artifact
- A feature behind a flag: flip a boolean, no deploy
- Config: separate system, check it was reverted too

Not reversible

- Data users or jobs wrote while the bad version was live
- A schema already written to in the new shape
- A cache full of the new object shape
- A queue full of new-format messages
- A webhook registered or a DNS record changed

Code rolls back because the previous artifact still exists. Everything the bad version wrote, told or cached stays where it is, and each of those needs its own repair.

Practise it while nothing is wrong

A rollback you have never performed is a plan, not a capability. There is always something: the button is under a menu you have not opened, the old image was garbage-collected after seven days, the command needs a flag, the deploy key has expired.

Do it deliberately on a quiet afternoon. Deploy, roll back, confirm the site is on the previous version, roll forward again. Write the exact command in the runbook. The whole exercise takes fifteen minutes and it is the highest-value fifteen minutes in this course.

Know the answers to these before you need them:

- What is the exact command or button, and who has permission?
- How far back can you go? How long are old artifacts kept?
- How long does a rollback take, measured, not estimated?
- Does it also revert environment variables and config? Usually not — config is a separate system with its own history, and a rollback that leaves the new config in place gives you a combination that has never been tested.

The kill switch, for when a deploy is too slow

A rollback takes as long as a deploy: two to five minutes at best. Sometimes that is too long, and sometimes the thing you want to disable is not the whole release.

A kill switch is a check on a value you can change **without a deploy** — an environment variable the platform can update, a row in a table, a key in Redis:

```
if (!flags.enabled("new-search")) return legacySearch(q);
```

Put one around anything expensive, anything new, and anything that calls a third party. Then a partner's outage becomes a thirty-second config change instead of an emergency release. The cost is a branch in your code and the discipline to delete flags once a feature is settled — a flag nobody has thought about in six months is a path nobody has tested in six months.

Ship dark, then to 1%

Feature flags also change how you deploy risk. Merge the code with the flag off; the code is in production, exercised by real traffic, doing nothing. Turn it on for your own account. Then 1% of users. Then 10%.

This gives you what a rollback cannot: a small blast radius on the way in. And the "rollback" for a flagged feature is a boolean, which is faster and more precise than a deploy.

The rollbacks that do not work

Reverting the code is not always enough, and these are the reasons why:

- **A cache full of the new shape.** The old code reads a serialised object it does not understand and throws. Include a version in your cache keys so a rollback misses the cache instead of poisoning itself.
- **A queue full of new-format messages.** Old workers cannot parse them and they go to the dead-letter queue. Version your job payloads and make consumers tolerate unknown fields.
- **Browsers holding the new client bundle.** A single-page app already loaded in someone's tab keeps calling an API shape you just removed. Keep the old endpoint alive for a while; do not delete an API in the same release that stops using it.

- **A third party you have already told.** A webhook registered, a DNS record changed, a payment plan created. Those do not come back on their own.

None of these makes rollback the wrong choice. They are the reasons to keep the version boundary narrow: change one thing at a time, and the thing you have to undo is one thing.

BEFORE YOU MOVE ON

Your app caches serialised objects in Redis. A release changes the object's shape, and you roll the code back thirty minutes later. The site stays broken, with deserialisation errors, even though the running code is now identical to what worked this morning. Why?

A. Redis lost its connection during the rollback and is returning empty values

B. The cache still holds objects written in the new shape under keys the old code reads, so the old code is deserialising data it was never written to handle

C. The rollback reverted the code but not the environment variables, so the app is pointed at the wrong Redis

D. Rolling back the code also rolled back the schema, so the cached objects no longer match the database

CASE STUDY · MODULE 2

Three deploys, or one, before the clinic opens

A two-doctor clinic in Nashik whose appointment system is maintained by the receptionist's nephew, on a Tuesday evening with a promised change and a two-hour window.

The clinic books about ninety appointments a day. Reception starts taking them at half past seven in the morning; the doctors see patients from eight. The system is a small web application on a container platform, running two instances, maintained by the receptionist's nephew Rohit in the evenings.

Dr Kulkarni had asked for something reasonable. The `patients` table has a column called `name`, which is printed on the prescription and shown at the top of the booking screen. She wanted the screen to show what the patient is actually called — Baby, Amma, a short form — while the prescription keeps the legal name. Rohit's plan was to rename `name` to `legal_name`, add `display_name`, and update the code.

He had Tuesday from eight until ten. The change was one migration and about forty lines.

What stopped him was reading the platform's deploy documentation properly for the first time. It performs a rolling deploy: it starts a new instance, waits for its health check, sends traffic to it, then stops an old one. For somewhere between thirty and ninety seconds, both versions of the code are running and both are serving real requests.

So shipping the rename and the code together does this. The migration runs. Every request that lands on an instance still running the old code asks for a column called `name`, which no longer exists, and fails. Reception is not open at nine at night, but two of the doctors' own phones poll the day sheet, and the pharmacy next door uses the same login.

He found a second problem while looking. The container's start command was `npm run migrate && npm start`. With two instances that is two migrations running at the same moment against the same database. It had never caused

trouble because every previous migration had been an added table. A rename applied twice does not fail politely.

And a third. The clinic runs a monthly report query that reads the whole `patients` table and takes about four minutes. An `ALTER TABLE` wants an `ACCESS EXCLUSIVE` lock, which conflicts with a plain `SELECT`. If the report is running when the migration asks, the migration waits — and every query on `patients` that arrives afterwards queues behind the waiting lock request. A change that takes one millisecond takes the clinic's booking system offline for as long as the report runs.

That left a decision with a cost on each side.

One deploy tonight. It is done, the doctor gets the feature she asked for two weeks ago, and it takes ninety minutes. The price is a window of failures during the rollout, no safe rollback — because rolling the code back leaves a schema the old code cannot read — and a migration that could stall behind the report.

Expand, migrate, contract. Add `display_name` as a nullable column tonight and write to both. Backfill in batches. Switch reads on Thursday. Stop writing the old column the following week. Drop it a fortnight later. Nothing ever breaks and every step is individually reversible. The price is three evenings instead of one, a feature the doctor has already waited for, and a conversation Rohit did not want to have, because the previous change had also been late and she had said so.

> **STOP HERE**
>
> Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

Rohit took the long route and compressed it as far as it would go. Tuesday evening: add `display_name` as a nullable column, with `SET lock_timeout = '3s'` on the line before the `ALTER`, and ship code that writes both columns

and reads the old one. The migration ran in four milliseconds at half past nine. Thursday: switch reads to the new column, still writing both. The following Tuesday: stop writing the old one. Two weeks later, `name` was dropped.

Dr Kulkarni got a working screen on Thursday, two days later than promised rather than a week, because the feature was usable as soon as reads switched.

Three other things changed that evening and outlived the feature. Migrations came out of the container start command into a separate release step that runs once before the new instances roll out. A `/version` route was added that returns the commit SHA and the instance name, so "is the fix live" stopped being an argument — during the Thursday deploy Rohit refreshed it and watched the answer flip between two SHAs, which is the clearest possible demonstration of why the rename would have broken. And on the following Sunday afternoon, with nothing wrong, he deployed something harmless and rolled it back, timing it: one hundred seconds, and the command is now written at the top of the runbook.

The migration and the code that needs it were going to ship in the same deploy. Why does that still break?

Because a rolling deploy runs both versions at once. The migration is applied the moment the release step runs, but old instances keep serving requests for another thirty to ninety seconds and they still ask for the column that has just been renamed. The two versions are not sequential, so the schema has to be compatible with both of them at every instant — which is the whole reason expand, migrate and contract takes three deploys.

Rohit's fallback plan was to roll the code back if the deploy went wrong. Why would that not have rescued him?

Code rolls back; schema does not. Once the rename has been applied, the previous build asks for a column that no longer exists, so rolling back replaces one broken version with another. Expand, migrate and contract exists precisely to keep the code rollback safe: at every step the old code still works against the current schema, so the reversible thing stays reversible.

What does `SET lock_timeout = '3s'` actually buy on a table that a four-minute report is reading?

Without it, the `ALTER TABLE` queues for the exclusive lock and every query arriving afterwards queues behind that request, so the whole table stalls for as long as the report runs and the symptom looks nothing like the cause. With it, the migration gives up after three seconds and fails cleanly, having disturbed nothing. A blocked migration becomes a five-second annoyance you retry in a minute instead of an outage.

MODULE 2 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 An API key has been pushed to a public repository. Which action actually ends the exposure?

A. Rewriting the history with git filter-repo and force-pushing every branch
B. Deleting the line in a new commit and making the repository private
C. Revoking the key at the provider and issuing a new one
D. Rotating the repository's deploy keys and inviting fewer collaborators

2 Why is `npm ci` the right install command in a pipeline, where `npm install` is not?

A. It skips the network by restoring a cached node_modules directory
B. It installs exactly the lockfile and fails when the manifest disagrees
C. It resolves dependencies to the newest versions the manifest allows
D. It omits development dependencies, so the resulting build is smaller

3 You curl /version repeatedly while a deploy is running and the commit SHA alternates between two values. What is that telling you?

A. One instance failed its health check and is restarting in a loop
B. The load balancer is not draining connections before removing instances
C. The deploy has partially failed and should be rolled back at once
D. Both versions of your code are serving real requests right now

4 A Terraform plan marks your production database as `replace`. Why is that the line to stop on?

A. Replacing means destroying the instance and creating a new, empty one
B. Replace operations cannot be applied without downtime being scheduled
C. The state file will be rewritten and any manual changes will be reverted
D. The resource will be recreated in a different region than the one you chose

5 You run `time docker stop` on your container and it takes about ten seconds every time. What does that measure?

A. In-flight requests draining correctly within the configured grace period
B. The health check taking ten seconds to report the container as unready
C. Your process never handling SIGTERM, so the runtime killed it at the deadline
D. A shutdown hook waiting for the database connection pool to close cleanly

6 Your migration tool wraps every migration in a transaction. You add a `CREATE INDEX CONCURRENTLY`. What happens?

A. It runs, but takes the same exclusive lock as an ordinary index build would
B. It succeeds, and the index is ignored by the planner until the table is analysed
C. It fails, because the statement cannot run inside a transaction block
D. It succeeds, and writes to the table are blocked until the build has finished

MODULE 3

Knowing what it is doing

A live site is opaque unless you make it otherwise. Logs, metrics, errors, health checks, alerts and traces are five different answers to five different questions, and people conflate them. Here you learn which question each one answers and how much of it you actually need.

> BY THE END OF THIS MODULE YOU CAN
>
> Answer "is it broken, for whom, and since when" in under five minutes using data you are already collecting

---

20 · 7 min

Reading logs at 2am

> THE ONE THING TO KEEP
>
> Give every request an ID and log it everywhere; that one habit turns guessing into searching.

Write for the person reading at 2am

That person is you, tired, on a phone, with none of the context you have right now. Every logging decision should be judged against that.

Here is what most people ship:

```
error [object Object]
```

Here is what helps:

```
{"level":"error","msg":"thread create
failed","request_id":"01J8QF7X",
 "identity_id":"idn_82f","room":"chai","duration_ms":30012,
 "err":"connect ETIMEDOUT 10.0.3.4:5432"}
```

One line, one event, structured so you can filter on it. The duration alone tells you this was a 30-second timeout, which tells you the database was unreachable rather than slow.

The request ID is the highest-value habit

Generate an ID for every incoming request. Put it on every log line produced while handling that request. Return it in a response header. Show it on your error page: "Something broke. Reference 01J8QF7X."

A user sends you eight characters. You search for those eight characters and get the exact sequence of events, in order, for that one request among millions. Without it, you are filtering by timestamp and guessing which of the 400 requests in that second was theirs.

Most frameworks give you this in a few lines. It is the cheapest thing in this course with the largest payoff.

Levels, used honestly

- **error** — a human needs to look at this eventually.
- **warn** — something degraded and was handled.
- **info** — business events. Account created. Payment captured. Deploy started.
- **debug** — off in production.

The common failure is not too few logs. It is an error that fires forty thousand times a day, that everyone has learned to scroll past. When everything is an error, nothing is. If a line is not actionable, it is not an error — demote it or delete it.

Never log

Passwords. Tokens. Full card numbers. One-time codes. Whole request bodies containing personal data. The `Authorization` header. Logs get shipped to a search platform, retained for thirty days, and read by more people than can read your database.

The order of questions at 2am

1. **Read the actual error string.** All of it. In a stack trace, find the first frame that is your code, not the framework's.
2. **When did it start?** Then look at your deploy list. Most breakage is the most recent deploy, and rolling back is faster than diagnosing.
3. **Is it everything or something?** Filter by route, by region, by user tier. "All requests from one region" and "all requests to one route" point at completely different causes.
4. **Is it you or a dependency?** A timeout to a host that is not yours is not your bug, though it is still your outage.

Silence is a signal

This is the one worth carrying with you. If your load balancer is reporting 502s and your application logs show **nothing at all** for that window — no errors, and no successful requests either — the requests never reached your code. Your process crashed, failed to start, or is not listening on the port the platform expects.

People lose an hour here because they read "no errors in the logs" as good news. An empty access log during a traffic spike is the loudest thing on the screen.

Logs are not monitoring

Logs tell you what happened once you go looking. Something still has to make you look. A check that hits `/health` every minute and messages you when it fails is enough to begin with, and it takes about ten minutes to set up.

BEFORE YOU MOVE ON

Users report checkout failing. Your load balancer shows a spike of 502s over a twelve-minute window. You filter application logs for that window and find nothing — no errors, and no request lines either. What does the silence most likely mean?

A. The requests never reached your application code, or the process died before it could write anything
B. The logging pipeline dropped lines under load; the app was handling and failing requests normally
C. The failures were logged at debug level, which is disabled in production
D. A 502 originates on the client side, so there would be nothing in server logs regardless

21 · 9 min

Where the logs go, and what they cost

THE ONE THING TO KEEP

Logs are a pipeline with a price per gigabyte, so log structured JSON to stdout, index only low-cardinality labels, sample the successful noise, and set a retention number on purpose rather than discovering one on an invoice.

Writing good log lines is half the job

The previous lesson was about what to write. This one is about where it lands, how you search it, how long it survives, and why logging is the line item that surprises people on the bill.

The honest starting position: if you have one machine, `journalctl` and `grep` are a complete logging system, and you should not pay for anything until they stop working. They stop working at a predictable moment — the second ma-

chine. Two machines means a user's request is in one of two files and you do not know which, so every question starts with SSHing into both.

Structured, so a machine can filter it

A line like `user 41 failed login from 10.2.0.4` is readable and unqueryable. The same facts as JSON are both:

```
{"ts":"2026-09-06T09:14:02Z","level":"warn","msg":"login
failed",

"user_id":41,"ip":"10.2.0.4","request_id":"c9f2a1","reason":"ba
d_password"}
```

Now "all failed logins for this user in the last hour" is a filter rather than a regular expression. Every language has a structured logger and they are all free: `pino` or `winston` in Node, `structlog` or the standard library's `logging` with a JSON formatter in Python, `slog` in Go, `zerolog` in Rust.

One rule that saves grief: log to stdout, as one line per event, and let something else decide where it goes. An application that writes its own files has to rotate them, and a full disk is an outage that also destroys the evidence of itself.

The pipeline, in three parts

1. **Collect.** An agent on the machine reads stdout or the journal and ships it. Free options: Vector, Fluent Bit, Promtail, or your platform's built-in agent.
2. **Store and index.** This is where money goes. Free and self-hostable: Loki (indexes labels only, greps the rest — cheap), OpenSearch, ClickHouse. Hosted: Datadog, Better Stack, Papertrail, Axiom, your cloud provider's own.
3. **Query.** Whatever the store gives you. The test of a logging setup is whether you can answer "show me every line with request ID c9f2a1, across all services, in order" in one query.

Why the bill is a surprise

Hosted log products charge by volume ingested and by retention. The numbers move, but the shape does not: roughly a few dollars per gigabyte ingested, held for a couple of weeks.

A busy small service produces more than people expect. One JSON access-log line is around 500 bytes. At 50 requests per second that is 25 KB/s — about 2 GB a day, 60 GB a month, before any application logging at all. Turn on debug logging "temporarily" during an incident and you can multiply that by ten while nobody is watching the bill.

The three controls, in order of effect:

- **Sample the boring lines.** Keep every error, every warning, and one in a hundred successful health checks and static asset requests. Most volume is 200s that nobody will ever read.
- **Set retention deliberately.** Two weeks of searchable logs plus cheap archived object storage for the rest is a common, sane split. Compliance may set a floor; nothing sets a ceiling except you.
- **Do not log what you already have as a metric.** A counter is a few bytes a minute; a log line per event is a few hundred bytes per event. If you only ever count them, count them.

The trap: high-cardinality fields in the wrong place

In a metrics system, a label with many distinct values explodes the cost. In a *log index* the equivalent mistake is indexing fields you never filter on. Loki's design makes this explicit — it indexes only the labels you declare, and labels like `request_id` or `user_id` will destroy it, precisely because they are unique per line. Keep those in the message body, where full-text search finds them, and keep labels to things with a handful of values: service, environment, level.

What never goes in a log line

Passwords, tokens, session cookies, card numbers, one-time codes, full request bodies from authentication endpoints, and personal data you would not

put in an email. Logs are copied, shipped to a third party, kept for months, and readable by everybody with dashboard access — which is a wider audience than your database.

A redaction list in the logger, applied to known field names, catches most of it. The rest is a code review habit: when you log an object, know what is in it.

The smallest setup that is genuinely enough

For a project with one or two machines and no budget:

- Log JSON to stdout, always with `request_id`, `user_id` where relevant, and the release SHA.
- Let systemd or the container runtime capture it.
- Run Grafana with Loki, either self-hosted on a small box or on Grafana Cloud's free tier, and ship with Promtail or Vector.
- Set retention to 14 days and forget about it.

That costs nothing, survives a second machine, and answers the only question that matters at 2am: what happened to this one request.

BEFORE YOU MOVE ON

A team adds `request_id` as an indexed label in Loki so incidents are easier to search. Query performance collapses and storage grows sharply. Why?

A. Request IDs are random, so compression of the log body becomes ineffective

B. Loki cannot search unindexed text, so it falls back to scanning every chunk

C. Indexed labels are stored twice, doubling ingest volume for that field

D. Loki's index is built per label combination, so a unique value per request creates a stream per request

22 · 9 min

Four numbers that describe any service

THE ONE THING TO KEEP

Averages hide exactly the requests you care about, so measure latency as percentiles from histograms and keep the number of distinct label values small.

Logs answer "what happened to this request". Metrics answer "how is it going"

They are different tools and people try to make one do the other's job. Grepping logs to work out whether the site is slower than last week is miserable; a graph answers it in a second. Conversely, a graph will never tell you why request `a7f3` failed.

The four signals

Google's site reliability book names four, and after fifteen years nobody has improved on them:

1. **Latency** — how long requests take. Split successful from failed, because a fast 500 makes your latency graph look wonderful.
2. **Traffic** — requests per second, by endpoint.
3. **Errors** — the rate of failures, as a proportion, not a count. Forty errors means nothing until you know whether it was out of a hundred or a million.
4. **Saturation** — how full the constrained resource is. Usually not CPU. Usually the database connection pool, the queue depth, or memory.

Two shorthands you will meet: **RED** (rate, errors, duration) for services handling requests, and **USE** (utilisation, saturation, errors) for resources like disks and pools. They are the same four ideas from two sides.

Start with three metrics, not thirty. A dashboard with forty panels is one nobody reads.

The average is lying to you

Take one hundred requests. Ninety-nine take 100ms. One takes 5 seconds.

- Mean: 149ms. Looks fine.
- p50 (median): 100ms. Looks fine.
- p99: 5,000ms.

The mean and median both hide the request that made somebody close the tab. Latency is not normally distributed — it has a long tail, and the tail is where the pain is. Always look at p50 alongside p95 or p99. p50 tells you what the typical experience is; p99 tells you what your worst hour looks like.

A hundred requests: ninety-nine at 100 ms, one at 5 seconds

The mean and the median both look fine. The request that made somebody close the tab only appears at p99, which is why latency is recorded as a histogram and read as percentiles.

Now the part that surprises people. Suppose a page makes 20 backend calls, and each has a 1% chance of being slow. The chance that *at least one* of them is slow is `1 - 0.99^20`, which is about **18%**. A p99 at the service level is close to a p82 at the page level. This is why tail latency matters far more than its name suggests, and why "only 1% of requests" is not the reassurance it sounds like.

Percentiles cannot be averaged

This one is worth memorising because it is done wrong constantly. You cannot take the p99 from each of your four servers and average them to get the fleet p99. You cannot average five one-minute p99s to get a five-minute p99. The result is a number with no meaning that will happily be plotted.

To aggregate correctly you need the underlying distribution, which is why metrics systems record latency as a **histogram** — a set of counters, one per bucket — and compute percentiles from the summed buckets at query time. If your monitoring only stores a pre-computed average per minute, you cannot get percentiles out of it later. Choose the histogram at the start.

Counters, gauges, histograms

- **Counter** — only goes up. Requests served, errors, bytes. You graph its rate, not its value.
- **Gauge** — goes up and down. Queue depth, memory in use, connections open.
- **Histogram** — a distribution. Latency, payload size.

Picking the wrong one is a common beginner error: recording latency as a gauge gives you the last value observed, which is noise.

Cardinality, and how a metric becomes a bill

Every distinct combination of label values is a separate stored series. `http_requests_total{path, status, method}` with 50 paths, 8 statuses and 4 methods is 1,600 series. Fine.

Add `user_id` and you have 1,600 series per user. At 50,000 users that is 80 million series. Your metrics store falls over, or your vendor's bill arrives with a number on it you did not expect, because most of them charge per series.

The rule: **labels are for values from a small, fixed set.** Path template (`/users/:id` , never `/users/8412`), status, region, method. Never user IDs, request IDs, email addresses, or full URLs with query strings. If you want per-user detail, that is what logs and traces are for.

The free path

Nothing here requires a vendor.

- **Prometheus** scrapes a `/metrics` endpoint your app exposes and stores the series. **Grafana** draws them. Both free, both self-hostable on the same

small VM as your app.

- **Grafana Cloud** has a free tier that is generous for a small project, if you would rather not run the storage.
- **VictoriaMetrics** is a lighter drop-in replacement for Prometheus storage.

Datadog and New Relic are excellent and priced per host plus per custom metric, which is how a monitoring bill quietly becomes larger than the hosting bill. Know what you are agreeing to before you add a label.

BEFORE YOU MOVE ON

Your dashboard shows a mean response time of 149ms, steady for weeks. Support keeps receiving complaints that the site "hangs sometimes". The mean has not moved. What is the most likely explanation?

A. The complaints are about client-side rendering, which server metrics cannot see

B. Users are on slow connections, so their experienced latency differs from the server's

C. The mean is computed over too long a window, so short spikes are smoothed away

D. A small fraction of requests are very slow, which barely moves the mean — the p99 would show it, and the mean is the wrong statistic for a long-tailed distribution

23 · 8 min

One screen that answers the question

THE ONE THING TO KEEP

A dashboard is a saved answer to a question somebody asks, so keep three screens — service health, dependencies, business counts — with thresholds drawn on and percentages rather than counts, and never mistake a screen for an alert.

Most dashboards are decoration

Open the monitoring of almost any small project and you find a wall of forty graphs: CPU per core, memory, disk, network in and out, garbage collection pauses, a pie chart of HTTP methods. During an incident nobody looks at it. During normal weeks nobody looks at it either.

The reason is that it was built from what the tool could draw rather than from a question somebody actually asks. A dashboard is a saved answer. If you cannot name the question, delete the panel.

Three dashboards, and only three

1. The service overview — one screen, no scrolling. Four panels, the same four for every service you will ever run:

- Request rate, split by status class.
- Error rate, as a percentage of requests, not a raw count.
- Latency at p50, p95 and p99, on one chart.
- Saturation: the one resource nearest its limit — database connections in use, queue depth, worker utilisation.

This is the whole picture of a web service. If those four are healthy, the service is doing its job, whatever the CPU graph says.

2. The dependency screen. One row per external thing you call: your database, your mail provider, your payment provider, your model API. Rate, error rate, and latency for each. This exists to answer "is it us or them" in five seconds, which is the second question in most incidents and the one that most often goes unanswered for twenty minutes.

3. The business screen. Signups, posts, orders, messages sent — the counts that mean the product is working. Keep it because technical health can be perfect while the thing is broken: a form that fails validation for everybody produces a beautiful graph of fast 200s.

That third screen is what catches the outages your monitoring is blind to, and it is the one small teams skip.

Rules that make a panel readable

- **Put a threshold line on the chart.** A latency graph without the number you consider bad requires the reader to remember it.
- **Same time range everywhere on the screen**, and default it to the last six hours, not the last hour. A spike is only obvious against yesterday.
- **Compare with the previous week** on the traffic panel. Every consumer service has a weekday shape, and "low" on Sunday morning is not an incident.
- **Percentages, not counts, for errors.** Fifty errors is meaningless without the denominator.
- **No stacked area charts for latency.** Percentiles do not add up, and stacking implies they do.
- **Name each panel as a question**: "Are we serving errors?" beats "http_requests_total by status".

Colour, honestly

About one in twelve men has a form of red-green colour blindness. A dashboard whose only signal is red versus green is unreadable to somebody on your team or in your audience. Use position, shape and text as well as colour: the number in the corner, the threshold line, the word "degraded". This is the

same accessibility rule as everywhere else, and dashboards break it more often than product pages do.

Do not put the dashboard on the critical path

Two failures worth planning around:

- **Your dashboard runs on the infrastructure it is watching.** If Grafana is on the same box as the app, a full disk takes both. Host monitoring somewhere else, or accept that during the worst incidents you are back to `journalctl`.
- **The dashboard is not an alert.** Nobody is looking at 3am. A screen answers "what is happening now that I already know something is wrong"; a page answers "wake up". Confusing the two is how teams end up staring at graphs all day and still missing the outage.

The free path

Grafana, self-hosted or on its free cloud tier, with Prometheus for metrics, is the standard free stack and the one most job ads name. Netdata gives you a decent machine-level dashboard with a single install command. Your cloud provider's built-in dashboards are free and usually enough for the overview screen.

And the smallest honest version: a single page in your own app, behind admin authentication, that prints ten numbers from ten SQL queries. It costs an afternoon, requires no new infrastructure, and for a project with a hundred users it will tell you more than any generic template.

The test

Show the overview screen to somebody who does not work on the service and ask: is it healthy, and how do you know. If they have to ask which line matters, the dashboard is not finished.

BEFORE YOU MOVE ON

Your overview dashboard shows normal traffic, sub-1% errors and flat p95 latency, but signups have been zero for three hours. Which design decision would have surfaced this soonest?

A. Adding CPU and memory panels for every instance

B. Alerting on p99 latency instead of p95

C. Tracking a business counter such as completed signups alongside the technical panels

D. Splitting the error-rate panel by endpoint

24 · 8 min

Errors that come to you

THE ONE THING TO KEEP

An error nobody sees is an error nobody fixes, so send exceptions somewhere that groups them, counts them and tells you which release they started in.

The gap nobody notices

You have logs. An exception is in them. And nobody reads logs when nothing appears to be wrong — which is exactly the situation where a bug affecting 2% of users lives for four months.

The fix is a piece of infrastructure that is genuinely small: catch unhandled exceptions, send them somewhere, and have that somewhere group identical errors together and count them.

What an error tracker does that a log file cannot

- **Grouping.** Ten thousand occurrences of the same exception become one item with a count, instead of ten thousand lines.

- **First seen and last seen.** The two most useful timestamps in an incident.
- **Release tagging.** Tie every event to the deployed commit and the question "did we cause this?" is answered by looking at when it started.
- **Breadcrumbs.** The sequence of events before the exception — the route, the query, the user action.
- **Stack traces that survive minification.** For browser JavaScript, this is the whole game. Without uploaded source maps, your stack trace reads `a.b.c:1:4213` and is worth nothing. With them, it points at a line of your code.

Wiring it up is twenty minutes

Three things to add on the server:

```
process.on("uncaughtException", (err) => { report(err); pro-
cess.exit(1); });
process.on("unhandledRejection", (err) => { report(err); });
```

Both matter. A Node process that swallows an `uncaughtException` keeps running in an unknown state, which is worse than crashing — a supervisor restarts a crashed process in a second; nothing restarts a wedged one.

Then a middleware that reports any error escaping a route handler, and a client-side handler for browser errors.

The noise, and why people stop looking

Turn on browser error reporting and your feed fills with things you cannot fix:

- Browser extensions injecting scripts and throwing inside them.
- `ResizeObserver loop completed with undelivered notifications` — benign, extremely common.
- Network errors from users on trains, in lifts, and on hotel wifi.
- Bots probing for `/wp-admin` and generating 404 handlers.
- A cross-origin script error reported as the useless string `Script error.`

If you do not filter these, the feed becomes wallpaper within a week and you stop opening it, which returns you to where you started. So: ignore-list the known noise, sample high-volume non-actionable errors, and be willing to mark things as ignored. A feed with six items you believe is worth more than one with six hundred you do not.

What not to send

Error payloads are a privacy leak waiting to happen. Sending the whole request object attaches the `Authorization` header, session cookies, the password from the login form and whatever the user typed.

Scrub at the client, before sending: drop headers by name, drop known sensitive fields, truncate bodies. Every SDK has a hook for this; use it before you turn the SDK on rather than after.

Errors are not alerts

An error tracker is a queue of things to look at, not a pager. The rule of thumb:

- **A new error type appearing after a deploy** — worth a notification.
- **An error rate crossing a threshold** — worth an alert, but that belongs with your metrics.
- **The 400th occurrence of a known issue** — worth nothing at all; you already know.

Wiring every error to a notification channel produces exactly the fatigue the next lesson is about.

The free path

Sentry has a free tier of a few thousand events a month, which is more than a small site produces, and its SDKs are the best in this space. **GlitchTip** is open source, self-hostable, and speaks the Sentry protocol, so the same SDKs work against it. **Bugsnag** and **Rollbar** are the paid alternatives.

And if you want none of them: a fifty-line handler that POSTs a JSON payload to a webhook and appends it to a table gets you grouping by a hash of the

stack trace and a count. That is 80% of the value. The point is not the tool. The point is that a human learns about the exception without going looking.

BEFORE YOU MOVE ON

You add browser error reporting and within a week the feed has 600 items a day, mostly from extensions, cross-origin `Script error.` entries and network failures on mobile. Your team has stopped opening it. What is the correct response?

A. Filter and sample the known-unactionable categories aggressively, so the remaining feed is small enough that every item gets read

B. Turn off client-side error reporting and rely on server-side errors, which are actionable

C. Route all errors into a chat channel so they are seen as they arrive

D. Raise the reporting threshold so only errors occurring more than a hundred times a day are recorded

---

25 · 9 min

The health check that lies

THE ONE THING TO KEEP

A health check that always returns 200 tests only that your web server can serve a constant, and the check you rely on must run from outside the thing it is checking.

The check that proves nothing

```
app.get("/health", (req, res) => res.json({ ok: true }));
```

This tells you that the process is running and the HTTP layer works. It will return 200 while the database is unreachable, the disk is full, the cache is down and every real request is failing.

It is not useless — a load balancer needs something cheap to poll — but it must not be the thing you trust when someone asks whether the site is up.

Liveness and readiness are different questions

- **Liveness: should I be restarted?** Answer no unless the process is genuinely wedged — a deadlock, an exhausted event loop. Keep it trivial.
- **Readiness: should I receive traffic right now?** Answer no while you are still warming up, while migrations are running, or while your database connection is broken.

Getting them the wrong way round is a classic and it is spectacular. Put a database check in your *liveness* probe and a database blip restarts every container at once — which drops every connection, which makes the database worse, which fails the next probe. A restart loop across your whole fleet, caused by a monitoring decision.

Rule: **liveness restarts, readiness routes.** Only readiness gets to depend on anything outside the process.

A deep check that does not become the problem

A useful readiness check touches the dependencies you cannot work without:

```
app.get("/ready", async (req, res) => {
  const checks = await Promise.allSettled([
    withTimeout(db.query("select 1"), 800),
    withTimeout(cache.ping(), 300),
  ]);
  const ok = checks.every((c) => c.status === "fulfilled");
  res.status(ok ? 200 : 503).json({ ok, checks:
summarise(checks) });
});
```

Two details that are not optional. **Timeouts** on every dependency, or the check hangs instead of failing and the platform learns nothing. And **cache the result for a few seconds**, because a probe every second across ten instances means ten extra database queries a second forever — and when the database is struggling, the health checks pile on and finish it off. A health check that makes an overloaded system worse is a feedback loop, not a diagnostic.

Return 503, not 200-with-a-false-field. Infrastructure reads the status code.

Your own monitoring runs on the machine that is down

This is the sentence to remember. If your health check is polled from inside the same box, the same cluster, or the same region, it goes silent exactly when you need it — and silence is not an alert.

So you need at least one check that runs somewhere else entirely, on infrastructure you do not control, from more than one region.

The free options are real:

- **UptimeRobot** — 50 monitors on the free plan, five-minute interval.
- **Uptime Kuma** — open source, self-hosted, and if you host it on a different provider from your app it counts as outside.
- **Better Stack** and **Cronitor** both have usable free tiers.
- A GitHub Actions cron job that curls your site and opens an issue on failure. Crude, free, works.

Monitor the thing that matters, not the homepage

A homepage is often static and cached, so it stays up while the application is dead. Check something that exercises the real path: log in as a test user and load a page that queries the database. That is a synthetic check, and one of them is worth twenty pings.

While you are there, monitor the things from the module ahead of you too — TLS certificate expiry and domain expiry are one HTTP check each and they prevent two of the most embarrassing outages there are.

Say 503 during a deploy

When you are shutting down, stop accepting new requests, finish the ones in flight, and answer readiness with 503 and a `Retry-After`. That is how a load balancer learns to stop sending you traffic *before* you disappear, which is the difference between a clean deploy and a handful of failed requests every time.

The signal handler is short and almost nobody writes it:

```
process.on("SIGTERM", async () => {
  ready = false;             // readiness now returns 503
  await sleep(5000);         // let the load balancer notice
  server.close(() => process.exit(0));
});
```

Those five seconds of doing nothing are the most valuable five seconds in your deploy.

BEFORE YOU MOVE ON

A brief database hiccup causes every container in your fleet to restart simultaneously, and the restart loop continues after the database recovers. Which configuration mistake produces exactly this?

A. The readiness probe has no timeout, so it hangs instead of failing

B. The health check queries the database on every probe, adding load during the incident

C. The database check is in the liveness probe, so a dependency failure is being treated as "this process is wedged, kill it"

D. The containers share a connection pool that was exhausted, so every instance failed at once

26 · 8 min

Alerts you would get out of bed for

THE ONE THING TO KEEP

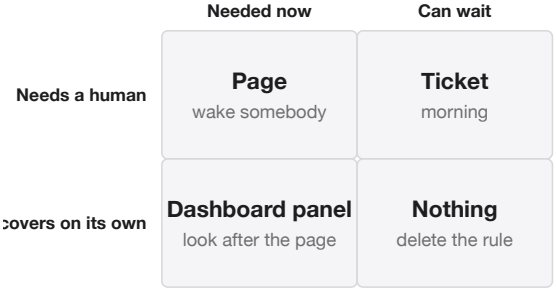
Every alert must pass two tests — a human is needed, and needed now — and an alert that fails either one is training you to ignore the ones that do not.

Two questions, asked before you create any alert

1. **Does this need a human?** If the system recovers on its own, or the fix is automatic, it does not.
2. **Does it need one now?** If the honest answer is "I would look at this on Tuesday", it is a ticket or a dashboard, not a page.

An alert that fails either test is not merely useless. It is actively harmful, because it spends your attention and teaches you that alerts can be ignored. That lesson generalises to the alert that mattered.

Two questions before any alert exists



Only the top-left box earns a page. Everything else is a ticket, a dashboard panel or nothing, and an alert that lands elsewhere trains you to ignore the one that belongs there.

Alert on symptoms, not causes

- **Cause alert:** CPU is above 90%.
- **Symptom alert:** more than 5% of requests are returning 5xx.

CPU at 90% with every user happy is a well-utilised machine and a fine Tuesday. CPU at 30% while the connection pool is exhausted is an outage. The resource number tells you nothing on its own; the user-facing number always means something.

So page on the small set of things a user would notice:

- The site is unreachable from outside.
- The error rate crossed a threshold and stayed there.
- Latency at p99 is several times normal for more than a few minutes.
- A queue is growing and not draining.
- Certificate or domain expiry is close.

Keep the cause metrics on a dashboard. They are what you look at *after* the page, to work out why.

Sustained, not instantaneous

Almost every false page is a threshold crossed for one sample. Require the condition to hold for a few minutes before it fires, and require it to clear for a few minutes before it resolves — otherwise you get a flapping alert, which is three pages instead of one and no extra information.

Burn rate, if you want to be precise about it

If you have set a target — say 99.9% of requests succeed — that is an error budget of about 43 minutes a month. You can then alert on how fast you are consuming it rather than on a raw error rate.

Burning budget at 14 times the sustainable rate over an hour means you will exhaust the month in about two days: page. Burning at 3 times over six hours means something is wrong but it can wait for the morning: ticket.

The value of this framing is that it makes the trade-off explicit. Any threshold implies a tolerance; burn rate just makes you write yours down.

Every alert needs a runbook link and an owner

The alert body should answer: what is broken, how bad, and what do I do first. A page that says `HighErrorRate: firing` at 3am and nothing else means the responder spends the first ten minutes reconstructing context they could have been handed.

If an alert has no runbook entry, either write one or delete the alert. There is no third option that survives contact with a real night.

The number that tells you the system is broken

Track what proportion of pages result in action. If more than a fifth are being acknowledged and ignored, the alerting system is decoration and you are being trained out of responding. The fix is to delete alerts, which feels irresponsible and is the correct move. Fewer, better alerts beat comprehensive coverage nobody reads.

The honest version for a small project

If this is a side project or a one-person product, most 3am pages are wrong. You are not going to fix a subtle data bug at 3am, and trying will cost you the whole next day.

So decide out loud, and write it in the runbook:

- **Wake me:** the site is completely down, or money is being lost or spent, or data is being corrupted.
- **Tell me in the morning:** everything else.

There is no shame in that list. It is a better on-call policy than an aspirational one that you will abandon in the third week.

The free path

Grafana's built-in alerting is free and self-hosted, and Uptime Kuma alerts on its own checks. For delivery to a phone: **ntfy.sh** is open source, self-hostable,

and pushes to your phone with a curl command. A Telegram bot works. So does a phone alarm triggered by a webhook.

PagerDuty and Opsgenie sell escalation policies, schedules and the guarantee that somebody eventually wakes — worth real money for a team with a rota, worth nothing for one person with a phone.

BEFORE YOU MOVE ON

You have twelve alerts. Nine of them fire at least weekly, and the team has learned which ones can be dismissed without looking. What should you do first?

A. Add severity levels so the important alerts are visually distinct from the noisy ones
B. Delete or downgrade the alerts that do not require a human right now, even though it feels like reducing coverage
C. Route the noisy alerts to a separate channel so the critical channel stays clean
D. Raise the thresholds on the nine noisy alerts until they stop firing weekly

---

27 · 10 min

Deciding how reliable it has to be

THE ONE THING TO KEEP

Pick a reliability target you can defend, express it as good events over valid events, and treat the remaining failure allowance as a budget you spend on shipping — with burn-rate alerts so one rule catches both the outage and the slow bleed.

"It should always work" is not a target

Ask anybody what reliability they want and the answer is 100%. It is the wrong answer, and not for the reason people expect.

100% is unreachable, because your users reach you through networks, DNS resolvers, browsers and phones you do not control, and those fail more often than your service will. Long before your own uptime becomes the limiting factor, you are spending money to improve a number nobody can perceive. Every extra nine costs roughly ten times the previous one, and buys a difference the user cannot distinguish from their own wifi.

So the useful question is not "how reliable can we be" but "how reliable do we need to be, and what are we willing to give up for it". That number, written down, is a service level objective.

The three words, in order

- **SLI — the indicator.** A measurement. "The proportion of requests that return a non-5xx status within 500 ms."
- **SLO — the objective.** A target for that measurement over a window. "99.5% of requests, over 30 days."
- **SLA — the agreement.** The same idea in a contract, with money attached when you miss it. Most small projects have no SLA and should not pretend otherwise.

Write the SLI as *good events divided by valid events*. That formulation forces you to define both halves, and the arguments happen at the right moment — before the incident, not during it. Are health checks valid events? Are requests from a bot? Is a 429 to an abusive client a failure? Decide once, write it down.

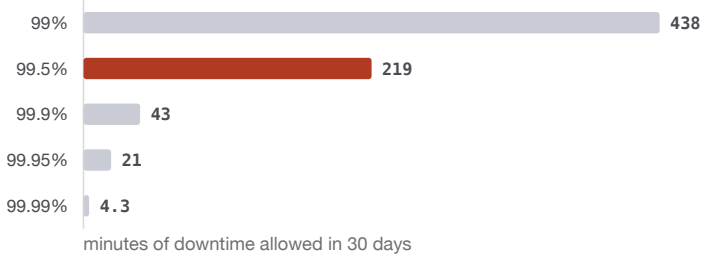
What the numbers mean in minutes

Over 30 days, allowable downtime is:

- 99% — 7 hours 18 minutes.
- 99.5% — 3 hours 39 minutes.
- 99.9% — 43 minutes.
- 99.95% — 21 minutes.
- 99.99% — 4 minutes 20 seconds.

Read that last line and notice what it demands: nobody can be woken, diagnose, and fix anything in four minutes. Four nines is not an operations target, it is an architecture — automatic failover, no single points of failure, and a team on rotation. A one-person project promising four nines is promising something it structurally cannot deliver.

What each extra nine allows in a month



Nobody can be woken, diagnose and fix anything in four minutes. Four nines is not an operations target, it is an architecture with automatic failover and a team on rotation.

For most small sites, 99.5% or 99.9% is honest, and the first is easier to defend than people think.

The error budget is the useful half

If the objective is 99.9%, then 0.1% of requests are allowed to fail. That is not a shameful residue; it is a budget, and it is meant to be spent.

The budget converts reliability from an argument into arithmetic. Plenty of budget left this month means ship the risky change, run the migration, try the new provider. Budget exhausted means stop shipping features and spend the time on reliability instead. The rule is written down in advance, so the decision during a bad month is not a negotiation between the person who wants to ship and the person who was woken up.

A small team can run this informally: one number on the dashboard, checked at the start of each week.

Burn rate, which is how you alert on it

Alerting whenever the error rate is non-zero produces noise. Alerting when the budget is fully spent is too late. Burn rate sits between: it asks how fast

you are consuming the budget relative to the whole window.

A burn rate of 1 spends exactly the month's budget in a month. A burn rate of 14.4 spends it in about two days, and a common pairing is: page if you have burnt 2% of the budget in one hour, ticket if you have burnt 10% in six hours. The fast alert catches outages, the slow alert catches a steady drizzle that would otherwise never trip a threshold.

The practical benefit is that one alert covers both the sudden total outage and the slow bleed, and neither needs a hand-tuned latency threshold.

Measure it where the user is

An SLI computed from your own server logs cannot see the requests that never arrived — DNS failure, TLS expiry, a load balancer returning 502 on its own, the whole region unreachable. Those are exactly the worst outages.

So compute the indicator as close to the user as you can afford: the CDN or load balancer's logs rather than the application's, plus an external synthetic check from somewhere else entirely. Free tiers of uptime services do the second part; a cron job on a different provider's free instance does it for nothing.

The honest version for one person

You do not need a platform for this. You need:

1. One SLI, defined in a sentence, in the repository.
2. One target, with a window.
3. One number on a dashboard: budget remaining this month.
4. One rule about what happens when it runs out.

That is a working reliability practice. Everything else — multi-window multi-burn-rate alerting, per-endpoint objectives, formal reviews — is scaling that up, and it is worth adding only when the simple version has been used for a few months and found insufficient.

BEFORE YOU MOVE ON

A service with a 99.9% monthly availability objective has burnt 60% of its error budget by day four, entirely in one 25-minute incident. Under a budget-based policy, what does that imply?

A. The objective was set too high and should be lowered to 99.5%

B. Nothing yet — the objective is measured over 30 days and the month is still within target

C. Most of the month's allowance is gone, so risky changes should wait and reliability work should take priority

D. An alert should fire immediately, because 60% consumed exceeds any sensible burn-rate threshold

28 · 10 min

Finding the slow bit

THE ONE THING TO KEEP

Slowness is nearly always waiting rather than computing, and a trace or a few timed log lines will show you what the request was waiting for.

From a request ID to a shape

You already give every request an ID and log it everywhere. A trace is the next step: instead of scattered lines, you get a tree of **spans**, each with a start time and a duration, all sharing one trace ID.

```
GET /orders                                   840ms
├─ auth.verify                                  4ms
├─ db.query orders                             31ms
├─ loop over 40 orders
│  ├─ db.query customer                         3ms
│  ├─ db.query customer                         3ms
│  └─ ... 38 more                             114ms
└─ http POST payments.example.com              610ms
```

You do not have to guess. The picture says the request spent 610ms waiting for a third party and 120ms doing a query it ran forty times.

The three culprits, in order of frequency

The N+1 query. One query to fetch a list, then one query per item. Forty items at 3ms is 120ms of latency created by a loop that reads perfectly well in code review. It is invisible until you count queries per request — and it scales with your data, so it appears three months after the code is written.

A third-party call with no timeout. Their p99 becomes your p99. Every outbound HTTP call needs an explicit timeout, and here is the subtlety: a 30-second timeout on a call inside a request that itself times out at 10 seconds is not a timeout. Set it below your own budget. Then decide what to do when it fires — a degraded page is usually better than an error page.

Waiting for a resource. The request spends 800ms not working at all: waiting for a connection from the pool, waiting for a lock, waiting behind a queue. This is the one that looks like "the database is slow" when the database is idle. If you instrument only the query and not the wait to *acquire* a connection, this time is invisible and you will chase the wrong thing for a day.

You can do this without a tracing backend

Tracing tools are good and you do not need one to start. Log a duration at each boundary with the request ID:

```
req=a7f3 span=db.orders ms=31
req=a7f3 span=http.payments ms=610
req=a7f3 span=total ms=840
```

Sort by `ms`. That is 80% of a trace for an afternoon's work, and it works with the log tooling you already have.

The database's own answers

Two Postgres features do most of the work:

`EXPLAIN (ANALYZE, BUFFERS)` runs the query and shows you the plan with real timings and real row counts. The thing to look for is a large gap between estimated and actual rows — that means the planner chose a strategy based on a wrong guess, which is how a query that was fast last month is slow now.

`pg_stat_statements` ranks queries by total time across all executions. This is where the counter-intuitive finding lives: the query taking 3ms but running 40,000 times an hour costs far more than the 900ms report somebody complained about. Optimising by total time rather than by worst single time is usually the higher-value order.

Sampling, when there is too much

Tracing every request at high traffic is expensive to store. The standard answer is **tail sampling**: keep 100% of traces that errored or exceeded a latency threshold, and 1% of the rest. You keep every interesting trace and pay for almost none of the boring ones.

The free path

OpenTelemetry is the vendor-neutral standard for producing traces; instrument once and you can change backend later. For the backend: **Jaeger** and **Grafana Tempo** are free and self-hostable; **SigNoz** bundles traces, metrics and logs in one open-source install. Honeycomb, Datadog and New Relic are the paid options, and Honeycomb in particular is very good at the high-cardi-

nality questions ("which requests are slow, and what do they have in common?") that metrics cannot answer.

Start with timed log lines. Move to real tracing when you can no longer answer "where did the time go" in one pass.

BEFORE YOU MOVE ON

A report endpoint takes 4 seconds. The database is at 12% CPU, and each of the queries the endpoint runs takes under 10ms when you run it by hand. Which explanation fits best?

A. The queries are slow only when run together, because they contend for the same table lock

B. The report result is too large to serialise quickly, so the time is spent building JSON

C. The query planner is choosing a bad plan under production data volumes

D. The endpoint runs many small queries and spends most of its time either repeating them in a loop or waiting to acquire a connection, neither of which shows up when you time one query by hand

CASE STUDY · MODULE 3

Four days of green

A district scholarship portal in Madhya Pradesh whose uptime monitor reported one hundred per cent availability for four days while no application could be submitted.

The portal takes scholarship applications from students at about two hundred government colleges. It runs on four instances behind a load balancer, in front of a managed Postgres. The deadline was the coming Saturday, which is when almost all the traffic arrives.

On Friday morning a clerk in Sagar telephoned to say that applications were not saving. The team's first reaction was that it was one college's internet, because the monitoring said the site had been up continuously for eleven weeks.

The monitoring said that because of one route:

```
app.get("/health", (req, res) => res.json({ ok: true }));
```

An external uptime service had been polling it every five minutes since launch. It returns two hundred as long as the process is alive and the HTTP layer works. It had been returning two hundred all week from four instances, none of which could reach the database.

The cause was found in about twenty minutes and was old and dull. A staff member had opened a psql session on Monday evening to check a count, run `BEGIN`, and gone home. The transaction stayed open. Autovacuum could not clean the applications table, which grew and slowed, and a long-running report query took connections and did not give them back. By Tuesday afternoon the pool was exhausted. Every write timed out. The homepage, which is static and served from a CDN, stayed up throughout, which is what everybody looked at.

The log volume was another part of the trap. Submissions were failing with a connection timeout, which was logged — but the same message had been appearing a few times a day for months, from a retry that always succeeded on

the second attempt. Nobody had demoted it, so an error that fired forty thousand times on Wednesday looked exactly like an error that had always been there.

The count, once somebody wrote the query, was 11,400 failed submissions over three and a half days, from about 6,000 distinct students, none of whom had complained to anybody at the department. They had assumed the form was hard.

That set up the decision the team argued about on Friday afternoon, once the immediate fault was fixed. The obvious fix is to make the health check tell the truth by querying the database. But the platform is configured so that a failing health check restarts the instance. Four instances polling a struggling database every five seconds, then all restarting together and reconnecting at once, is a way of finishing off a database that was merely unwell — and every one of them would then fail the next check and restart again.

Against that: a check that returns two hundred from a process that cannot serve a single request had just cost six thousand students an application window.

There was a second decision, and it belonged to the department rather than the team: whether to extend the deadline. Extending is not free. The scrutiny committee's dates are fixed, the disbursal is tied to the term, and every previous extension had produced a fresh set of last-minute applications and a complaint from the colleges that had made their students meet the original date.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They separated the two questions the check was being asked to answer. Liveness stayed trivial — it says only whether the process should be restarted, and it depends on nothing outside the process. A new readiness route runs

`select 1` with an eight hundred millisecond timeout, caches its result for five seconds so ten instances cannot become ten queries a second, and returns 503 rather than a two hundred with a false field, because the load balancer reads the status code and not the body. Readiness routes traffic; liveness restarts. Getting those the wrong way round is what would have produced the restart storm.

They added a synthetic check on a free tier of an external service, running from outside their own cloud, which logs in as a test applicant every five minutes and saves a draft. It exercises the database, so it fails when the database is unreachable.

They set `statement_timeout` and `idle_in_transaction_session_timeout` on the application role, which addressed the actual cause rather than its symptom.

And they built the screen they had never had: submissions per hour, drawn against the same hour of the previous week. On the Monday evening the two lines separate and never rejoin. Technical health had been perfect and the product had been dead, and the business count is the only graph that shows it. That chart, printed, is what persuaded the department to extend the deadline by four days.

The homepage was reachable throughout. Why is that not reassurance?

The homepage is static and served from a CDN, so it stays up while the application behind it is dead — it never touches the database. A check is only worth what it exercises. What is needed is a synthetic check that does what a user does: sign in, load a page that queries the database, and fail when that fails. One of those is worth twenty pings at the front door.

Why is putting the database check into the liveness probe the dangerous version of this fix?

Liveness answers "should I be restarted". If it depends on the database, a database blip restarts every instance at once, which drops every connection and makes the database worse, which fails the next probe, which restarts everything again — a fleet-wide restart loop caused by a monitoring decision. Only readiness may de-

pend on anything outside the process, because readiness merely stops traffic being routed to an instance that cannot serve.

Which single graph would have caught this within an hour, and why is it the one small teams skip?

Submissions per hour, compared with the same hour last week. It is skipped because it looks like a product metric rather than an engineering one, and because every technical graph on the dashboard was healthy — which is exactly the point. A form that fails for everybody produces a beautiful chart of fast responses. The business count is the only screen that can see an outage your infrastructure is blind to.

MODULE 3 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A user reports an error and quotes the reference shown on your error page. What does that reference buy you that a timestamp does not?

A. The exact sequence of log lines for that one request among millions
B. A guarantee that the error was reported to your exception tracker
C. The ability to reproduce the request against a staging environment
D. A record of which instance and which release served the request

2 Four servers each report a p99 latency per minute. Why can you not average the four to get the fleet's p99?

A. The servers do not receive equal traffic, so the average is only approximate
B. Percentiles are not additive; you need the underlying distribution to combine them
C. The minute boundaries are not aligned across servers, so the samples overlap
D. Averaging discards the tail, so the result is closer to a p95 than a p99

3 You add `request_id` as an indexed label on your log lines and your log store slows to a crawl. Why?

A. Request IDs are long strings, so the index grows faster than the messages
B. The label is written before the message, so every line is parsed twice
C. Every distinct value creates its own stream, and every line has a distinct value
D. The store refuses to compress lines whose labels change on every entry

4 Your readiness check queries the database. Ten instances are probed every second. Why cache its result for a few seconds?

A. Because probes would otherwise add ten queries a second to a struggling database
B. Because the load balancer would otherwise see readiness flapping between states
C. Because most database drivers cannot serve concurrent probes and real requests
D. Because an uncached check cannot finish inside the platform's probe timeout

5 Which of these belongs on a pager at three in the morning?

A. Disk usage on the application server has crossed seventy per cent
B. CPU on every instance has been above ninety per cent for ten minutes
C. Five per cent of requests have returned 5xx for the last ten minutes
D. The nightly export job took roughly twice as long as it usually does

6 You have a 99.9% objective. Why alert on burn rate rather than on a raw error-rate threshold?

A. It removes the need to agree a reliability target in advance
B. It fires less often, because it always uses a longer window
C. It converts latency problems into availability problems automatically
D. One rule catches both a sudden outage and a slow steady bleed

MODULE 4

Load, cost and the traffic that is not customers

Everything in this module is the same problem seen from three sides: work arriving faster than you can do it. Caching removes work, capacity planning finds the ceiling, queues move work out of the request, and rate limits keep strangers from spending your money.

> BY THE END OF THIS MODULE YOU CAN
>
> Find your first real ceiling with a load test and raise it using caching, pooling or a queue, without raising the bill

29 · 9 min

The bill, and the traffic that runs it up

> THE ONE THING TO KEEP
>
> Alerts tell you the money is gone; only caps, limits and caching stop it leaving.

Compute is rarely the surprise

People budget for the server. The bills that shock people come from four other places.

Egress. Data leaving the cloud. Coming in is free; going out is $0.08–$0.12 per GB on the large providers. Do the arithmetic once: a 4MB hero image on a page with 500,000 views is 2TB of egress, about $170–180 for one image. This is why people move media to Cloudflare R2 or Backblaze B2, which

charge nothing for egress. It is the single highest-leverage cost decision in most small projects.

Per-request pricing with no ceiling. Serverless bills per invocation and per gigabyte-second. Wonderful at your traffic. Unbounded if something loops.

Things you forgot are running. An idle load balancer is roughly $18 a month for doing nothing. A stopped VM still bills for its attached disk. Log ingestion at around $0.50/GB with 90-day retention accumulates quietly. Go and read your line items; there is usually something on there you cannot identify.

Per-token API calls. These scale with your users' enthusiasm, and with anyone who discovers an endpoint of yours that does not require an account.

The three shapes of a shock bill

- **A recursive trigger.** A function writes to a bucket; a write to that bucket triggers the function. This has produced five-figure overnight bills more than once, at real companies.
- **A retry storm.** Something downstream slows down, clients retry, every retry is another billed invocation, and the retries make the slowdown worse.
- **Someone else using your endpoint.** An unauthenticated route that calls a paid API is a free paid API for the entire internet, and it will be found by scanners within days of going live.

Alerts tell you; caps stop it

Set budget alerts — every provider has them. Put one at half of what you can genuinely afford and one at the full amount. Then be clear with yourself about what an alert is: **most cloud providers do not offer a hard stop.** The alert is a notification, often delayed by hours, sent to a human who may be asleep. It is a smoke detector, not a sprinkler.

The sprinklers are the things that act without you:

- Maximum instance count, or maximum concurrency, on your functions.
- A hard monthly spend limit on the API key at the provider that offers one.
- A queue with a bounded size in front of expensive work.

Prefer a cap that breaks a feature over an uncapped bill you have to pay in full.

Rate limits are cost control, not only abuse control

Every write path, and every path that costs money per call, gets a limit. Three decisions:

What you key on. Per-IP is weak: a whole university in Lagos or an office in Manila can share one address, so you punish a crowd; and an abuser rotates addresses for cents. Per-account is stronger. Per-API-key is strongest. Most real systems run both — a loose per-IP limit to blunt the crude stuff, a tight per-identity limit that actually protects the expensive thing.

The algorithm. A token bucket is the sensible default: a bucket holds N tokens, refills at R per second, each request takes one. It permits a burst and then enforces a steady rate. Fixed windows have a well-known flaw — send N at 11:59:59 and N at 12:00:00 and you have sent 2N in a second while staying "within limits".

The failure mode. When the rate limiter itself is unavailable, do you let traffic through or reject it? Fail open on reads, so a limiter outage is not a site outage. Fail closed on expensive or destructive writes. Decide this on a calm afternoon, because otherwise you discover your choice during the incident.

When you refuse, return `429` with `Retry-After: 30`. A silent drop or a generic 500 makes well-behaved clients retry harder.

The cheapest control is caching

A response served from a CDN never reaches your billed compute at all. `Cache-Control: public, max-age=60` on a page that changes about once a minute removes most of your load and most of your bill, and it takes one line.

BEFORE YOU MOVE ON

You ship an unauthenticated endpoint that summarises any URL using a paid model, and set a budget alert at $50. Two weeks later the bill is $2,900. The alert email did arrive, on day four. Which change would actually have prevented most of the spend?

A. Requiring an account and applying a per-identity limit, plus a hard spend cap on the model key so spending halts without a human
B. Lowering the budget alert to $10 so you find out much sooner
C. Putting the endpoint behind a CDN so repeated requests are served from cache
D. Moving from per-request serverless to an always-on VM with a fixed monthly price

30 · 9 min

The request you never serve

THE ONE THING TO KEEP

Caching converts traffic into nothing at all, and the two ways it goes wrong are serving one person's data to another and every copy expiring at the same instant.

The arithmetic

A page gets 100 requests a second. You cache it at the CDN for 60 seconds. Your origin now serves **one request a minute** instead of 6,000. That is a 99.98% reduction, achieved with one header, on hardware you already pay for.

No amount of code optimisation competes with not running the code. Caching is the first thing to reach for when load or cost is the problem, and it is the thing most people reach for fourth.

The four places a response can be cached

1. **The browser.** Free, closest to the user, and you cannot clear it.
2. **The CDN.** Shared across users, purgeable, geographically close. This is where most of the win is.
3. **Your application.** Redis, Valkey, or an in-process map. For computed results and database rows.
4. **The database.** Its own page cache, which you influence but do not control.

Different tools, one idea: keep the answer near whoever asks next.

Four places a response can be cached

Browser	free, closest to the user, and you cannot clear it
CDN	shared across users, purgeable, nearby; s-maxage and stale-while-revalidate live here
Your application	Redis, Valkey or an in-process map for computed results and rows
The database	its own page cache, which you influence but do not control

Keep the answer near whoever asks next. Most of the win is at the CDN, where one copy serves everybody for sixty seconds and the origin sees one request a minute instead of six thousand.

Two headers do most of the work

`Cache-Control` states the policy. Learn four values:

- Hashed static assets — `app.4f3a9c.js`, whose name changes when the content does:

`Cache-Control: public, max-age=31536000, immutable` A year, and `immutable` tells the browser not to even revalidate on refresh.

- HTML that everyone sees the same:

`Cache-Control: public, max-age=0, s-maxage=60, stale-while-revalidate=600` Browsers revalidate every time; the CDN (`s-maxage`) serves a copy for a minute and may serve a stale copy for ten more while it fetches a fresh

one behind the scenes. That last clause is the difference between a slow page after expiry and a fast one.

- Anything personalised:

`Cache-Control: private, no-store`

- Something that may be cached but must be checked:

`Cache-Control: no-cache` — which, confusingly, does not mean "do not cache". It means "cache, but revalidate before reuse". `no-store` is the one that means do not keep it.

`ETag` is a fingerprint. The browser sends it back as `If-None-Match`; if it still matches you return `304 Not Modified` with no body. You still pay the round trip, but not the bytes. Useful for large responses that rarely change.

The bug that leaks one user's data to another

This is the dangerous one and it is easy to write.

A page renders "Welcome back, Priya" and is served with `Cache-Control: public, max-age=300`. The CDN stores it. The next visitor is greeted as Priya, and if the page contains her order history, they can read it.

Three rules that prevent it:

- **Never mark a response `public` if it varies by session.** If the request had a session cookie and the response differs because of it, it is `private, no-store`.
- **The cache key must include everything the response varies on.** If a page differs by language, the key needs the language — either in the URL or via a `Vary` header. `Vary: Cookie` is technically correct and practically useless, because every cookie value becomes a separate entry and your hit rate goes to zero. Put the varying thing in the path instead: `/en/pricing`, `/hi/pricing`.
- **Watch for `Set-Cookie` on a cacheable response.** A cached response carrying somebody's session cookie hands their session to the next visitor.

> Most CDNs refuse to cache responses with `Set-Cookie` for exactly this reason — do not defeat that default.

Invalidation: change the name, do not purge

Purging is eventually consistent, rate-limited, and awkward at scale. The alternative is to make the URL contain a content hash, so a new version is a new URL and the old one simply stops being requested. This is what every bundler does with `app.4f3a9c.js`, and it is why static assets can be cached for a year without risk.

For data caches, the same trick works: put a version number in the key. `user:8412:v3`. Bumping the version invalidates everything without deleting anything.

The thundering herd

A popular key expires. In the same millisecond, 10,000 requests miss, and all 10,000 go to the database to compute the same value. The database falls over. Then it recovers, the key expires again, and it happens again.

Two fixes, both small:

- **Single-flight.** Only the first request computes; the rest wait for its result. Most cache libraries have this; it is a lock around the recompute.
- **Jitter the TTL.** Instead of 300 seconds, use 300 plus a random 0-60. Now the expiries spread out instead of aligning. One line, and it prevents the synchronised version of this problem entirely.

The free path

Cloudflare's free plan caches static assets globally and will cache HTML if you tell it to — that alone is a CDN most projects never outgrow. `nginx`'s `proxy_cache` and Varnish are free and run on your own box. Redis and Valkey are free. Every one of these is a one-afternoon addition, and any of them will do more for your latency and your bill than a bigger server.

BEFORE YOU MOVE ON

A logged-in dashboard page is served with `Cache-Control: public, max-age=300` and sits behind a CDN. What is the failure you should expect?

A. One user's cached page is served to the next visitor, exposing their data

B. The page becomes stale, so users see data up to five minutes old

C. The CDN refuses to cache it because the request carried a session cookie

D. Cache hit rates stay near zero because each user has a different cookie

31 · 9 min

Serving from near the user, and what bandwidth costs

THE ONE THING TO KEEP

A CDN wins twice — cached bytes never cross the ocean, and even uncacheable requests get a nearby handshake — and the number to watch is cache hit ratio, because egress metered at cents per gigabyte is what turns one large image into a bill.

Distance is the part you cannot optimise away

Your server is in one place. Your users are not. A request from Mumbai to a machine in Virginia crosses roughly 13,000 km of fibre, and light in glass manages about 200,000 km/s, so the round trip cannot be faster than about 130 ms — and in practice, with routing, it is 200 to 280 ms. Nothing you write in your handler changes that number.

A content delivery network answers from a machine near the user instead. There are two separate wins in that, and people conflate them:

1. **The cached response never travels far.** An image served from a point of presence in Mumbai arrives in 20 ms rather than 250 ms.

2. **Even uncached requests get faster**, because the expensive TCP and TLS handshakes complete against the nearby edge, which then reuses a warm, long-lived connection back to your origin. Dynamic HTML that cannot be cached at all still improves, often by 100 ms or more.

The second point is why putting a CDN in front of a fully dynamic site is still worth doing.

Anycast, in one paragraph

A CDN announces the same IP address from hundreds of locations, and the internet's routing sends each user to the nearest one. That is anycast. It is also why CDNs absorb volumetric attacks well: a flood aimed at one address is spread across every location that answers for it, instead of arriving at your single machine.

The number that tells you it is working

Cache hit ratio. If 95% of requests are answered at the edge, your origin sees one request in twenty and your bill and your CPU both reflect it. If it is 40%, something is wrong, and it is nearly always one of three things:

- A `Set-Cookie` header on responses that should be anonymous. Most CDNs refuse to cache a response carrying one, which silently disables caching for your whole site.
- `Cache-Control: private` or `no-store` applied globally by a framework default.
- Query strings that vary per user — a tracking parameter appended to every link makes every URL unique.

Check the response headers your origin actually sends, not the ones you believe it sends:

```
curl -sI https://example.com/logo.svg | grep -i -E 'cache-con-
trol|cf-cache-status|age|set-cookie'
```

`Age` tells you how long the edge has held the copy. A status header naming a hit or a miss is provided by most CDNs, and watching it change as you reload is the fastest way to learn your own configuration.

Bandwidth is where the money is

Compute costs are predictable. Egress — data leaving a provider's network — is the line that surprises people, and the prices differ by more than an order of magnitude between vendors.

As a rough shape at the time of writing: the large clouds charge in the region of 8 to 12 US cents per gigabyte out to the internet, with the first 100 GB or so free each month. Several CDN and object-storage products charge nothing at all for egress to the internet as a deliberate competitive position, and some VPS providers bundle terabytes into a flat monthly fee.

The arithmetic that matters: one 2 MB hero image on a page viewed a million times a month is 2 TB of egress. On a metered provider that is a few hundred dollars for one image. Behind a CDN with a 95% hit rate and free egress, it is close to nothing.

So the order of operations for bandwidth cost is: make the assets smaller first (a WebP or AVIF version of that image is often 200 KB), then serve them from an edge that does not meter egress, then worry about the origin.

Two terabytes leaves a metered provider at about nine cents a gigabyte. Shrink the file first, then serve it from an edge that does not meter egress, then worry about the origin.

What a CDN does not fix

- **Personalised HTML.** A page that says "Hello, Priya" cannot be shared between users. Either split it — cacheable shell, personalised fragment fetched separately — or accept an origin hit.
- **Writes.** A POST goes to the origin, every time, from wherever the user is. Sign-up forms and checkouts feel the distance even on a heavily cached site.
- **A slow origin.** A cache miss is your normal latency plus one extra hop.
- **Correctness.** A CDN will faithfully serve a wrong response to the whole world for as long as you told it to. The cache-key and private-data warnings from the caching lesson apply with a much larger audience.

Edge compute, honestly

Several providers let you run small pieces of code at the edge — Cloudflare Workers, Fastly Compute, Deno Deploy, Vercel and Netlify edge functions. It is genuinely useful for redirects, authentication checks, A/B splits, header rewriting and serving a cached page while revalidating behind it.

The constraint people meet is data. Your database is still in one region, so an edge function that queries it has crossed the ocean anyway and gained nothing. Edge code pays off when it can answer from the request itself or from edge-local storage; otherwise it is a slower place to run the same query.

The free path

Cloudflare's free plan gives anycast, TLS, HTTP/3, caching and basic bot protection at no cost, which is the single largest free upgrade available to a small site. BunnyCDN and Fastly are cheap alternatives. If you self-host, a nginx or Varnish cache in front of your app on the same machine gets you the caching but none of the geography — worth knowing, because it is often mistaken for having a CDN.

BEFORE YOU MOVE ON

After putting a CDN in front of a site, the cache hit ratio sits at 30% even for static images, and the origin still handles most traffic. Which cause fits best?

A. The CDN only caches responses smaller than a fixed size, and the images exceed it
B. Anycast is routing users to different points of presence, so each must fetch its own copy
C. The application sets a session cookie on every response, so the CDN treats them as private and refuses to cache
D. HTTP/3 is enabled, and QUIC connections bypass the edge cache

32 · 10 min

Finding your ceiling before your users do

THE ONE THING TO KEEP

Capacity is measured, not guessed, and past the knee of the curve latency rises while throughput does not, so shedding load beats queueing it.

Guessing is the norm and it is always wrong

Ask most people what their app can handle and you get a shrug or a number from a blog post. Both are worthless because the answer depends on your queries, your data volume and your pool sizes.

It takes about an hour to find out for real.

The tools are free

- **k6** — write the scenario in JavaScript, run one binary. The best default.
- **oha** and **hey** — one command, immediate answer, good for a single endpoint.

- **wrk** — very high throughput from one machine.
- **Locust** — Python, good for complex user journeys.

```
oha -z 60s -c 50 https://staging.example.com/search?q=chair
```

Fifty concurrent connections for sixty seconds. Note it points at staging. Load-testing production is occasionally right, but do it deliberately, at a quiet hour, with a plan for stopping.

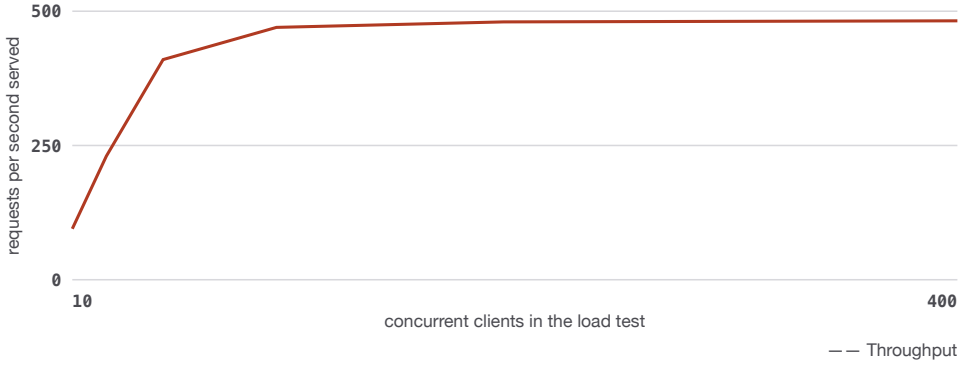
Run the same test at increasing concurrency — 10, 25, 50, 100, 200 — and record throughput and p99 latency at each step. That table is your capacity curve.

The knee

Throughput rises with concurrency, then flattens. Past that point, adding load does not add throughput; it only adds latency, because the extra requests are queueing.

That flattening is the **knee**, and it is your capacity. Beyond it the system is not "a bit slower" — latency climbs sharply, because queue wait grows without bound while service rate stays fixed. This is why a site under overload does not degrade gracefully by default; it goes from fine to unusable across a narrow band of traffic.

The knee: throughput stops rising, the queue does not



Past about a hundred concurrent clients this service adds no throughput at all. Every extra client only lengthens the queue, so p99 latency at 400 clients was eight times the figure at 50. Shed load there rather than queue it.

Little's Law, which is the only formula here

Concurrency = throughput × latency.

At 50 requests per second with 200ms average latency, you have 10 requests in flight at any moment. If your database connection pool has 5 connections, half of them are already waiting for a connection rather than doing work.

Run this backwards to size things. Want to serve 200 requests per second at 100ms? That is 20 in flight, so you need at least 20 pool slots and enough workers to hold 20 concurrent requests. One multiplication, and it replaces most capacity guesswork.

What actually breaks first

In rough order of frequency:

1. **Database connections.** Almost always. Details in the next lesson.
2. **A single unindexed query** that was fine at 5,000 rows and is a three-second sequential scan at 5,000,000.
3. **Memory**, then swapping, then the OOM killer removing your process without explanation.
4. **File descriptors**, on the default limit of 1024 on some systems.
5. **CPU.** Genuinely last, and much later than people expect.

Most side projects fall over at 20 requests a second, and it is almost never the language or the framework. It is one query.

Shed load, do not queue it forever

When you are past capacity, you have two choices: queue everything, or refuse some of it.

Queueing feels fairer and is worse. A user who waited 30 seconds has already closed the tab, but your server is still working on their request, still holding a connection, still making the queue longer. You end up doing 100% of the work and delivering 0% of the value.

So: set a request timeout, cap the queue depth, and return `503` with `Retry-After` when it is exceeded. Serving 90% of people quickly and telling 10% to come back is a better outcome than serving nobody. This is called **load shedding** and it is the single most useful overload behaviour to configure in advance, because you will not be configuring anything thoughtfully at the time.

Vertical before horizontal

Doubling the size of one machine is a two-minute change with no architectural consequence. Going from one machine to three means sessions must not live in memory, caches must be shared or accept divergence, scheduled jobs must not run three times, and your logs are now in three places.

Scale up until it is genuinely expensive, then scale out. A single modern VM with 4 vCPU and 8 GB serves a surprising amount of traffic — thousands of simple requests per second — and almost every product that "needed" to scale out at ten requests a second needed an index.

Do the test again after you change anything

A capacity number ages. New features add queries; new data changes plans. Re-run the same test after any significant release and keep the numbers in a file. Two numbers a month apart tell you more than one number ever will.

BEFORE YOU MOVE ON

You load-test an endpoint at increasing concurrency. Throughput rises to 180 requests per second at 60 concurrent, then stays at about 180 while p99 latency climbs from 300ms to 4 seconds at 400 concurrent. What does this tell you, and what should you configure?

A. The server needs more CPU, since throughput has plateaued at its processing limit
B. Latency is growing because of network saturation between the load generator and the server
C. Capacity is about 180 per second, the extra concurrency is only adding queue time, and you should cap the queue and return 503 rather than accept work you cannot deliver
D. The test is invalid because a single load generator cannot produce accurate results above 60 concurrent connections

33 · 9 min

The day you run two copies

THE ONE THING TO KEEP

Running two copies breaks everything held inside one process — sessions, counters, uploads, caches and cron — so externalise that state rather than pinning users to an instance, and test it by restarting the process while logged in.

The second instance breaks things the first one hid

One process is a simple world. Anything you keep in a variable is available to every request, a file written by one request is readable by the next, and a scheduled task runs once because there is one of it.

Start a second copy and every one of those assumptions quietly stops being true. The site does not fall over; it misbehaves in ways that look like bugs in

your code. This lesson is the list of what breaks, because knowing it in advance is the difference between an afternoon and a fortnight.

The five things that break, in the order you meet them

1. Sessions in memory. A user logs in on instance A and their next request lands on instance B, which has never heard of them. The symptom is being randomly logged out, roughly half the time with two instances. Fix: put sessions in Redis, in the database, or in a signed cookie. Not sticky sessions, for reasons below.

2. Rate limits and counters in memory. With two instances your limit of 100 requests an hour is now 200, and neither process knows about the other. Anything that counts across requests needs shared storage.

3. Uploads on the local disk. The file is written to instance A, and the request that displays it hits instance B, which returns 404. This one is covered properly in the next lesson; the short version is that local disk is scratch space, never storage.

4. Caches in memory. Not wrong, but now inconsistent: two processes hold two versions of the same value and invalidating one leaves the other. Acceptable for short TTLs, dangerous for anything a user just edited.

5. Scheduled jobs. Your nightly email runs twice, because both instances have the same cron. Every scheduled task needs a lock — a row in the database taken with a conditional update, or a single dedicated scheduler process.

The pattern under all five: state that lives inside a process cannot be shared by processes.

Sticky sessions are a trap that works

A load balancer can pin each user to one instance, which makes problems 1, 2 and 4 disappear without changing any code. It is genuinely tempting, and it is why so many systems have it.

What you have bought: when that instance restarts — and it restarts on every deploy — everybody pinned to it loses their state at once. Load also stops dis-

tributing evenly, because a long-lived user stays where they were regardless of how busy that box is, and autoscaling cannot drain an instance without dropping people.

Use stickiness deliberately for things that genuinely cannot move, such as an open websocket. Do not use it as a substitute for externalising state.

Statelessness is a property you can test

The test is blunt and takes two minutes: while using the site as a logged-in user, restart the process. If your session, your draft, your cart or your upload disappears, that state was in the process.

Run the same test with two instances behind a proxy on your laptop. Docker Compose with `--scale web=2` and any small proxy in front is enough. Almost every problem in this lesson shows up locally in ten minutes, and none of them show up in a single-process development server — which is why they are usually discovered in production.

Autoscaling does not do what people expect

Three honest limitations:

- **It is slow.** A new instance takes 30 seconds to several minutes to boot, install and warm up. A traffic spike that arrives in ten seconds is over before capacity appears. Autoscaling handles daily curves well and sudden spikes badly.
- **It scales the thing you measured, not the bottleneck.** Scaling on CPU when the constraint is database connections adds instances that queue harder on the same database — and, past that point, makes latency worse rather than better.
- **Scale to zero means cold starts.** Free tiers that sleep after inactivity give the first visitor a multi-second wait, and that visitor is often the one you cared about.

A fixed number of instances with headroom is a legitimate, cheaper answer for most small services. Measure the ceiling as in the previous lesson, run at

half of it, and revisit quarterly.

Where the real limit lives

Adding web instances is easy because they are stateless once you have done the work above. What does not scale by copying is the thing they all share: one database. That is the subject of the next lesson, and it is why "just add servers" stops working at a predictable and surprisingly early point.

The free path

Docker Compose with two replicas and Caddy or nginx in front costs nothing and reproduces every problem here. Redis or Postgres serve perfectly well as your shared session and counter store — you almost certainly already run one of them, and adding a second dependency for this is rarely justified at small scale.

BEFORE YOU MOVE ON

A team scales from one instance to four and users start being logged out at random. They enable sticky sessions and the complaints stop. What have they actually done?

A. Fixed the problem, since sessions now consistently reach the instance that holds them

B. Introduced a correctness bug, because sticky sessions cannot guarantee routing to the same instance

C. Hidden the symptom and made deploys a source of mass logouts, because the session state still lives in one process's memory

D. Traded latency for reliability, since requests now travel further to reach their pinned instance

34 · 10 min

It is almost always the database

THE ONE THING TO KEEP

The first ceiling is nearly always connections and the second is one missing index, and both are found with two queries you can run today.

Connections run out before anything else

Postgres defaults to `max_connections = 100`, and each connection is a separate operating system process using several megabytes. This is not a setting to raise casually; a thousand connections mostly means a thousand processes competing to do nothing.

Now count what your application opens. Three containers, each with a pool of 20, is 60. Add a background worker, a migration job, your own `psql` session and a metrics exporter and you are near the limit while serving very little traffic.

Serverless makes this dramatically worse: every concurrent function instance opens its own connection, and there is no shared pool. A hundred concurrent invocations is a hundred connections, and the error is immediate and total:

```
FATAL: sorry, too many clients already
```

The fix is a pooler, not a bigger `max_connections`. PgBouncer is the classic; managed equivalents include Supabase's pooler, Neon's pooled endpoint, and Cloudflare Hyperdrive, which is what this project uses from Workers. A pooler holds a small number of real connections and multiplexes many clients over them.

One caveat that appears only under load, which is the worst time to learn it: in **transaction pooling** mode a client gets a different backend for each transaction. So session-level state — `SET` statements, session variables, `LISTEN`,

temporary tables, and historically server-side prepared statements — does not survive between statements. Your ORM may need a flag. It works perfectly in development, where you have one client, and fails in production.

The missing index

A sequential scan over 1,000 rows is instantaneous. The same scan over 5,000,000 rows is seconds, and it reads the whole table into memory, evicting everything useful from the cache on the way.

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM orders WHERE cus-
tomer_id = 8412;
```

Read three things from the output: `Seq Scan` versus `Index Scan`, the actual time, and the gap between estimated and actual rows. A large gap means the planner's statistics are wrong, which is what makes a query that was fast for a year suddenly slow.

Two rules that cover most cases. Index the columns you filter and join on. And for a composite index, **column order matters** — an index on `(customer_id, created_at)` serves a query filtering on `customer_id` alone, but an index on `(created_at, customer_id)` does not.

Indexes are not free: each one is written on every insert and update, and takes disk. Find the ones earning nothing:

```
SELECT relname, indexrelname, idx_scan
FROM pg_stat_user_indexes WHERE idx_scan = 0;
```

Find the expensive query, not the slow one

Enable `pg_stat_statements` and sort by total execution time:

```
SELECT calls, round(total_exec_time) AS total_ms,
       round(mean_exec_time::numeric, 2) AS mean_ms, query
FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 20;
```

The result is usually a surprise. The 900ms report somebody complained about runs four times a day. The 3ms query at the top runs 40,000 times an hour and is consuming most of your database. Optimise by total time and you fix the thing that is actually holding you back.

Two settings that contain the damage

```
ALTER ROLE app SET statement_timeout = '15s';
ALTER ROLE app SET idle_in_transaction_session_timeout = '30s';
```

The first stops one runaway query from occupying a connection for an hour. The second kills a client that opened a transaction and wandered off — which is the single most destructive thing a client can do, because an open transaction blocks `VACUUM` from cleaning up, so the table grows and grows and every query on it gets slower. Bloat from a forgotten transaction is a genuinely common cause of "the database got slow over the weekend".

Read replicas are not a free win

A replica moves read load off the primary, and introduces replication lag — usually milliseconds, occasionally seconds under write pressure. Which means: a user posts a comment, the write goes to the primary, the page reloads from the replica, and the comment is not there.

The fix is to route reads that must be current back to the primary, per request. That is a real design decision, not a configuration flag, and it is why replicas belong later in a project's life than most people put them.

The free path

All of the above is Postgres and free. PgBouncer is free. **pgAdmin**, **DBeaver** and `psql` are free. Neon and Supabase have free tiers that suspend when idle, so the first query after a quiet period takes around a second — fine for a side project, wrong for a checkout page. The paid tiers mostly sell you connections, uptime and backups, and by the time you need them you will know which.

BEFORE YOU MOVE ON

Your API runs as serverless functions. Under a traffic spike, requests start failing with "too many clients already" from Postgres, even though database CPU is low. What is the right first move?

A. Increase `max_connections` on the database to match peak concurrency
B. Put a connection pooler in front of the database so many function instances share a small number of real connections
C. Add a read replica so connections are spread across two databases
D. Reduce the pool size configured inside each function instance

35 · 10 min

Work that does not belong in a request

THE ONE THING TO KEEP

Move anything slow or externally dependent out of the request into a queue, and design every job to be safe to run twice, because at some point it will be.

The test

Would this still be correct if it took thirty seconds and sometimes failed? If yes, it does not belong in the request.

The usual list: sending email, generating thumbnails or PDFs, calling a slow third-party API, syncing to another system, building an export, sending webhooks out, anything that fans out to many recipients.

Leave them in the request and two things happen. The user waits for work they did not ask to wait for. And when the request times out, the work is lost halfway — the order is saved and the confirmation email is not, with nothing recording that.

Your database is a perfectly good queue

Before reaching for infrastructure, know that Postgres does this well:

```
CREATE TABLE jobs (
  id bigserial PRIMARY KEY,
  kind text NOT NULL,
  payload jsonb NOT NULL,
  run_after timestamptz NOT NULL DEFAULT now(),
  attempts int NOT NULL DEFAULT 0,
  status text NOT NULL DEFAULT 'pending'
);
```

A worker claims work with one statement:

```
UPDATE jobs SET status = 'running', attempts = attempts + 1
WHERE id = (
  SELECT id FROM jobs
  WHERE status = 'pending' AND run_after <= now()
  ORDER BY id FOR UPDATE SKIP LOCKED LIMIT 1
)
RETURNING *;
```

`FOR UPDATE SKIP LOCKED` is the clause that makes this work: each worker locks a row and other workers step over locked rows instead of waiting. Ten workers, no coordination, no duplicates, no extra service to run or pay for. This handles far more load than most projects will ever produce, and it means your jobs are in the same transaction and the same backup as your data.

Move to Redis with BullMQ, RabbitMQ, SQS or Cloudflare Queues when you need throughput a database cannot give you, or when you want the operational features they bring. Not before.

Every job will run twice

Queues deliver **at least once**. A worker can do the work, then crash before marking the job done, and the job comes back. This is not a bug you can configure away; it is the nature of the thing.

So every job must be **idempotent** — safe to run twice with the same effect as running once. In practice:

- Give the job an idempotency key and record completed keys. Check before acting.
- Prefer operations that are naturally repeatable: setting a value rather than incrementing it, `INSERT ... ON CONFLICT DO NOTHING` rather than a bare insert.
- Where the side effect is external, use the provider's idempotency key — every serious payment and email API has one.

Otherwise the user gets the email twice, or gets charged twice, and the second one is not a small problem.

Retries, backoff and jitter

Retry with exponential backoff: 1s, 2s, 4s, 8s, capped at a few minutes. And **add jitter** — a random offset on each delay.

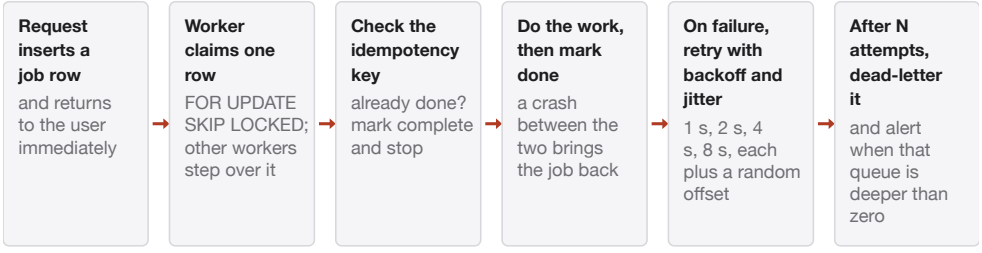
Without jitter, a failing dependency that comes back up is hit by every retry from every worker at the same instant, which knocks it over again. This is a real, repeated pattern in outage reports: the recovery is what causes the second outage. Jitter is one line of code and it prevents it.

The dead-letter queue nobody reads

After N attempts, stop retrying and move the job to a dead-letter queue. Otherwise one malformed message is retried forever, consuming a worker and filling your logs.

A dead-letter queue that nobody looks at is a data-loss queue with a nicer name. **Alert on its depth**, at greater than zero. These are, by definition, things that were supposed to happen and did not.

A job's life, designed for the fact that it will run twice



Postgres with FOR UPDATE SKIP LOCKED is a perfectly good queue for most projects. Whatever the store, delivery is at least once, so every job carries an idempotency key and every retry is backed off with jitter.

The number that tells you if you are keeping up

Not throughput. **Oldest message age.**

Throughput of 500 jobs a minute sounds healthy and tells you nothing about whether you are falling behind. Oldest-message-age says exactly that: if it is climbing, arrivals exceed capacity and the graph is a straight prediction of when it becomes a customer problem. Graph it, alert on it.

Also make the **visibility timeout** — how long a claimed job is hidden before it is offered again — comfortably longer than the job takes. Set it too short and a slow job is handed to a second worker while the first is still running it, which is the duplicate-execution problem arriving on purpose.

Scheduled jobs need a lock

`cron` on one machine is a single point of failure. `cron` on every machine runs the job N times. And during a rolling deploy, both old and new instances may fire the same nightly job.

Take a lock, and let losers do nothing:

```
SELECT pg_try_advisory_lock(918273);
```

The one-machine version also has a subtler failure: the machine is replaced, nobody notices, and the nightly export silently stops. So a scheduled job should record that it ran, and something should alert when the record is miss-

ing — a check on absence, not on failure. Jobs that stop running are much harder to notice than jobs that fail.

BEFORE YOU MOVE ON

Your worker sends a confirmation email, then updates the job row to "done". A deploy restarts workers mid-job, and some customers receive two identical emails. What is the correct fix?

A. Mark the job as done before sending the email, so a crash cannot cause a second send

B. Increase the visibility timeout so a restarted worker does not pick the job up again

C. Disable retries for email jobs, since a duplicate is worse than a missed message

D. Make the send idempotent, by passing an idempotency key to the mail provider or recording the key before sending and checking it first

---

36 · 9 min

The traffic that is not customers

THE ONE THING TO KEEP

Limit per account and per endpoint rather than only per IP address, and put the tightest limits on whatever spends money per request.

What arrives once you are public

Within hours of a public IP existing, it is being scanned. Within days of a domain resolving, forms are being submitted by things that are not people. The mix is consistent:

- Scanners probing for `/wp-admin`, `/.env`, `/.git/config` and a few thousand other paths. `/.env` is the one that matters: an exposed environment

file is a complete compromise, and it is checked because it works.

- Credential stuffing — lists of leaked email and password pairs from other breaches, tried against your login form.
- Signup spam, to use your service, your storage, or your outbound email as a relay.
- Scrapers and crawlers, including a large and growing volume of AI training crawlers, some of which ignore `robots.txt` entirely.
- Somebody's misconfigured client retrying forty times a second because their first request failed.

None of this is targeted. You do not need to be interesting.

Token bucket, and why fixed windows leak

A **fixed window** — 100 requests per minute — allows 200 requests across a two-second boundary: 100 at 11:59:59 and 100 at 12:00:00. For a limit protecting an expensive resource, that doubling matters.

A **token bucket** holds N tokens, refilled at a steady rate. Each request takes one. It permits a burst up to the bucket size and then enforces the long-run rate. It is what you want, it is a few lines of code, and it is what this project's rate limiter uses.

Per-IP is not enough on its own

Layer three limits, because each one fails differently:

- **Per IP** — catches crude floods. But carrier-grade NAT means an entire city can share an address, a university looks like one client, and behind a CDN every request appears to come from the CDN unless you read the forwarded header correctly. Keep this limit loose, or you will block real people in blocks.
- **Per account** — the accurate one for anything a signed-in user does. This is where the real limits belong.
- **Per endpoint** — because `/login` and `/search` and `/api/generate-image` have nothing in common. One should tolerate thousands of requests;

one should tolerate five failures in a row.

And be exact about the expensive endpoints, because rate limits protect the bill as much as the box. Anything that calls a paid API, sends an email or SMS, generates an image, or runs a model needs a hard per-account daily cap, not just a rate. That is what this project's `ai_usage` table does: ten AI chats per person per day during the beta, claimed last and refunded if the provider fails.

Answer with 429, and say when

```
HTTP/1.1 429 Too Many Requests
Retry-After: 30
```

A 429 with no `Retry-After` teaches every well-behaved client to retry immediately, which is the opposite of what you asked for. Include it.

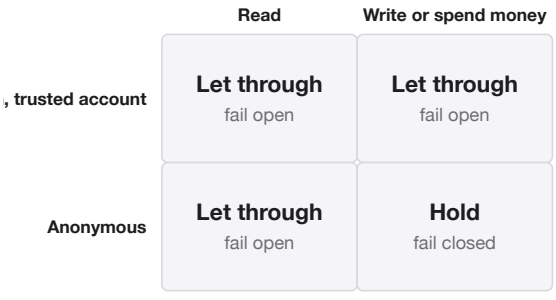
Do not use 403 for rate limiting. A client needs to distinguish "you are going too fast" from "you are not allowed", and so does the person reading your logs at 2am.

Fail open or fail closed?

When the limiter itself is unavailable — the store is down, the timeout fires — you must choose in advance.

The reasonable default is to **fail open for authenticated, trusted users** and **fail closed for anonymous writes**. Losing the limiter should not be an outage for the people you know; it also should not be an open door for the people you do not. This is the same principle the moderation pipeline in this project uses when a classifier tier is degraded: keep trusted members working, hold the untrusted.

When the limiter itself is down



Losing the rate limiter should not be an outage for the people you know, and it should not be an open door for the people you do not. Decide this on a calm afternoon and write it down.

Whatever you choose, write it down. The default is usually "fail open" simply because nobody decided, and that is worth knowing.

Signup abuse specifically

Email verification before anything expensive. **Cloudflare Turnstile** is free and does not make users identify traffic lights; **hCaptcha** has a free tier. Block disposable email domains if it fits your product, and know that the lists are never complete.

One specific warning, because it costs real money: **do not put SMS OTP on an open signup form.** SMS pumping — fraudulent traffic to premium-rate numbers, where the attacker takes a share of the carrier revenue — has cost companies seven-figure sums. If you need SMS, rate-limit it per number and per IP, restrict country codes to those you serve, and cap the daily spend at the provider.

The free path

Cloudflare's free plan gives you basic WAF rules, bot-fight mode and a small number of rate-limiting rules at the edge, which stops the traffic before it reaches your bill. `nginx`'s `limit_req` is free. `fail2ban` on a VM reads your logs and blocks repeat offenders at the firewall.

And put `/.env`, `/.git` and `/wp-admin` in a deny rule. They will never be legitimate, and blocking them at the edge removes a large share of your log noise on day one.

BEFORE YOU MOVE ON

You add a rate limit of 100 requests per minute per IP address. Complaints arrive from users in one city who are being blocked while browsing normally, and separately your login endpoint is still being brute-forced. What is going on?

A. Carrier-grade NAT puts many real users behind one address while the attacker rotates addresses, so a per-IP limit punishes the wrong people and misses the attack
B. The limit is too low, and raising it to 1,000 per minute resolves both problems
C. The rate limiter is using a fixed window, allowing double the intended rate at boundaries
D. The login endpoint is excluded from the limiter by a route-matching bug

37 · 9 min

Files people send you

THE ONE THING TO KEEP

Uploads belong in object storage, written directly by the browser through a short-lived presigned URL with the key, type and size constrained — and everything the client tells you about the file, including its name and content type, is attacker-controlled until you check the bytes.

The local disk is not storage

The first version of file upload always writes to a folder next to the code. It works on your laptop and it fails in production for four separate reasons: a second instance cannot see the file, a container's filesystem disappears on restart, the disk fills and takes the whole machine down with it, and nothing about it is backed up.

The replacement is object storage: a service that holds blobs by key over HTTP. Amazon S3 defined the interface, and almost everything else speaks

the same API — Cloudflare R2, Backblaze B2, DigitalOcean Spaces, Hetzner, Wasabi, and MinIO if you want to run it yourself for free. That compatibility is worth insisting on, because it means moving provider is a configuration change rather than a rewrite.

The price shape is the thing to learn: storage is cheap (roughly 1.5 to 2.5 US cents per gigabyte-month), requests are almost free, and **egress is where the difference is** — the large clouds meter it at several cents per gigabyte, while R2 and B2 charge nothing or almost nothing to serve it out. For a site serving images, that single difference can be the whole hosting bill.

Do not route the bytes through your app

The obvious design has the browser POST the file to your server, which then forwards it to the bucket. It doubles the bandwidth, ties up a worker for the length of a mobile upload, and puts a 200 MB video through a process sized for 2 KB JSON requests.

The standard fix is a presigned URL. Your server authorises the upload and hands back a short-lived URL; the browser sends the bytes straight to the bucket.

```
url = s3.generate_presigned_url(
    "put_object",
    Params={"Bucket": "uploads", "Key": key,
            "ContentType": "image/jpeg"},
    ExpiresIn=300)      # five minutes is plenty
```

Three things to constrain when you sign: the key (so a user cannot overwrite somebody else's object), the content type, and the maximum size — with presigned POST policies you can enforce a content-length range, which is the only way to stop somebody uploading 5 GB to a form meant for avatars.

After the upload, the browser tells your server it finished. Never trust that message alone: check the object exists and its size before you record it, because the client controls what it claims.

What arrives is not what was promised

Everything about an uploaded file is attacker-controlled, including the parts that look like metadata.

- **The filename.** `../../etc/passwd` and a 2,000-character Unicode name are both things you will receive. Never use the supplied name as a path. Generate your own key — a UUID, or a hash of the content — and keep the original name as a display label only.
- **The content type.** The browser sends whatever it likes. Check the actual bytes: `file --mime-type`, or a magic-number library. A file named `.jpg` can be an HTML page, and if you serve it back from your own domain with the type the client claimed, you have a stored cross-site scripting hole.
- **The size.** Enforce it at the edge, at the bucket policy, and in your own record. A limit implemented only in JavaScript is decoration.
- **The contents.** Images can carry EXIF data including the exact GPS coordinates where a photo was taken. If your users are posting personal photographs, strip metadata on ingest — this is a privacy obligation, not an optimisation. `exiftool`, ImageMagick and libvips all do it, all free.

Serve user content from a different domain from your application, or at minimum with `Content-Disposition: attachment` and a strict content type. Same-origin user files are how one uploaded HTML page becomes a session-stealing page on your own domain.

Public buckets, the classic accident

Misconfigured buckets are one of the most reliable sources of data leaks in the industry, and the mistake is almost always the same: making the bucket public because a private URL was inconvenient.

Keep the bucket private and serve reads through presigned GET URLs, or through a CDN configured with a signing key. Both are a few lines. If content is genuinely public — a logo, a course image — a public bucket is fine; the rule is that publicness is a decision per object, not a shortcut taken once at three in the morning.

Derivatives, and the work you did not plan

An uploaded 8 MB phone photo should not be what a visitor downloads. You need thumbnails, and there are three honest options: resize on upload in a background job (predictable, storage-heavy), resize on first request and cache (lazier, needs a cache), or use an image CDN that transforms on the fly by URL parameter. libvips is the fast free library, `sharp` wraps it for Node, and Thumbor and imgproxy are free self-hosted services that do the whole job.

Do the resizing in a queue, not in the request. A large image processed inline is exactly the kind of slow, memory-hungry work that the queues lesson exists to move.

Storage grows and never shrinks by itself

Deleted rows leave orphaned objects. Failed uploads leave partial ones. Set a lifecycle rule on the bucket to expire incomplete multipart uploads after a day and to move old derivatives to a cheaper class, and write a small job that reconciles objects against the database occasionally. Nobody does this until the bill arrives, and the fix is a configuration setting that takes two minutes.

BEFORE YOU MOVE ON

An app stores uploads in a private bucket, generates its own UUID keys, and serves files back from `app.example.com/files/<uuid>` using the content type the browser reported at upload. Where is the hole?

A. UUID keys are guessable, so one user can enumerate another user's files

B. Presigned URLs expire, so the served links will break for users after a few minutes

C. Private buckets cannot be served through an application without making them public

D. A file claiming to be an image can actually be HTML, and serving it from the app's own domain with that type executes script in the site's origin

38 · 9 min

Why your email went to spam

THE ONE THING TO KEEP

Transactional email fails silently, and the three DNS records that stop it failing are the ones the major providers now check on every message.

The failure nobody reports

A user requests a password reset. The email goes to spam, or is dropped by the receiving server without a bounce. They do not tell you. They tell nobody. They leave.

Email is infrastructure with no error message, which is why it deserves a lesson of its own in a course about shipping.

You cannot send it from your own server

Two facts settle this.

Outbound port 25 is blocked by essentially every cloud provider by default — AWS, Google Cloud, Azure, DigitalOcean, Hetzner. That is deliberate, because compromised virtual machines are the internet's main source of spam.

And even with it open, a brand-new IP address has no reputation, which means the large mailbox providers treat it with suspicion for weeks. Building sending reputation is a months-long process that a relay has already done for you.

So use a relay. Do not run your own mail server for transactional email. Running one for your own inbox is a defensible hobby; putting your password resets behind it is not.

The three records

SPF — a TXT record on your domain listing who may send on your behalf.

```
v=spf1 include:_spf.example-provider.com -all
```

There is a hard limit of 10 DNS lookups when evaluating SPF, and each `include:` may cost several. Add four providers and SPF silently fails with a permerror, which is a genuinely common cause of "it worked until we added the newsletter tool".

DKIM — a cryptographic signature on each message, with the public key published in DNS at a selector your provider gives you. This is what proves the message was not forged or altered, and it survives forwarding, which SPF does not.

DMARC — a policy saying what to do when SPF and DKIM fail, plus an address to send reports to.

```
v=DMARC1; p=none; rua=mailto:dmarc@example.com
```

Start at `p=none`, which changes nothing and only collects reports. Read them for two weeks — they will show you every system sending as your domain, including the two you forgot. Then move to `p=quarantine` and later `p=reject`. Going straight to `p=reject` is how people stop their own invoicing system from delivering.

Since February 2024, Google and Yahoo require all three from bulk senders, plus one-click unsubscribe on marketing mail and a spam complaint rate kept under 0.3%. That is not a guideline; mail that fails it is rejected.

Check any domain from a terminal:

```
dig +short TXT example.com | grep spf
dig +short TXT _dmarc.example.com
```

Authentication is necessary and not sufficient

The rest is behaviour, and it is where most deliverability is actually won or lost:

- **Send from a subdomain.** `mail.example.com` for transactional, a different one for marketing. Then a badly received campaign cannot damage the reputation carrying your password resets. Keep them separate from the first day; separating them later is much harder.
- **Warm up gradually.** A new sending domain going from zero to 50,000 messages in a day looks exactly like a compromised account.
- **Never buy a list.** Spam traps are seeded addresses that exist only on purchased lists, and hitting one gets you blocked outright.
- **Honour unsubscribes immediately**, and make the link obvious. A complaint is far more damaging than an unsubscribe — a user who cannot find the unsubscribe link presses the spam button, and that is the metric that gets you blocked.
- **Set a real `Reply-To`.** `no-reply@` addresses depress engagement, and engagement is a ranking signal.

Process bounces and complaints

Your provider sends webhooks for delivery, bounce and complaint events. Handle them, and store the result.

A **hard bounce** means the address does not exist. Continuing to send to it is one of the strongest negative signals there is. Mark it dead and stop. A **soft bounce** is temporary — a full mailbox — and can be retried a few times. A **complaint** means somebody pressed spam; suppress that address permanently, whatever your records say about consent.

Testing it

Send to a real Gmail account and open "Show original". You want to see `SPF: PASS`, `DKIM: PASS` and `DMARC: PASS`. **mail-tester.com** is free, gives you a score out of 10 and names what is wrong. Do this before launch, not after the first support ticket.

In development, do not send at all. **Mailpit** or **MailHog** — one binary each, free — capture everything and show it in a web UI.

The free path

Brevo has a free tier of a few hundred messages a day; **Resend** and **Mailgun** have small free tiers; **Amazon SES** is the cheap option at large volume and gives you almost no hand-holding; **Postmark** costs more and is the most reliable for transactional mail specifically. Free tiers move constantly, so check before you build against one, and keep the sending code behind a small interface so switching provider is an afternoon rather than a project.

BEFORE YOU MOVE ON

Your password reset emails deliver to Gmail for months, then start landing in spam shortly after marketing launches a newsletter from the same domain. Authentication records are unchanged and still pass. What is the most likely cause?

A. Gmail changed its filtering rules, and there is nothing you can do but wait

B. The newsletter tool was added to SPF, pushing it past the 10-lookup limit

C. The newsletter's complaint rate has damaged the domain's sending reputation, which the transactional mail shares because both send from the same domain

D. The transactional volume is too low to maintain reputation on its own

CASE STUDY · MODULE 4

The bill from an endpoint nobody was told about

A two-person handicrafts export business in Jaipur, nine days into the festival season, reading a cloud invoice that is forty-five times last month's.

Meera sells brass and blue pottery to buyers abroad. The site is a catalogue with an enquiry form, and one feature she is proud of: a box where a visitor pastes a few specifications and gets a product paragraph written for them, in English, by a model API. It was meant for her own use, writing listings. She left it open on the site because it was interesting, and because visitors who used it were about three times as likely to send an enquiry.

The route required no account. It took a text field and called the paid API.

For five weeks it was used thirty or forty times a day and cost around nine hundred rupees a month. On the ninth of October the provider's dashboard projected forty-one thousand rupees for the month. The budget alert she had set at five thousand had arrived fourteen hours earlier, at two in the morning, in an inbox she reads once a day.

The logs told a plain story. From the fourth of October, a steady eleven requests a second, from a rotating set of addresses, each sending a prompt that had nothing to do with pottery. An unauthenticated route that calls a paid API is a free paid API for the entire internet, and it will be found by scanners within days of going live. Hers had been live for five weeks.

While she was in the invoice she found a second line she had never read. Egress: two point one terabytes. The catalogue's hero image was a four megabyte photograph of a haveli courtyard, served from the origin at full size to every visitor on every page, and October's traffic was a festival campaign.

So two costs, arriving together, in the fortnight that carries most of the year's revenue. They are worth separating, because they have nothing in common except the invoice. The egress is a consequence of a file being larger than it needs to be, served from further away than it needs to be, and it will keep costing money in proportion to her success. The model spend is somebody

else using a paid endpoint she left open, and it is not proportional to anything — it is whatever the script decides to do next.

She had also, without meaning to, given the abuser a discount. There was no maximum output length on the request, so every call generated as much text as the model would produce, and output tokens are priced at three to five times input tokens on the plan she was on. A cap of two hundred tokens would not have stopped the abuse and would have cut its cost by most of it.

The decision she had to make that morning was about the description box, and it was not obvious.

Turn it off now. The meter stops within a minute. It also removes the feature during the highest-converting fortnight of her year, on a site whose whole business is enquiries from strangers.

Leave it up and add a limit per IP address today. It is an hour of work. It is also weak in both directions: whoever is doing this is rotating addresses for a fraction of a paisa each, and a college or an office in a country she sells to shares one address between hundreds of people, so a tight per-address limit refuses a crowd of real buyers to inconvenience one script.

Leave it up and build the real control. An account, email verification, a daily quota per person, a hard cap on output length, a monthly ceiling at the provider. Two evenings of work — during which the meter keeps running at roughly four and a half thousand rupees a day.

> STOP HERE
>
> Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

She turned it off within the hour, and the rule she wrote in her notes afterwards was: prefer a cap that breaks a feature over an uncapped bill you have

to pay in full. The forty-one thousand rupees was payable; the provider refunded nothing, and was not obliged to.

Over the next two evenings she rebuilt it. The box now needs a free account with a verified email. There is a quota of fifteen generations per account per day, counted in a table, claimed before the call is made and refunded if the provider fails, so a provider error does not silently spend somebody's allowance. Every request carries a hard maximum output length, because without one a single pathological input generates the model's maximum. There is a monthly spend ceiling at the provider itself, with an alert at half of it — the alert tells her the money is going, the ceiling is what stops it. And the limiter is a token bucket per account rather than a fixed window, so a burst is allowed and the long-run rate is not.

The feature reopened four days later. Enquiries recovered within a week.

The egress line took twenty minutes. The hero image became a WebP at one hundred and eighty kilobytes, and Cloudflare's free plan went in front of the site. Her cache hit ratio on the first day was thirty-eight per cent, which she traced to a framework default sending `Set-Cookie` on every response, including the catalogue pages that are the same for everybody. Most content delivery networks refuse to cache a response carrying a `Set-Cookie` header, which had silently disabled caching for the whole site. With that removed the ratio settled at ninety-six per cent and the egress line effectively disappeared.

She had a budget alert and it fired. Why did it not prevent the bill?

An alert is a notification, usually delayed, sent to a person who may be asleep — hers arrived at two in the morning and was read fourteen hours later, by which time most of the money had gone. Alerts tell you the money is leaving; caps stop it. The controls that act without you are a maximum concurrency, a hard monthly ceiling on the key at the provider, a per-account quota enforced in code, and a bounded queue in front of expensive work.

Why was a per-IP limit the wrong primary control for this particular endpoint?

Because it fails in both directions. Addresses are cheap to rotate, so the abuser pays almost nothing to get around it, while carrier-grade NAT and institutional networks put hundreds of legitimate people behind one address, so a limit tight

enough to matter refuses real buyers in blocks. Anything that spends money per call belongs behind an identity: a verified account with a hard daily cap, with a loose per-address limit only as a blunt first layer.

A cache hit ratio of thirty-eight per cent on a catalogue of static product pages. What is the most likely cause, and why does it hide so well?

A `Set-Cookie` header on responses that should be anonymous, usually from a framework session default. Most CDNs refuse to cache any response carrying one, because a cached response with somebody's session cookie in it hands that session to the next visitor. It hides well because nothing is broken: the pages render, the site is fast for you, and the only visible symptom is a number on a dashboard nobody reads and an origin bill nobody attributes to it.

MODULE 4 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A page that greets the signed-in user by name is served with `Cache-Control: public, max-age=300`. What is the failure?

A. The CDN stores one user's page and serves it to whoever asks next

B. The browser keeps a stale greeting for five minutes after signing out

C. The CDN refuses to cache it, so your hit ratio silently drops to zero

D. Each user gets a separate cache entry, so the origin load is unchanged

2 A popular cache key expires and ten thousand requests recompute the same value at once, repeatedly. Which one-line change prevents the recurrence?

A. Increase the TTL from five minutes to an hour

B. Add a random offset of up to sixty seconds to each TTL

C. Serve the stale value and refuse new writes while recomputing

D. Move the cache from the application into the CDN layer

3 You want to serve 200 requests per second at 100 ms each. What does Little's Law tell you to size?

A. Twenty concurrent requests in flight, so at least twenty pool slots

B. Two hundred connections, one for each request arriving per second

C. Two workers, because each can serve ten requests per second

D. One hundred slots, matching the latency figure in milliseconds

4 Traffic is past the knee of your capacity curve. Why is shedding load better than queueing it?

A. A queue costs memory, and memory pressure invites the OOM killer
B. Queued requests are served out of order, which corrupts session state
C. Beyond the knee, queueing serves everybody slowly and helps nobody
D. Rejected requests are retried by clients, which spreads the load out

5 Your serverless functions produce `FATAL: sorry, too many clients already` under load. What is the fix?

A. Raise max_connections, since each connection needs only a few megabytes
B. Reduce each function's pool to one connection and retry on failure
C. Move the read queries to a replica so the primary has spare connections
D. Put a pooler in front, so many clients share a few real connections

6 A user uploads a file called photo.jpg whose bytes are an HTML page. You serve it back from your own domain with the type the browser claimed. What have you built?

A. A page on your own origin that can read your users' sessions
B. A broken image that fails to render in every browser
C. An extra origin request that will always miss the CDN cache
D. A file your antivirus scanner will quarantine on the next pass

MODULE 5

Staying alive: the bad day and the day after

The last module is about time. Things expire, data accumulates obligations, and eventually something breaks while people are using it. These are the habits that decide whether that is forty minutes or a weekend, and what is left behind for whoever runs this next.

> BY THE END OF THIS MODULE YOU CAN
>
> Run the first thirty minutes of a real outage from a written runbook, and leave behind a postmortem and a handover that change what happens next time

---

39 · 8 min

The restore you have never tested

> THE ONE THING TO KEEP
>
> An untested backup is a guess; the drill is the only thing that turns it into a plan.

A backup you have not restored is a hypothesis

That is the whole lesson, and the rest is detail. You do not have backups. You have files you believe are backups, and belief is untested until the day it is tested for you.

What makes a copy a real backup

The usual advice is 3-2-1: three copies, two kinds of media, one off-site. For a small project the part that actually matters is narrower:

At least one copy must live outside the blast radius of the thing it protects. If the same credentials that can drop your database can also delete the bucket holding its backups, then one compromised key, or one confident `terraform destroy`, ends the company. A different account, ideally a different provider, with write-only or append-only access from the app side.

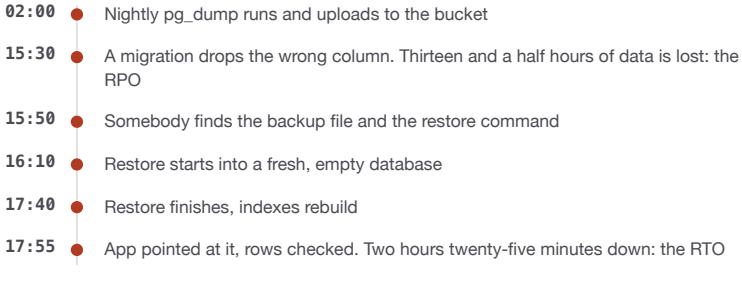
Two words worth knowing

RPO — how much data you can afford to lose. A nightly dump means your RPO is 24 hours. If the disaster lands at 15:30 and the dump ran at 02:00, thirteen and a half hours are gone.

RTO — how long you can be down. A 40GB dump does not restore in five minutes. It can take one to three hours, plus index rebuilds, plus the time spent finding the file and remembering the command.

Write both numbers down. If your honest RPO is "an hour" and you have nightly dumps, you have not made a decision — you have avoided looking.

One bad afternoon with a nightly dump



Everything written between the dump and the disaster is gone: that gap is the RPO. The time from disaster to a working site is the RTO, and it is nearly always two to five times what was guessed, because finding the file and remembering the command are part of it.

Point-in-time recovery is the alternative: the database keeps its write-ahead log, so you can restore to 15:29:58, one second before the bad migration. Managed Postgres usually offers around 7 days of PITR on paid plans

and often none on free ones. Go and check today which you have. "The provider handles it" is not an answer; the retention number is.

The drill

Put it on a calendar. Quarterly is plenty for a small project.

1. Take the real backup file, from where it really lives.
2. Restore into a fresh, empty database.
3. Point a copy of the app at it and open a page.
4. Check specific rows: your own account, the newest row in the busiest table.
5. Time the whole thing and write the number down. That is your RTO, and it is usually two to five times what you guessed.

Every team that does this for the first time finds at least one of these:

- The dump has been zero bytes since a flag changed in March.
- The cron ran fine, the upload failed silently, and nobody checked the exit status.
- `pg_dump` from version 16 refuses to restore into version 15.
- The backup is encrypted and the key lived on a server that no longer exists.
- Nobody actually knows the restore command.

What is not a backup

- **A read replica.** It faithfully replicates your `DELETE FROM users` in under a second.
- **Provider redundancy or RAID.** Protects against a disk dying. Offers nothing against you.
- **Soft deletes.** Protects against a user clicking the wrong thing. Offers nothing against a migration.

All three protect against hardware. Almost nothing that destroys a small project's data is hardware.

Enough, and cheap

A nightly `pg_dump | gzip` to a bucket at a different provider, thirty days of retention, and — this is the part people skip — **an alert when today's file is missing or smaller than a plausible threshold.** "Backup job succeeded" logs lie; file size does not. Add the quarterly restore drill. Storage costs a few cents to a few dollars a month.

That is a real backup system, and it fits in about fifteen lines of shell.

BEFORE YOU MOVE ON

A team runs a nightly `pg_dump` to object storage. Monitoring confirms the job exits 0 every night and a file of roughly the expected size appears. On a Tuesday afternoon a bad migration wipes a table. What is the most likely way this still goes badly?

A. Nobody has ever performed a restore, so the first attempt happens under pressure and whether the dump is complete and loadable is still unknown

B. The exit code is unreliable, because a shell pipeline reports the status of the last command and can hide a `pg_dump` failure

C. They lose up to a day of data because the dump is nightly, and that is the core failure

D. Object storage versioning will cause the restore to pull an older copy than intended

40 · 10 min

The short list that actually matters

THE ONE THING TO KEEP

Small sites are compromised through leaked credentials, exposed services, missing authorisation checks and unpatched dependencies, so spend your effort there before anywhere else.

Nobody is attacking you personally

That is the useful frame. Almost nothing that happens to a small site is targeted. It is automated, indiscriminate, and looking for a short list of known-easy things. Which means a short list of defences covers most of the risk, and the exotic material you read about is not where your effort belongs yet.

The machine

If you run your own virtual machine:

- **SSH keys only.** `PasswordAuthentication no`, `PermitRootLogin no`. Then look at the logs a week after launch — `journalctl -u ssh | grep -c "Failed password"` on a fresh public box regularly runs to thousands per day. That is the ambient rate, arriving at every address on the internet.
- **A firewall with three ports open**: 22, 80, 443. Everything else closed by default.
- **Your database must not have a public IP.** "It has a password" is not a plan. Shodan and similar services index open Postgres, Redis and MongoDB instances continuously, and unauthenticated Redis in particular has been mass-exploited for years. Bind to a private network, or to localhost, and reach it through the application or an SSH tunnel.
- **Automatic security updates.** `unattended-upgrades` on Debian or Ubuntu. It is one package and it closes the window on the kernel and OpenSSL issues you will otherwise never patch.

The accounts

This is where the highest-value five minutes live, and it is not code.

Multi-factor authentication on: your **domain registrar**, your **DNS provider**, your **email**, your **cloud account**, your **GitHub**. The registrar and the email are the crown jewels — with control of your email, an attacker resets everything else; with control of your DNS, they receive your traffic and can issue a valid certificate for your domain.

Use a password manager, use unique passwords, and store the MFA recovery codes somewhere a second person can reach them. Locking yourself out is a more likely outcome than being attacked.

The dependencies, and an honest word about the noise

Turn on **Dependabot** or **Renovate**, both free. Run `npm audit` or your language's equivalent in CI.

The honest part, which most advice omits: a large fraction of CVE alerts in a lockfile are for code paths you never execute — a vulnerability in the dev server of a build tool, or in a function your dependency does not call. If you treat every alert as urgent you will spend your week on it, and then you will start ignoring the feed, and then you will miss the one that matters.

So triage. Is this package in the running application, or only in the build? Is the vulnerable function reachable from your code? Is it remotely triggerable? Fix the reachable ones promptly, batch the rest monthly, and keep updating regularly so no single update is frightening.

The application, in four items

Parameterised queries, always. SQL injection is decades old and still near the top of every breach report, and it is still string concatenation. Your ORM does this for you until the moment somebody builds a `WHERE` clause with a template literal.

Authorisation checked per object, on the server. The most common real bug in small applications is not injection; it is `/invoice/1234` where

changing the number to 1235 shows somebody else's invoice. Every handler that loads a record by ID must check that this session is allowed that record. Hiding the link in the UI is not a check.

Cookies set properly. `HttpOnly` so JavaScript cannot read the session, `Secure` so it is never sent over HTTP, `SameSite=Lax` so other sites cannot make authenticated requests on the user's behalf.

A rate-limited login, from the previous module, plus a check of submitted passwords against known-breached lists. Credential stuffing is the highest-volume real attack on a small site.

Least privilege for the database user

The account your application connects with does not need to create tables, drop them, or read other databases. Give it `SELECT`, `INSERT`, `UPDATE`, `DELETE` on the tables it uses. Migrations run as a different, more powerful role, in a job you control.

This costs ten minutes and converts several categories of injection from catastrophe into incident.

Headers, and the one with teeth

Most security headers are one line each and worth adding. `Content-Security-Policy` is the one that actually prevents cross-site scripting, and the one people get wrong, because a strict policy breaks inline scripts and third-party embeds.

Deploy it in report-only mode first (`Content-Security-Policy-Report-Only`), collect violations for a week, then enforce. Going straight to enforcement on a live site breaks your own analytics tag on a Friday afternoon.

Free tools, all of them

`ssh-audit` for your SSH configuration, `testssl.sh` for your TLS, **OWASP ZAP** for a basic application scan, and Mozilla's Observatory for headers. None

of them cost anything, and running each once before launch is an afternoon that puts you ahead of most sites on the internet.

BEFORE YOU MOVE ON

You have limited time before launch. Which of these gives the largest reduction in real risk for a small public site?

A. A strict Content-Security-Policy, enforced from day one

B. Multi-factor authentication on the registrar, DNS, email and cloud accounts, and no public IP on the database

C. An automated vulnerability scanner running weekly against the application

D. Resolving every dependency alert reported by `npm audit` before launch

---

41 · 9 min

Data you must keep, and data you must delete

THE ONE THING TO KEEP

Every store of personal data is a liability with a lifetime, so decide the number for each one and enforce it with a job rather than a policy document.

You are collecting more than you think

A live site accumulates personal data whether anyone designed for it or not:

- IP addresses and user agents in web server logs.
- Email addresses, obviously — and also in support threads, in error reports, in a CSV somebody exported.
- Uploaded images with EXIF metadata, which routinely contains GPS coordinates and a device serial number.
- Session records, password reset tokens, login history.
- Whatever your analytics provider set in a cookie.

The first move is not a policy. It is to write down what you have, where it lives and why. That list is usually shorter than feared and longer than expected, and you cannot manage what you have not enumerated.

Collect less

The cheapest way to not leak something is to not have it.

You do not need a date of birth to run a forum. You do not need a full address to send an email. You do not need to store the raw IP when a truncated one or a daily-rotated hash answers the same question. Every field you decline to collect is a field that cannot appear in a breach, cannot be subpoenaed, and cannot be asked about in a data request.

Retention is a number, enforced by a job

"We keep data as long as necessary" is not a retention policy; it means forever. Pick a number for each store and make something delete it:

- Web server and application logs: 14 to 30 days.
- Anything request-level containing an IP or user ID: shorter, 7 days is common.
- Backups: 30 to 90 days, on a rolling schedule.
- Soft-deleted records: purged after 30 days.

Then write the job, schedule it, and alert if it stops running. A retention policy that lives only in a document is a claim you are making publicly and not honouring, which is materially worse than not making it.

Deletion has to reach every copy

When someone asks to be deleted, the row in your primary database is the easy part. The copies are:

- Read replicas — they follow, so this is fine.
- **Backups.** The one everybody forgets.
- The search index.

- The cache.
- The CDN, if user content is served through it.
- Your analytics and error-tracking providers, which are separate systems with their own deletion APIs.
- The CSV someone exported to a laptop last March.

The honest position on backups is that you cannot surgically remove one person from a backup archive without destroying its integrity. The accepted answer is to state the retention window plainly: deleted from live systems immediately, gone from backups within N days as the backups age out. Write that N in your policy. A specific number is both honest and defensible; silence is neither.

Soft delete, hard delete, and the lawful exception

Soft delete — a `deleted_at` column — keeps referential integrity, keeps replies from orphaning, and keeps an audit trail. It is the right default for content.

It does not satisfy "erase my personal data", because the data is still there. So you need a real erasure path: anonymise the identity, replace the display name and email with placeholders, drop the avatar, and keep the content rows if they must stay for others' sake.

And there are lawful exceptions worth being precise about rather than vague. This project keeps a `legal_holds` record for content it is legally obliged to preserve and report; that record deliberately survives deletion of the account, and its author reference is intentionally not a foreign key so that erasing the identity does not remove it. That is retention under a legal obligation, and it is a recognised exemption from erasure.

The lesson generalises: **name your exceptions in your policy.** "We delete everything on request, except records we are legally required to retain, which are X and kept for Y" is honest and correct. "We delete everything" while quietly keeping something is neither.

The rules of thumb, without a lawyer

You do not need legal advice to be broadly right:

- Under the GDPR you have **one month** to answer an access or erasure request, and **72 hours** to notify a supervisory authority of a personal data breach. India's DPDP Act 2023 is different in detail and similar in shape.
- Tell people what you collect, why, how long, and who else sees it. In plain language, on a page anyone can reach.
- Give them a working button, not an email address that goes to somebody who left.

That is most of it. The failure mode for small teams is not usually the wrong retention period; it is a privacy page that describes a system nobody built.

Never log the secret

Redact at the logging layer, not by remembering: `Authorization` headers, cookies, passwords, tokens, OTP codes, full card numbers. Configure the redaction in one place, because "we will be careful" fails the first time somebody logs an entire request object during a debugging session and it goes to production.

BEFORE YOU MOVE ON

A user asks you to delete their account. You delete their rows from the primary database. Which part of a correct answer are you most likely to have missed?

A. The read replicas, which may still hold their rows

B. The soft-delete flag, which leaves the row present in the table

C. Their session records, which will expire on their own

D. The backups, the search index and the third-party services, none of which are reached by deleting the row — and the backup window is the number your policy should state

42 · 8 min

Everything expires, and nothing tells you

THE ONE THING TO KEEP

Most preventable outages are something with a date on it that nobody owned, so keep one file listing what expires, when, and on whose card.

The least glamorous lesson here

It prevents more downtime than anything in this course except backups.

A live product is held up by a dozen things that have expiry dates. None of them fail because of load or bugs. They fail on a specific day, usually to nobody's surprise except everybody's.

The list

The domain. The one that ends the product rather than degrades it. The usual sequence: the card on file expired, auto-renew failed silently, the notification went to an address nobody reads. After expiry there is typically a grace period of about 30 days during which you can renew normally, then a **redemption period** where recovery costs a restore fee — commonly $80 to $150, against a renewal of $11. After that it can be registered by anyone, and domains that have carried traffic are bought within minutes.

Register for several years at once, turn on auto-renew, keep the card current, and turn on the registrar's transfer lock (`clientTransferProhibited`) so nobody can move it away.

The TLS certificate. Covered in module one. Worth repeating here because Let's Encrypt no longer sends expiry warnings, so a broken renewal is now completely silent until the site goes red.

OAuth client secrets and API keys. Google OAuth client secrets, Apple's signing keys, some payment provider keys — several have hard expiry dates

measured in months. When they expire, sign-in stops working for everybody at once, on a date that was known a year in advance.

Every card on file. One card, five vendors, one expiry date. When it expires you get five failures across a week, in the order the vendors happen to bill.

Free tiers and trial credits. A cloud trial with $300 of credit ends. A "free for 12 months" tier starts charging. A database free tier changes its terms. This is the availability half of what the billing lesson covered — some services suspend rather than bill.

Deprecated APIs with a sunset date. The provider emailed the address on the account, which is the personal address of somebody who left in March.

Your own monitoring's free tier. The check that would have told you stopped running, quietly, and its silence looks exactly like health.

The fix is one file and one calendar

Nothing sophisticated. A `RENEWALS.md` in the repository:

```
domain addaly.com   registrar: X   renews 2027-03-14  auto-re-
new ON  card ending 4412
TLS                 Let's Encrypt  auto, 90d          monitor:
cert-expiry check
Google OAuth secret expires 2027-01-08                rotate
procedure: docs/…
Cloud account       card ending 4412                  expires
06/28
Mail provider       free tier 300/day                 review
2027-01
```

Then a calendar event 30 days before each, and — this is the part that makes it work — **use a role address, not a person's.** Every vendor account registered to `ops@yourdomain`, forwarded to whoever is currently responsible. Personal addresses are how the sunset email reaches nobody.

Two checks worth automating today

```
# domain expiry
whois addaly.com | grep -iE 'expir'

# certificate expiry
echo | openssl s_client -connect addaly.com:443 -servername ad-
daly.com 2>/dev/null \
  | openssl x509 -noout -enddate
```

Wrap both in a scheduled job that alerts at 45 days for the domain and 21 for the certificate. It is twenty lines and it retires two of the most avoidable outages there are.

The habit underneath

When you add anything that has a date — a key, a card, a plan, a contract — the same moment you add it, write the line in the file and make the calendar entry. Not later. Later is where these live until they expire.

This is a small discipline that reads as bureaucratic and is not. It is the difference between a product that quietly keeps working and one that has a bad Tuesday every few months for reasons that were all knowable in advance.

BEFORE YOU MOVE ON

Your site goes down completely. DNS lookups for your domain return nothing, the servers are healthy, and no deploy has happened for two weeks. What should you check first?

A. Whether the domain registration lapsed, most likely because auto-renew failed on an expired card

B. Whether the TLS certificate expired, since that also takes the whole site offline

C. Whether the DNS provider is having an outage affecting your zone

D. Whether a recent DNS record change propagated incorrectly

43 · 9 min

Timeouts, retries, and the failure you made worse

THE ONE THING TO KEEP

One dependency without a timeout can take down every page you serve, and retries without backoff, jitter and a breaker turn a small fault into a storm you generated yourself.

Your service is only as available as the things it calls

A handler that calls a payment provider, a mail API and a model endpoint has four ways to fail, not one. If each dependency is up 99.9% of the time and you need all of them, your ceiling is about 99.6% before you have written a single bug.

Most of the difference between a service that degrades and one that collapses is three settings and one decision. The settings are timeouts, retries and backoff. The decision is what to do when the answer never comes.

Every outbound call needs a timeout, and most do not have one

The defaults are worse than people assume. Python's `requests` waits forever unless you pass `timeout=`. Node's `fetch` has no default timeout. Many database drivers will wait indefinitely for a connection. "Forever" is not a hypothetical: a network can drop packets in a way that leaves the socket open and silent, and your worker sits there until something else kills it.

What that does to a live site is specific and worth picturing. Each stuck request holds a worker. Requests keep arriving. Within a minute every worker is waiting on the same dead dependency, and your site is down — including all the pages that never needed that dependency at all.

Set a timeout on every outbound call, and make it smaller than the time your own caller is willing to wait. If your page must answer in 3 seconds, a dependency cannot have a 10-second timeout.

```
resp = session.post(url, json=payload, timeout=(2, 5))  # con-
nect, read
```

Then decide, per call, whether a timeout means failure or a fallback. A recommendation service that times out should return the popular items. A payment that times out must not be treated as success.

Retries help, and then they attack you

Retrying a failed call is usually right — most failures are transient. Three rules keep it from becoming the outage:

1. **Only retry safe operations.** GETs, yes. A POST that charges a card, only with an idempotency key, so the second attempt returns the first result rather than charging again.
2. **Back off exponentially, with jitter.** Fixed-interval retries from a thousand clients synchronise into waves. Random jitter spreads them. The formula is `min(cap, base * 2**attempt) * random()`.
3. **Cap the total.** Three attempts, then fail. A retry budget — refuse to let retries exceed a small percentage of total requests — is the version that survives contact with real traffic.

The failure mode to understand is the retry storm. A dependency slows down, every caller retries, the dependency now receives three times its normal load while already struggling, and it never recovers. The retries did that, not the original fault. This is why a struggling system sometimes only recovers after clients are stopped entirely.

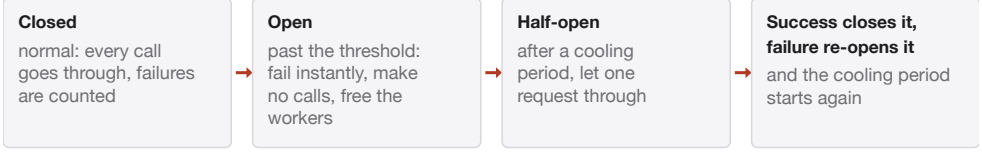
Retries multiply through layers, too. Three retries at the browser, three in your gateway and three in your service is 27 requests for one user action. Retry at one layer, deliberately chosen — usually the innermost one that knows whether the call is safe.

The circuit breaker

When a dependency is clearly broken, calling it again is a way of spending your own capacity to learn something you already know. A circuit breaker counts recent failures and, past a threshold, stops calling for a while, failing instantly instead.

Three states, and the third is the one people miss: closed (normal), open (failing fast, no calls made), and half-open — after a cooling period, let a single request through. If it succeeds, close; if not, stay open. Without the half-open state you have an on-off switch that never turns itself back on.

A circuit breaker's three states



Open turns a thirty-second hang for every user into an instant, cheap failure. Half-open is the state people forget, and without it the breaker is a switch that never turns itself back on.

A breaker turns a 30-second hang for every user into an immediate, cheap failure that frees your workers to serve everything else. Libraries exist in every language; a counter, a timestamp and twenty lines also work.

Decide in advance what degraded looks like

For each dependency, write down the answer to "what does the user see when this is down":

- Search unavailable → show the browse page, with a line saying search is temporarily down.
- Recommendations unavailable → show the most recent items.
- Mail provider unavailable → queue the message, tell the user it will be sent, retry in the background.
- Payment unavailable → say so plainly, and do not create an order in an ambiguous state.

A feature flag per dependency makes this operational rather than theoretical: during an incident you turn off the broken feature and the rest of the site works. That is often the fastest available fix, faster even than a rollback.

The two-line summary

Timeouts stop one slow dependency from consuming your service. Backoff, jitter and breakers stop your recovery attempts from becoming the second outage. Both are configuration you write on a calm afternoon, and neither can be added while everything is on fire.

BEFORE YOU MOVE ON

A downstream API slows from 100 ms to 4 s. Your service retries three times with no delay, and within a minute your own error rate is near 100% even for endpoints that never call that API. What is the mechanism?

A. The downstream API is rate-limiting your retries, so all calls now fail

B. The connection pool to the downstream API has leaked sockets, exhausting file descriptors

C. Retries without jitter synchronise into waves that overload your own event loop

D. Every worker is occupied waiting on the slow call and its retries, so no capacity remains for any request

44 · 8 min

The outage that is not yours

THE ONE THING TO KEEP

When the broken thing belongs to somebody else, your work is to prove it quickly, degrade or serve stale rather than wait, and communicate from infrastructure that does not share the failure — and most small projects should buy cheap redundancy for stateless providers rather than an architecture they cannot operate.

Sooner or later it is not your fault

At some point the thing that is broken is a certificate authority, a DNS provider, a cloud region, a payment gateway or an identity provider — and nothing in your repository can fix it. This has happened at industry scale several times: a configuration change at a single CDN taking a large share of the web offline for an hour, a DNS provider under attack making dozens of well-known services unreachable, a cloud region failing and taking down the products built on it, including, memorably, some of the status pages meant to report the failure.

Your job during those hours is not to fix somebody else's system. It is to know quickly that it is theirs, to keep as much of your product working as you can, and to tell your users something true.

Establishing that it is them, in five minutes

A rough order that usually settles it:

1. **Check from outside your own network.** A phone on mobile data, or a free external checker. Half of "the site is down" reports are one office's wifi or one ISP's DNS resolver.
2. **Check the dependency directly.** `curl` its health endpoint or an unauthenticated URL from your server. Time it.

3. **Check their status page and their social feed.** Status pages lag, often by twenty minutes, because a human has to publish. The social feed is usually faster, and third-party outage trackers faster still.
4. **Check your own dependency dashboard** — the second screen from the dashboards lesson exists for precisely this moment.

Write the answer down with a timestamp as you go. In the postmortem, "we knew it was the provider at 14:07" is a much better sentence than "we think it was maybe an hour".

What you can do, ranked by how much it actually helps

- **Degrade the feature, keep the site.** The failing-well lesson's flag per dependency pays for itself here. Turning off search or recommendations is a ten-second action that keeps everything else alive.
- **Queue the work.** If the dependency is an email or webhook target, buffer and retry later. Users do not need those to be synchronous, and they rarely notice a twenty-minute delay.
- **Serve stale.** A cache configured with `stale-if-error` will keep answering from an old copy while the origin or an upstream is failing. This one line of configuration converts a total outage into slightly out-of-date content.
- **Switch provider, if you already can.** A second mail provider or model provider behind one interface is a genuinely cheap redundancy, because both are stateless HTTP APIs. This is realistic for mail, SMS, models and image processing.
- **Wait, and communicate.** For DNS, identity or your database's region, there is often no alternative and pretending otherwise wastes the hour.

What is worth being redundant, and what is not

An honest ranking for a small project:

Worth it, because it is cheap: a second DNS provider (the protocol supports multiple authoritative servers), a second mail sender, a second model or

API provider behind your own interface, backups stored at a different company from the database.

Not worth it for most: multi-region active-active, multi-cloud, a standby database on another provider. Each of these multiplies your cost and, more importantly, your complexity — and complexity causes outages. Teams that adopt multi-region for reliability often find their new failure modes outnumber the ones they removed. Single-region with good backups and an honest recovery time is a defensible choice, and saying so plainly is better engineering than an architecture you cannot operate.

The question to ask is not "could this fail" but "if this fails for four hours, what does it cost us, and is that more than the standing cost of the alternative".

Say something, and say it somewhere they can reach

A status page hosted on the same infrastructure as your product is not a status page. Use a separate provider — several are free at small scale — or a static page on different hosting, or a social account. Whatever it is, it should be linked from your normal site so people know where to look before they need it.

The message during a third-party outage should say what is affected, what still works, that the cause is upstream, and when you will next update. It should not blame the vendor by way of excuse. Your users chose you, not them; the dependency was your decision.

Afterwards

Write it up like any other incident. The useful actions are almost never "switch provider" and almost always the smaller ones: add the flag that would have let you degrade, set `stale-if-error`, move the status page off the affected infrastructure, add the dependency to the dashboard you check first. Those are what shorten the next one.

BEFORE YOU MOVE ON

Your image-processing provider is down. Your site returns 500 on every page because the header renders user avatars through that provider. What is the most effective immediate action?

A. Turn off the avatar feature behind a flag and serve a placeholder, restoring the rest of the site
B. Fail over to a second image provider configured during the incident
C. Increase the timeout on the provider calls so requests have time to succeed
D. Publish a status update explaining the upstream outage and wait for recovery

45 · 10 min

The first thirty minutes of an outage

THE ONE THING TO KEEP

Stop the bleeding before you understand the cause, because diagnosis works fine from logs and a rolled-back artifact while the fire does not need to keep burning.

The order matters more than the skill

Under pressure, people debug first and communicate never. Both are backwards. Here is a sequence that works, and it is worth reading before you need it, because you will not be reading anything at the time.

1. Confirm it, and scope it — two minutes

"The site is down" from one person is sometimes their office wifi or their DNS resolver. Check from outside: your phone on mobile data, a different network, your external uptime monitor, a public checking service.

Then scope it. All users or some? All pages or one? All regions? Signed-in only? The scope is the strongest clue you will get in the first ten minutes — "only signed-in users, only in Europe" points somewhere very specific, and it costs a minute to establish.

2. Say something — within five minutes

One line, early, buys you an hour of patience:

> *We are aware of errors affecting logins and are investigating. Next update in 20 minutes.*

Three properties make it work: what is affected, that a human is on it, and **when you will speak again**. Then speak again at that time even if the news is "still working on it". A missed promised update costs more trust than the outage itself.

Host your status page **somewhere else**. A status page on the infrastructure that is down is a joke you make exactly once. A static page on GitHub Pages or Netlify, or a pinned post on whatever social account you have, costs nothing and works when nothing else does.

3. Stop the bleeding before you understand it

This is the step people skip and it is the most important one.

Roll back. Turn off the feature flag. Disable the expensive endpoint. Take the broken feature offline and leave the rest of the site working. You do not need to know the cause to do any of these — and none of them destroy the evidence, because the logs, the metrics and the bad artifact all still exist.

The most common way a fifteen-minute incident becomes a two-hour one is somebody choosing to understand it first. Curiosity is the right instinct at the wrong moment. End it, then be curious.

4. Ask what changed

Most incidents are a change. Check, in this order:

- Deploys and migrations in the last few hours.
- Config and feature-flag changes — often made by someone who does not think of it as a deploy.
- Your vendors' status pages.
- Anything that expired: a certificate, a card, a key.
- A traffic change: a spike, a crawler, a marketing email that just went out.

If nothing changed on your side, it changed on someone else's. Check the vendor status pages early, because five minutes of reading beats forty of debugging code that is fine.

5. One coordinator, one channel, timestamps

Even alone, keep a running log as you go:

```
14:02 alerted, 5xx on /login ~40%
14:05 posted status update
14:08 rolled back to build 8f2a1c
14:11 error rate falling
14:19 clear. cause not yet known
```

Two minutes of typing during the event, and you have the postmortem timeline for free. You will not remember this accurately at 4am, and reconstructing it afterwards from logs takes an hour.

With more than one person, name a coordinator whose job is deciding and communicating rather than typing commands. Two people debugging in parallel without coordination will undo each other's changes.

6. Change one thing at a time

Under pressure the urge is to try five fixes at once. Resist it. If five changes fix it, you do not know which one worked, so you cannot write it down, cannot prevent it, and are carrying four unexplained changes into next week — one of which may break something else on Thursday.

What not to do

- **Do not restart everything reflexively.** It sometimes helps and it destroys the state you needed to diagnose, which guarantees the same incident next month.
- **Do not scale up to hide a bug.** Sometimes the right emergency move, but write down that you did it, or the leak stays hidden behind larger machines and the bill.
- **Do not silence the alert.** Acknowledge it. Silencing is how the second, worse incident goes unnoticed.
- **Do not blame anyone during the incident.** It is the least useful thing available and it delays the moment someone says "I think I did this", which is often the fastest route to the fix.

Prepare one thing

A runbook, written on a calm afternoon: the rollback command, the vendor status URLs, where the logs are, how to reach the database, who else can help, and the login for the status page. One page.

The value is not the information — you know most of it. The value is that at 3am you are not reconstructing it from memory while people wait.

BEFORE YOU MOVE ON

Twenty minutes after a deploy, error rates jump to 30% and you have no idea why. Your rollback takes about three minutes. What should you do first?

A. Check the logs to identify the failing code path so the rollback is informed

B. Scale up the service, in case the new version is resource-hungry

C. Roll back immediately, confirm recovery, then diagnose from the logs and the failed artifact

D. Post a status update first, then investigate before deciding whether to roll back

46 · 9 min

Writing down what happened

THE ONE THING TO KEEP

A postmortem is worth writing only if it changes something, so measure detection time, name systemic causes rather than people, and leave with fewer than four owned actions.

The purpose is one thing

Not a record. Not accountability. A postmortem exists to make the next incident less likely or less long. Everything in its structure should serve that, and anything that does not is ceremony.

Blameless means something specific

It does not mean pretending nobody ran the command. It means the question is **"what made this easy to do and hard to notice?"** rather than "who did it?"

The reasoning is practical rather than kind. In an organisation that punishes the person who caused an incident, people stop volunteering that they caused one. Detection time goes up, causes go unrecorded, and the same failure recurs with less information each time. Blamelessness is what keeps information flowing.

"Human error" is never a root cause. It is a place to start asking. Why was that command available? Why did nothing check? Why did the interface make the destructive option adjacent to the safe one?

The structure

Summary. Two sentences. What broke, for how long.

Impact, in user terms and in numbers. Not "some users saw errors". Instead: *42 minutes, roughly 3,100 failed requests, 180 signups lost, checkout unavailable in Europe.* Numbers make the priority argument for you afterwards; adjectives do not.

Timeline with timestamps. From your incident log. Include when it started, when it was **detected**, when the first action was taken, when it recovered.

What happened. The mechanism, plainly.

Contributing factors. Plural, always.

What went well. Real, not decorative — the alert fired correctly, the rollback worked, the runbook was accurate. Knowing which defences worked tells you which ones to keep investing in.

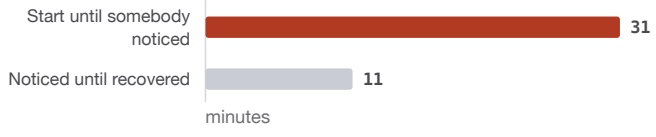
Actions. Owned and dated.

The most useful number is detection time

If the incident lasted 42 minutes and you learned about it at minute 31 from a customer, then the outage was 11 minutes of response and 31 minutes of ignorance. The thing to fix is detection, not the bug.

Teams reliably fix the bug and leave the detection gap in place, which is why the next unrelated bug also runs for half an hour. Split the duration into detect, decide and fix, every time. It tells you where your effort belongs.

A 42-minute outage, split the useful way



The bug was fixed in eleven minutes once anybody knew. The other thirty-one were ignorance, which means the action that shortens the next incident is detection, not the bug.

Five whys is not enough

The technique of asking "why" five times produces a single causal chain, and real incidents are not chains. Three things were true at once: the check did not

exist, the alert threshold was wrong, and the runbook was stale. A chain picks one and calls it the root cause.

Ask instead: **"what else had to be true for this to happen?"** at each step. That produces a set of contributing factors, and it is a much better guide to what to change, because fixing any one of them may have been enough.

Three actions, owned, dated

A postmortem with fourteen unowned action items produces nothing at all, and worse, it teaches everyone that writing them is theatre. Pick the two or three that would most reduce recurrence or detection time. Give each a name and a date. Track them where your other work is tracked, not in the document.

One action that actually lands beats twelve that do not, by an infinite ratio.

Count them over a quarter

Individual postmortems fix individual incidents. Reading a quarter of them together shows the pattern, and the pattern is usually small: most teams find that three causes account for most of their incidents. That is a much better basis for deciding what to build than any individual document.

Even alone, write it

A `docs/incidents/2027-01-14-login-outage.md` costs twenty minutes. Six months later it is the only record of why the odd guard clause exists, why that timeout is 3 seconds and not 30, and why the deploy script refuses to run on a dirty tree. Your future self is a different person with none of your current context, and this is the cheapest way to send them a message.

BEFORE YOU MOVE ON

An outage lasted 42 minutes. It began at 14:02, a customer emailed at 14:33, you rolled back at 14:41 and it recovered at 14:44. The postmortem's single action item is a fix for the bug that caused it. What has been missed?

A. The timeline should include what each person was doing at each step

B. Nothing was learned about detection: 31 of the 42 minutes were before anyone knew, and no alert existed for this failure

C. The impact should be quantified in failed requests and lost signups

D. The postmortem should assign responsibility for the change that introduced the bug

47 · 8 min

Breaking it on purpose, on a Tuesday

THE ONE THING TO KEEP

Every recovery mechanism is a belief until somebody has used it deliberately while nothing was wrong, so rehearse restores, rollbacks, instance kills and dependency failures on a calm Tuesday and record how long each really took.

Untested recovery is a belief, not a capability

You have backups. You have a rollback command. You have a runbook with the steps for a database failover. Every one of those is a claim about what will happen under conditions you have never created.

The backups lesson made this argument for restores. It generalises: every recovery mechanism you own is broken until somebody has used it while nothing was wrong. In practice, roughly half of them are broken the first time, and the reasons are dull — a credential that expired, a script that assumes a directory, a document that names a person who left, a command that needs a permission the on-call person does not have.

The fix is to rehearse, on a Tuesday morning, deliberately, with a rollback plan and a timer.

The five drills worth running, in order of value

1. Restore the database into a scratch environment. Time it end to end. Note the number, because that number is your real recovery time and it is usually two to five times what people guess. Quarterly.

2. Roll back a deploy. Not a hypothetical — actually deploy something harmless and roll it back. Measure the minutes. Monthly is easy because it costs nothing.

3. Kill an instance. Terminate one process or container while traffic flows. Do users notice? Does it come back on its own? This tests the supervisor, the health checks and the load balancer at once.

4. Take a dependency away. Block the mail provider or the model API with a firewall rule for five minutes. Does your site degrade or collapse? This is the drill that exposes missing timeouts, and it is the one that most often finds something.

5. Remove a person. Pick the person who knows the most and forbid them from touching the keyboard while somebody else follows the runbook. This is the cheapest test of documentation ever devised, and it is deeply uncomfortable, which is the point.

How to run one without causing the incident you were preparing for

- **Announce it.** Everybody who might see an alert knows the window. An unannounced drill teaches people to distrust their tools.
- **Write the hypothesis first.** "We believe killing one web instance will cause no visible errors." A drill without a stated prediction is just poking things.
- **Define the abort condition and who calls it**, before you start.

- **Start ridiculously small.** One instance, in staging, in working hours. Production drills come after several successful staging ones.
- **Timebox it.** Thirty minutes, including the write-up.
- **Write down what broke, and fix it that week.** A drill that finds three problems and fixes none is a way of feeling prepared.

Chaos engineering, sized honestly

The industrial version of this — automatically terminating instances in production, injecting latency, running continuously — came from organisations with hundreds of services and full-time reliability teams. The tooling exists and is partly free, but for a project with one or two services it is a distraction.

What scales down perfectly is the *method*: form a hypothesis about steady-state behaviour, introduce one real fault, observe whether the hypothesis held, fix the gap. You need a terminal and a calendar reminder, not a platform.

The game day, once a quarter

A game day is a scripted incident. Somebody writes a scenario — "the primary database is unreachable, it is 09:00 on a Monday" — and everybody else works through it using only the real tools and documents. In its cheapest form, nothing is actually broken: people open the runbook, say what they would type, and discover that the runbook does not say where the credentials live.

For a solo maintainer this works as an hour with a notebook. Read your own runbook as though you had never seen it, follow it literally, and write down every point at which you had to use knowledge that is not written down. That list is your documentation backlog, and it is more accurate than any amount of reflecting on what might be missing.

The number to keep

After each drill, record two figures: how long the recovery took, and how many surprises it produced. Both should fall over a year. If your restore time has

been "about an hour, probably" for three years, you do not have a recovery time — you have a guess that has never been contradicted.

BEFORE YOU MOVE ON

A team runs a drill: they block their mail provider for five minutes. The site keeps working, but every page takes 8 seconds to load. What has the drill demonstrated?

A. The mail provider is a hard dependency and the site should fail closed instead

B. Blocking at the firewall is an unrealistic fault, since real outages return errors rather than hanging

C. A call to the mail provider sits somewhere in the page's critical path with a long or absent timeout

D. The retry policy is too aggressive, adding delay on every request

---

48 · 9 min

Writing it down for whoever is next

> THE ONE THING TO KEEP
>
> Four short documents — how to run it, how to operate it, what expires, and why it is like this — are what turn a running site into something somebody else can keep alive.

Whoever is next is usually you

Not a new hire. You, in eight months, after four other projects, looking at a deploy script you wrote and cannot remember the reason for. All the context that makes the system obvious today is gone by then, and none of it is recoverable from the code.

So this last lesson is about what to leave behind. It is four documents, none of them long, and it is the difference between a project that survives a gap and

one that gets rewritten because nobody dares touch it.

1. README — how to run it

What this is, in two sentences. Then the commands, in order, that take a clean machine to a running local copy. Then the environment variables, with what each one is for and where to get it. Then how to run the tests.

The test for a README is not whether it is complete. It is whether **someone else can follow it on a clean machine without asking you anything.** Every question they have to ask is a bug in the document, and you cannot find those bugs yourself, because the two steps you left out are exactly the two you do without noticing.

2. RUNBOOK — how to operate it

Everything you would need at 3am:

- How to deploy. The exact command.
- How to roll back. The exact command, and how far back you can go.
- How to restore a backup, with the last date this was actually tested.
- Where the logs are, and how to search them.
- Each alert, what it means, and the first thing to check.
- Vendor status pages, support contacts, account details.

Write it after an incident, when you remember what you wished you had. Then it stays accurate because you keep needing it.

3. RENEWALS — what expires and on whose card

From the earlier lesson. What you pay for, where, which card, what expires when, and which role address owns the account.

4. DECISIONS — why it is like this

The shortest document and the one that saves the most damage. A handful of entries, three lines each:

> ***Soft delete on threads, not hard delete.*** *A legal-hold record can reference the content, and a hard delete orphans replies. See docs/09.*

> ***Client components must not import database modules.*** *It pulls the Postgres driver into the browser bundle and 500s every page.* `tsc` *and* `eslint` *both pass while this is broken;* `verify:client-boundary` *is the check.*

Without these, a reasonable person doing a reasonable cleanup deletes the guard, and everything breaks in a way that takes a day to trace. A decision with no written reason has no defence. This is the same practice as the post-mortem, applied before the incident instead of after.

Access, which is not a document

The accounts must not live in one person's personal email or personal password vault.

- Role addresses — `ops@`, `admin@` — on every vendor account, forwarding to whoever holds the role now.
- A shared password manager. **Bitwarden** has a free tier and **Vaultwarden** is a free self-hosted server that speaks the same protocol; 1Password is the paid standard.
- MFA recovery codes stored where a second person can reach them, because the more likely disaster is being locked out, not being broken into.
- Registrar and DNS access documented explicitly. They are the two that cannot be recovered by any other route.

The honest test

Ask somebody to deploy a trivial change using only the documents. Watch, and do not help. Every time they stop, you have found a gap.

If there is nobody to ask, do it yourself on a machine you have never used, with your notes closed. It is uncomfortable and it is the only reliable check available.

Where this course ends

You started with a thing that worked on your laptop. You now have a picture of how a request reaches it, a deploy that is boring and reversible, enough instrumentation to answer "is it broken and since when", a sense of where it will fall over and what that costs, and a plan for the day it breaks.

None of it is glamorous. That is the point of the subject. Shipping is not the last step of building — it is a separate practice, made out of small boring artifacts that each prevent one specific bad evening.

Go and deploy the thing. Then roll it back once, on purpose, while nothing is wrong.

BEFORE YOU MOVE ON

You write a thorough README and hand the project to a colleague, who gets stuck twice and asks you for help both times. What does that tell you?

A. They lacked the background knowledge the README reasonably assumes
B. The README is fine, and their questions belong in a separate troubleshooting guide
C. The project is unusually complex, and some handover friction is unavoidable
D. The two questions are two bugs in the README, and they are ones you could not have found yourself because you do those steps without noticing

CASE STUDY · MODULE 5

Roll back, and lose twenty-five minutes of bookings

A teleconsultation booking service used by four rural clinics in Vidarbha, at five past nine on a Monday morning, twenty-five minutes after a deploy.

The service books video consultations between village clinics and doctors in Nagpur. Monday between nine and eleven is the busiest window of the week by a distance, because that is when the clinics open and the queue outside is already formed.

The deploy went out at 08:40. It contained a new table, `booking_slots`, and code that writes a row to it for every booking as well as to the old `bookings` table, plus an SMS confirmation to the patient's phone. It had been tested for a fortnight on staging.

At 09:05 the alert fired: more than a third of requests to the booking route returning 5xx, sustained for four minutes. The maintainer, who is one person with a laptop in Nagpur, opened the logs.

What he found first was not a cause. It was an arithmetic problem. Between 08:40 and 09:05, about sixty bookings had succeeded. Each of those had written a row into `booking_slots` — a table the previous build knows nothing about — and each patient had received a confirmation message telling them to come at a stated time. Rolling back would make those sixty bookings invisible to the application. Sixty people would arrive at four clinics, holding a message from a system with no record of them.

The error itself gave no hint. The stack trace ended in the HTTP client. Latency on the booking route had gone from two hundred milliseconds to thirty seconds. Requests to routes that had nothing to do with booking were also failing, which is the shape of a resource problem rather than a bug in one handler — something is holding workers, and everything queues behind it.

The scope told him a little more, and took a minute to establish. It was all four clinics, not one, so it was not somebody's connection. It was every route rather than one, which pointed away from the code he had just changed and towards

something shared. And it had begun before the alert: the error rate graph, read backwards, rises out of the noise at about 08:52, thirteen minutes before anything told him.

What he did not have was a dependency screen. The service calls three external things — the video provider, the SMS provider and the mail relay — and none of them had a rate or a latency graph. "Is it us or them" is the second question in most incidents, and at 09:06 he had no way to answer it in under ten minutes.

So at 09:07 there were two paths.

Roll back now. The fire stops. The evidence — logs, metrics, the bad artifact — all survives a rollback, so diagnosis loses nothing. The price is sixty invisible bookings, which is not an abstraction: it is somebody telephoning four receptionists and getting names written into a paper book before the patients arrive, and it is a repair job to write afterwards.

Spend ten more minutes understanding it. If it is something small and obvious, the fix is forward and nothing is lost. If it is not, thirty-eight per cent of the busiest two hours of the week continue to fail while he reads.

There was a third question, smaller and easy to skip: whether to say anything publicly before knowing what was wrong. The service has a status page. The instinct is to wait until there is something definite to say.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

He posted at 09:07, before doing anything else: one line saying that bookings were failing, that somebody was working on it, and that the next update would be at 09:30. The status page is a static page on a different host from the appli-

cation, which is the only reason it was available. He then rolled back at 09:10. The error rate was falling by 09:12 and clear by 09:14.

The sixty rows were exported from `booking_slots` to a CSV at 09:20 and read down the phone to four receptionists, who wrote them into the paper book. Two people, forty minutes. Every patient was seen.

The cause was found at 10:20, from the artifact and the logs, with the site up the whole time. The new code called the SMS provider inline, in the request, with no timeout — and Node's fetch has no default timeout. The provider was slow that morning, not down. Each booking held a worker for thirty seconds. Within a minute every worker in every instance was waiting on the same silent socket, which is why pages that never touch SMS were failing too.

The postmortem recorded the numbers rather than adjectives: forty-two minutes, about 3,100 failed requests, roughly two hundred bookings not made. It also recorded that the failure began at 08:52 and was detected at 09:05 — thirteen minutes of ignorance against twenty-nine minutes of response. Three actions, each with a name and a date: an explicit connect and read timeout on every outbound call, smaller than the caller's own budget; SMS moved into the job queue with a circuit breaker in front of the provider; and a rule that a deploy adding a table ships the day before the code that writes to it. A monthly rollback drill was added, and timed at one hundred seconds.

Rolling back cost sixty bookings and forty minutes of telephone calls. Why was it still the right call at 09:07?

Because the cost of rolling back was known and bounded, and the cost of continuing was neither. Sixty rows can be exported and repaired; an outage of unknown length in the busiest window of the week cannot be estimated. Rolling back also destroys no evidence — the logs, the metrics and the bad artifact all survive, and the cause was in fact found an hour later with the site up. The common way a fifteen-minute incident becomes a two-hour one is somebody deciding to understand it first.

What made this particular rollback expensive, and what would have made it cheap?

Shipping the new table and the code that writes to it in the same release. That put data in a shape the previous build cannot read, so undoing the code stranded twen-

ty-five minutes of real bookings. The expand, migrate and contract sequence would have made it cheap: ship the table on its own, then the code that writes to both, then the code that reads the new one. At every point the previous build still works, so the code rollback stays free while only the schema moves forward.

The postmortem records a detection gap of thirteen minutes. Why is that number more useful than the cause?

Because the cause was specific to one missing timeout and will not recur, while thirteen minutes of blindness applies to every future failure whatever its cause. Teams reliably fix the bug and leave the detection gap in place, which is why the next unrelated incident also runs for a quarter of an hour before anybody knows. Splitting the duration into detect, decide and fix tells you where the effort actually belongs.

MODULE 5 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 Somebody argues that the read replica makes nightly dumps unnecessary. What does the replica not protect against?

A. A disk failing in the primary's storage array
B. A region becoming unreachable for several hours
C. A `DELETE FROM users` run by mistake
D. A network partition between availability zones

2 Your only backup is a dump that runs at 02:00. Data is destroyed at 15:30. What have you chosen?

A. An RTO of thirteen and a half hours, and an RPO of one day
B. An RPO of thirteen and a half hours, whatever the restore then takes
C. An RPO of twenty-four hours regardless of when the loss occurs
D. Neither, because point-in-time recovery covers the gap automatically

3 Why register every vendor account to a role address such as ops@ rather than to a person's mailbox?

A. Shared mailboxes are exempt from most providers' two-factor requirements
B. It makes the account transferable without contacting the provider's support
C. Deprecation and expiry notices then reach whoever currently holds the role
D. Billing statements can be filed automatically without a person reading them

4 A circuit breaker has closed and open states. What does the half-open state add?

A. A period in which failures are counted but calls are still allowed
B. A single trial call that decides whether to close again or stay open
C. A queue that holds requests until the dependency recovers
D. A slower rate of calls while the dependency is still degraded

5 Ten minutes into an outage the cause is still unclear. Why roll back before you understand it?

A. Because a rollback fixes the underlying fault in nearly every case
B. Because logs are cleared when the process restarts, so speed matters
C. Because errors make the logs too noisy to read while they are being produced
D. Because the evidence survives a rollback and your users' patience does not

6 An incident lasted forty-two minutes; you learned about it at minute thirty-one, from a customer. What should the postmortem's main action address?

A. Detection, because eleven minutes of response followed thirty-one of ignorance
B. The bug itself, because fixing it prevents this incident recurring at all
C. The rollback procedure, since it was reached later than it should have been
D. Communication, because the customer heard about it before your team did

MODULE 6

Shipping a model, not just a web app

A prediction endpoint is a web service with four extra problems: the weights are large, the latency is dominated by arithmetic rather than waiting, the cost is per token or per GPU-second, and the thing can get worse without anybody deploying anything. This module is the operations of machine learning — serving, cost, versioning, skew, drift and the fallback for when the model is not there.

> **BY THE END OF THIS MODULE YOU CAN**
>
> Put a model behind an HTTP endpoint with a measured latency budget and a stated cost per request, ship a new version by shadow then canary rather than by hope, and detect the week its inputs stop resembling the data it was trained on

49 · 9 min

A prediction endpoint is a web service with a large secret

THE ONE THING TO KEEP

A model endpoint is an ordinary web service whose memory, start-up time, concurrency and per-request cost are one to three orders of magnitude larger, so load weights once, size workers against cores rather than against waiting, and push anything slow into a queue.

The same shape, with different constants

A model behind an HTTP endpoint is still a process holding a port open. Everything from the first module applies: a proxy in front, a health check, a supervisor, logs with request IDs. What changes is the size of four constants, and every operational surprise in this module comes from one of them.

- **Memory.** A web process is 100–300 MB. A model process is the size of its weights plus the runtime. A gradient-boosted tree is a few megabytes. A small transformer at 8-bit is around 8 GB. An embedding model is a few hundred megabytes.
- **Start-up time.** Loading weights from disk takes seconds to minutes. A web app starts in one second.
- **Concurrency.** A web worker spends most of its life waiting on the database, so twenty threads on four cores is fine. A model spends its life computing, so twenty concurrent requests on four cores makes all twenty slow.
- **Cost per request.** Measured in fractions of a cent for a database query, and in cents for a large model call.

Load once, at start-up, and never inside the handler

The first version of every prediction endpoint contains this bug:

```
@app.post("/predict")
def predict(req: Item):
    model = joblib.load("model.pkl")   # 400 ms, every single
request
    return {"score": float(model.predict_proba([req.features])
[0][1])}
```

Load at import or in a start-up hook, keep it in module scope, and let every request share it. The corollary is that your process now has a warm-up: readiness must stay false until the weights are loaded, or the load balancer will send traffic to a process that is still reading a file.

The same fact explains why scaling to zero hurts more here. A web app's cold start is a second; a model's is however long the weights take to load, and the unlucky user waits for all of it.

Workers, threads and the number that is not what you expect

For CPU inference, the arithmetic library is already using every core. Running four worker processes that each start four maths threads gives sixteen threads fighting over four cores, and the result is slower than one worker. This is the most common self-inflicted performance problem in model serving.

```
OMP_NUM_THREADS=1 gunicorn -w 4 app:app       # 4 workers, 1
thread each
# or
OMP_NUM_THREADS=4 gunicorn -w 1 app:app       # 1 worker, 4
threads
```

Measure both. Which wins depends on whether your model parallelises well internally, and it is a ten-minute experiment with a load generator.

On a GPU the rule is stricter: one process owns the device, because two processes each allocate their own copy of the weights and then fight for memory. Concurrency comes from batching inside that process, not from more processes.

Do not put a slow model inside a fast request

If inference takes more than a few hundred milliseconds, treat it as background work and apply the queue lesson: accept the request, return a job ID, do the work in a worker, notify when done. Document upload, transcription, batch scoring and image generation all belong here.

What stays inline is anything the user is watching: a search ranking, a classification that gates a form submission, a chat response you stream token by token. For those, streaming is the honest trick — a first token in 300 ms with the rest arriving over four seconds feels dramatically better than four seconds of nothing, even though total time is identical.

The free paths, and they are unusually good here

You do not need a GPU to learn any of this.

- **Classical models:** scikit-learn, XGBoost or LightGBM exported to ONNX and served with ONNX Runtime. A tree model scores in under a millisecond on a laptop CPU.
- **Embeddings:** `sentence-transformers` models are small enough to run on a free-tier box and are the workhorse of search and recommendation.
- **Language models locally:** llama.cpp and Ollama run quantised models on ordinary hardware. A 7–8 billion parameter model at 4-bit needs roughly 5 GB of RAM and produces a handful of tokens per second on a modern CPU — slow, free, and enough to learn every lesson in this module.
- **Serving frameworks:** FastAPI for a plain endpoint, vLLM or TGI when you have a GPU and need throughput, BentoML or Ray Serve when you need packaging.

Hosted APIs from any provider are the other legitimate answer, and for most small products the right one — the next two lessons are about deciding which, with numbers.

The health check that means something

`GET /health` returning 200 from a process whose weights failed to load is a lie of exactly the kind the health-check lesson warned about. Make readiness assert that the model object exists and can score one fixed example. Keep that example in the repository and check the output has not changed: it is a smoke test for a corrupted download, a silently swapped file, or a library upgrade that changed a default.

BEFORE YOU MOVE ON

A CPU-served model runs at 40 ms per request with one worker. The team moves to four workers on a four-core machine and latency rises to 130 ms under the same load. What is the most likely explanation?

A. Each worker starts its own arithmetic threads, so sixteen threads contend for four cores

B. The four workers each hold a copy of the weights, exhausting memory and causing swapping

C. The load balancer is now spreading requests, so each worker's cache of recent inputs is less effective

D. Four workers exceed the model's supported batch size, forcing requests to queue

50 · 10 min

Where inference time actually goes

THE ONE THING TO KEEP

Language-model latency splits into a parallel prefill that sets time-to-first-token and a memory-bandwidth-bound decode that sets tokens per second, which is why smaller and quantised models are faster, why batching raises throughput almost for free, and why output length is the lever that matters most.

Two very different clocks

For a small model — a tree, a logistic regression, an embedding — inference is arithmetic on a fixed-size input, and it takes the same time every request. Measure it once and you know it.

For a language model, generation is a loop, and the cost is in two distinct phases that behave differently:

- **Prefill.** The whole prompt is processed at once. This phase is compute-bound and parallel, so a 2,000-token prompt is not twenty times slower than a 100-token one, but it does set your time to first token.
- **Decode.** Tokens come out one at a time, each requiring a full pass over the weights. This phase is memory-bandwidth-bound: the limit is how fast the weights can be read from memory, not how fast the chip can multiply.

That second fact explains almost every surprising benchmark result. If decoding is limited by memory bandwidth, then a smaller model is faster for a simple mechanical reason — there is less to read per token — and quantising a model to 4 bits roughly halves the bytes moved, which roughly doubles the token rate. It also explains why decoding one sequence leaves an expensive GPU mostly idle.

Two clocks inside one language-model request

Prefill: the whole prompt at once

- Compute-bound and parallel
- A 2,000-token prompt is not twenty times a 100-token one
- Sets time to first token
- Shortening the prompt helps here

Decode: one token at a time

- Memory-bandwidth-bound: a full pass over the weights per token
- Fewer bytes to read means faster: 4-bit roughly doubles the rate
- Sets tokens per second
- Output length is the lever that matters most

Prefill sets the time to first token. Decode sets tokens per second, and because it is limited by reading the weights rather than by arithmetic, a smaller or quantised model is faster for a purely mechanical reason.

Batching, which is why hosted APIs are cheap

Because decode is bandwidth-bound, processing eight sequences at once costs barely more than processing one: the weights are read once and used for all eight. Throughput multiplies; per-request latency barely moves.

This is what a serving stack like vLLM or TGI does with continuous batching — new requests join the running batch between token steps rather than waiting for a batch to fill. It is also why a provider serving thousands of users per GPU can charge less per token than your idle GPU costs you.

The trade-off is real and worth stating: batching raises throughput and raises the latency of the individual request slightly. Under a strict latency budget with sparse traffic, small batches win.

The KV cache, and why long conversations get expensive

During decode, the model keeps the intermediate attention state for every token so far — the key-value cache — so it does not recompute the whole conversation for each new token. It is the reason generation is not quadratic in practice.

It also lives in memory, and it grows with context length times batch size. On a GPU it competes with the weights for the same memory, which is why a server that comfortably handles ten short conversations falls over on three long ones. If you are self-hosting and see out-of-memory errors that correlate with long chats rather than with request volume, this is it.

Measure your own, in one command

Do not trust a benchmark from a machine you do not have. For a hosted API:

```
curl -s -w '\ntotal=%{time_total}s ttfb=%
{time_starttransfer}s\n' \
  -o /dev/null -X POST https://api.example.com/v1/chat \
  -H "Authorization: Bearer $KEY" -H 'content-type:
application/json' \
  -d '{"model":"...","stream":true,"messages":
[{"role":"user","content":"..."}]}'
```

`time_starttransfer` with streaming on is your time to first token; the gap to `time_total` divided by the number of output tokens is your per-token rate. Record both for the p50 and the p95 — model latency has a much longer tail than database latency, and the average will flatter you.

For a local model, `llama-bench` reports prefill and decode rates separately, which is exactly the split above.

The four levers, in order of effect

1. **Generate fewer tokens.** Output length is the dominant term in both latency and cost. Ask for a shorter answer, set a max-tokens limit, and prefer structured output over prose.
2. **Use a smaller model.** Often the largest single win, and often free of quality cost for classification and extraction. Test it rather than assuming.
3. **Quantise, if self-hosting.** 8-bit is usually indistinguishable in quality; 4-bit is noticeably cheaper and sometimes noticeably worse. Measure on your own examples, not on a leaderboard.
4. **Shorten the prompt.** Helps prefill and cost, but has less effect on total latency than people expect, because decode dominates for long outputs.

And one that is not on the list: retrieving more context to improve quality makes prefill longer and the bill larger. That is a legitimate trade, but make it on purpose.

Stream, and the perceived time halves

A four-second response delivered as a stream, with the first token in 400 ms, is experienced as fast. The same response delivered whole is experienced as broken. Server-sent events are the standard mechanism and are supported by every provider; the cost is that your proxy must not buffer, which is one configuration line and a common cause of "streaming does not work in production but works locally".

BEFORE YOU MOVE ON

Self-hosting a model, you cut the prompt from 1,500 tokens to 400 but keep the same 800-token answers. Total latency barely improves. Why?

A. The KV cache still holds the original context, so memory traffic is unchanged

B. Prompt tokens are billed but not processed, so shortening them changes cost rather than time

C. Most of the time is decode, which depends on the number of output tokens, not on prompt length

D. Quantisation already removed the prefill cost, so prompt length no longer matters

51 · 10 min

What a prediction costs, worked out

THE ONE THING TO KEEP

Compute cost per request from token prices or from GPU-hours divided by realistic utilisation, and the comparison settles itself: hosted APIs win while you are idle, your own hardware wins only at sustained volume, and caching plus a per-user quota is the lever you control either way.

Do the arithmetic before you choose

Every argument about hosted APIs versus self-hosting is settled by two numbers you can compute in ten minutes: cost per request, and requests per month. People skip this and choose by preference, then discover the answer on an invoice.

Prices change constantly, so learn the *method* and re-run it with today's numbers. The shape of the answer has been stable for years even as the constants fall.

Cost per request, for a hosted API

Providers bill per million tokens, input and output priced separately, with output typically three to five times the price of input. Take a worked example with round numbers: input at $0.30 per million tokens, output at $1.20 per million.

A request with a 1,200-token prompt and a 400-token answer costs:

- Input: 1,200 ÷ 1,000,000 × $0.30 = $0.00036
- Output: 400 ÷ 1,000,000 × $1.20 = $0.00048
- Total: about **$0.0008**, or 0.08 cents.

At 100,000 requests a month that is $80. At 10 million it is $8,000. The interesting part is what moves it: doubling the answer length adds more than dou-

bling the prompt length, and a retrieval step that adds 3,000 tokens of context to every request roughly triples the input cost. Both are decisions you make in code, not properties of the model.

Two discounts are worth knowing because they are large and often unused. **Prompt caching** reprices the repeated prefix of a request — a long system prompt or a fixed document — at a fraction of the normal input rate, often around a tenth, when the same prefix is sent again within a short window. **Batch endpoints** halve the price for work that can wait hours. Anything nightly should be on the batch tier.

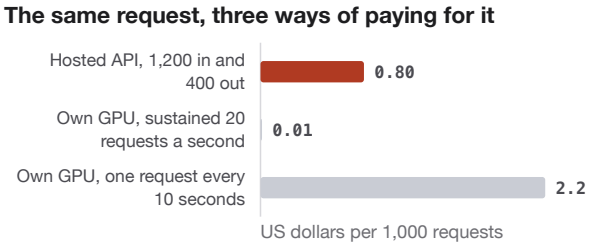
Cost per request, for your own GPU

Rent, do not buy, until you have a year of data. A mid-range inference GPU rents for roughly $0.40 to $1.50 an hour depending on the card and the provider, and the smaller specialist clouds are consistently cheaper than the large ones for this.

The key figure is **utilisation**, and it is the one people get wrong. A GPU costs the same when idle. At $0.80 an hour:

- $576 a month, continuously.
- If it serves 20 requests per second, sustained, that is 52 million requests a month and about $0.00001 each — a hundred times cheaper than the API.
- If it serves one request every ten seconds — a realistic figure for a small product — that is 260,000 requests a month and about $0.0022 each, nearly three times *more* expensive than the API.

That is the whole comparison. Self-hosting wins on volume and loses on idleness, and small products are mostly idle. Add to the self-hosted side the engineering time to run it, which is the cost nobody puts in the spreadsheet and often the largest one.



A rented GPU at 80 cents an hour costs 576 dollars a month whether it works or idles. Busy, it undercuts the API a hundredfold. At one request every ten seconds, which is a realistic small product, it costs nearly three times more.

The honest cases for self-hosting are: sustained high volume, data that must not leave your infrastructure, a fine-tuned model that no provider serves, and unmetered experimentation on a machine you already own.

Caching, which is the biggest lever you control

The cheapest inference is the one you do not run.

- **Exact-match cache.** Hash the normalised input and store the output. On any product with repeated queries this removes a surprising share — often 10 to 30% — for the cost of a Redis lookup.
- **Embedding cache.** Embeddings are deterministic for a given model and text. Never compute one twice; store it with the row.
- **Semantic cache.** Match near-duplicate questions by embedding similarity. Effective and genuinely risky: a threshold that is too loose returns the answer to a similar-but-different question. Log every hit and check them for a week before trusting it.

Put a ceiling on it, per user and per day

The cost lesson earlier in this course made the general point; models make it urgent, because a single user with a script can spend more in an hour than your monthly hosting. Three controls, all of which belong in code before launch:

1. A **per-user daily quota**, counted in a table, claimed before the call and refunded if the provider fails.

2. A **hard max-tokens** on every request. Without it, one pathological input can generate the model's maximum output.
3. A **monthly spend cap** at the provider, plus an alert at half of it. Caps stop money leaving; alerts only tell you it left.

Record the actual token counts from each response into your own table alongside the user and the feature. Providers report a total; only you can say which feature spent it, and that breakdown is what lets you cut cost without cutting the product.

The free path, for learning

Ollama or llama.cpp on your own laptop costs nothing and teaches every concept here, at a token rate slow enough to make the arithmetic obvious. Free API tiers exist at several providers and are enough for a course project. Nothing in this module requires you to spend money to understand it.

BEFORE YOU MOVE ON

A team rents a GPU at $0.80/hour to replace an API costing $0.0008 per request. They serve about 2 requests per minute. What does the arithmetic say?

A. Self-hosting is cheaper, because the fixed cost is spread over unlimited requests
B. Roughly break-even, since GPU pricing is designed to match API pricing at typical load
C. Self-hosting costs about $576 a month for around 86,000 requests, roughly $0.0067 each — about eight times the API price
D. Cost is unchanged and the decision should be made on latency alone

52 · 9 min

Which model, which prompt, which version

THE ONE THING TO KEEP

A model system has four independently moving versions — model snapshot, prompt, decoding parameters and retrieval index — so pin each one and record all four with every response, or no complaint about last Tuesday can be investigated.

The complaint you cannot investigate

A user writes: last Tuesday your assistant told me something wrong, here is a screenshot. You open the code. The prompt has been edited twice since. The provider has silently moved their model pointer to a newer snapshot. Somebody changed the temperature. The retrieval index was rebuilt on Wednesday.

You cannot reproduce the answer, so you cannot say whether it was fixed, and you cannot tell whether the change you are about to make would have prevented it. The deploy lesson gave you a commit SHA for code. A model system needs the same discipline over four more things that all change independently.

The four versions to record with every response

1. **Model identity, pinned.** Not `gpt-latest` or `claude-latest` or whatever the alias is called this year, but the dated snapshot. Aliases move under you, and when they move, quality changes without a deploy. Pin the version, and treat a provider's deprecation notice as a scheduled piece of work rather than a surprise.
2. **Prompt version.** The prompt is code. It belongs in the repository, in a file, with a hash — not in a database row somebody edits in an admin panel, and certainly not pasted in three places.

3. **Decoding parameters.** Temperature, top-p, max tokens, stop sequences, tools available. A temperature of 0.9 makes reproduction impossible in principle; recording the seed where a provider supports one makes it possible again.
4. **Retrieval state.** If context is fetched, record the index version and the identifiers of the documents that were actually retrieved. "The model hallucinated" and "the retriever returned the wrong document" look identical from outside and require completely different fixes.

Store all of it on the row with the response, or in a log line with the request ID:

```
{"request_id":"c9f2a1","model":"provider/model-2026-04-
12","prompt":"answer_v7:9f31",
 "temp":0.2,"max_tokens":600,"index":"kb-2026-08-30","docs":
["kb/412","kb/77"],
 "in_tokens":1204,"out_tokens":388,"latency_ms":2140}
```

That single line answers the complaint, feeds the cost breakdown from the previous lesson, and tells you which release a regression started in.

A registry, for models you train yourself

If you produce your own models, the artifact is a file — and a file with no provenance is a liability. What to record, at minimum: the training data snapshot or its hash, the code commit, the hyperparameters, the evaluation numbers on the held-out set, who trained it and when.

MLflow is the free, self-hostable standard for this, and its model registry gives you a version number, a stage, and the run that produced it. DVC handles the data-and-artifact half in git. If both feel heavy, a directory named by date and a `model.json` beside every artifact carrying those six fields is a legitimate starting point — the point is that the fields exist, not that a platform holds them.

One rule: the artifact you evaluated must be the artifact you deploy. Re-training "the same model" for production, with the same script and different ran-

domness, produces a different model with unverified numbers. This is the build-once-run-anywhere argument, transferred.

Evaluate before you ship, on a set that never moves

This course is not the evaluation course, and the details of building a held-out set belong there. The shipping-side rule is short: a model or prompt change is a deploy, so it needs a check that runs automatically and can block it.

In practice that is a file of 50 to 300 examples with expected properties, run in CI, reporting a number. Not a perfect measure — nobody has one — but a tripwire. Teams without it discover regressions from users; teams with it discover them in the pull request. The gap between those two is usually one afternoon of work.

Record the number against the prompt hash so you have a history. A quality figure that only exists in somebody's memory of last month is not a baseline.

Prompts in the database, honestly

There is a real argument for editable prompts: non-engineers can improve them without a deploy. The cost is that your production behaviour is no longer in version control, and the reproduction problem above returns in full.

The compromise that works: prompts live in the repository as the source of truth, an interface lets people propose edits, and an edit becomes a pull request. If you must allow live editing, version every change, record the version with every response, and keep an audit trail of who changed what. Never allow a live edit that leaves no trace.

The smallest version of this lesson

One table, one row per model call, six columns: request ID, model, prompt version, parameters, retrieved document IDs, token counts. It costs an hour. Without it, every question anybody asks about your model's behaviour is answered with a shrug.

BEFORE YOU MOVE ON

Quality drops noticeably one week with no deploy, no prompt change and no code change. Which recorded field would most directly identify the cause?

A. Token counts per request, showing whether prompts grew

B. Latency percentiles, showing whether the provider slowed down

C. The pinned model snapshot identifier stored with each response

D. The retrieval index version, showing whether documents changed

53 · 10 min

The feature computed two different ways

THE ONE THING TO KEEP

Most deployed models fail because the features at serving time are computed by different code, from different data, at a different moment than the features they were trained on — so share one implementation and diff the two paths on real rows before anybody trusts the score.

The bug that only exists in production

A model scores 0.91 on the held-out set. It ships. Live performance is closer to random, and nothing in the code is wrong in any way a test would find.

The usual cause is that the numbers going into the model at serving time are not the numbers it was trained on. The training pipeline computed `days_since_signup` from a batch table in pandas; the serving code computes it in the web application, in a different timezone, with a different definition of "today". The model sees a feature it has never seen and returns confident nonsense.

This is training-serving skew, and it is the most common serious failure in deployed machine learning. It is an operations problem, not a modelling one,

which is why it belongs in this course.

The four ways the two paths diverge

1. Two implementations. Training in SQL or pandas, serving in application code. Every difference in rounding, null handling, string normalisation and default value is a silent bug. This is the big one.

2. Different data. Training uses a warehouse table that has been cleaned, deduplicated and back-filled overnight. Serving uses the live row, which has nulls the warehouse fixed.

3. Time travel, in both directions. Training joins a value as it is *today* rather than as it was at the moment being predicted — the label leaks backwards, the offline score is beautiful, and no such information exists at serving time. Or the reverse: serving reads a value that has not been updated yet, because the job that computes it runs at 03:00.

4. Categories that did not exist. A new country code, a new plan name, a new device type. The encoder was fitted on last quarter's values and maps the unknown one to whatever its default is — often index 0, which means "the most common category", which is a lie.

The one habit that prevents most of it

Compute each feature in exactly one piece of code, and call that code from both paths.

```
# features.py — imported by the training job AND the serving
app
def build(row, now):
    return {
        "days_since_signup": (now.date() -
row.signup_at.date()).days,
        "posts_7d": row.posts_7d or 0,
        "country": normalise_country(row.country),
    }
```

This is unglamorous and it removes an entire class of failure. If the two paths genuinely cannot share code — different languages, different runtimes — then the tests below stop being optional.

The test that catches it in ten minutes

Take a hundred rows. Score them through the training pipeline and through the live serving path. Compare feature by feature, not just the final prediction.

```
# expect an empty diff; anything printed here is skew
diff <(jq -S . offline_features.json) <(jq -S .
online_features.json)
```

Mismatches will be boring: a timezone, a null becoming zero in one path and the mean in the other, a string trimmed on one side. Boring and fatal. Run this as a scheduled job, not once — skew arrives when somebody edits one path.

Then log the served feature vector for a sample of live requests. Compare its distribution against the training distribution weekly; that comparison is also your drift monitor two lessons from now, so the work is shared.

Feature stores, and whether you need one

A feature store is a system that computes features once and serves them to both training and inference, with a record of what each value was at each point in time. Feast is the free, open-source one; every cloud has a managed version.

Be honest about the threshold. A feature store earns its complexity when several models share features, when features are computed by scheduled jobs rather than from the request, or when point-in-time correctness across many tables is hard to get right by hand. For one model with ten features derived from the current row, a shared Python module and the diff test above give you most of the benefit for none of the operational cost.

Save the fitted transforms with the model

The scaler, the encoder, the vocabulary and the imputation values are part of the model. Serialise them together — a scikit-learn `Pipeline`, an ONNX graph that includes preprocessing, or one directory containing all of them — and load them together.

The failure otherwise is subtle and common: the model file is deployed, a slightly older encoder is already on the box, the column order shifts by one, and every prediction is wrong in a way that no error message reports. A checksum of the preprocessing artifact, logged at start-up next to the model version, turns that into something you can see.

BEFORE YOU MOVE ON

An offline model scores 0.91 AUC. In production its ranking is barely better than random, yet the serving code runs without error and the model file is correct. What should you check first?

A. Whether the served feature values match those produced by the training pipeline for the same rows

B. Whether the model is overfitted, by re-splitting the training data

C. Whether inference latency is causing timeouts that drop predictions

D. Whether the traffic distribution has drifted since training

54 · 9 min

Shadow, then canary, then everybody

THE ONE THING TO KEEP

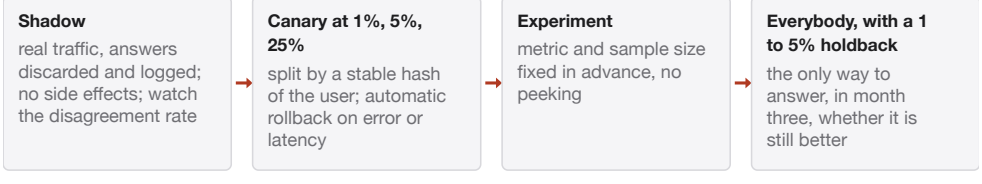
Ship a model in three stages — shadow to find what breaks, canary on a user-hashed slice with automatic rollback, then a properly sized experiment — and keep a small holdback afterwards, because the artifact rolls back but the data it wrote does not.

An offline improvement is a hypothesis

The new model beats the old one on the held-out set. That is evidence about a fixed dataset, collected in the past, scored by a metric you chose. It is not evidence about live traffic, which contains inputs your set does not, arrives at a different rate, and is followed by user behaviour your metric never measured.

The gap between offline and online results is one of the most consistently reported findings in applied machine learning: a substantial share of changes that improve an offline metric fail to improve the online one, and some make it worse. Nothing about that is a criticism of offline evaluation. It is a reason to ship models the way you ship code — gradually, with a way back.

Shadow, then canary, then everybody



Each stage answers a different question. Shadow finds what breaks, canary finds what the users do, the experiment settles whether it is better, and the holdback keeps that comparison alive after the launch.

Stage one: shadow

Send real traffic to the new model, discard its answers, and record them beside the old model's.

The user sees only the current model, so the risk is close to zero. What you learn is not quality but the things that break first: latency under real input sizes, memory under real concurrency, unexpected inputs, error rates, and how often the two models actually disagree. Disagreement rate is the most useful number here — if the new model differs on 0.4% of requests, the online effect will be small whatever the offline table said; if it differs on 30%, be careful.

Shadowing costs double inference for its duration, which is a real bill, so run it for hours or a day rather than a fortnight, and shadow a sample of traffic rather than all of it.

One rule: shadow requests must not have side effects. No emails, no writes, no charges. Route them through a flag that disables everything downstream, and test that the flag works before you turn shadowing on.

Stage two: canary

Give the new model a small share of real traffic — 1%, then 5%, then 25% — and compare the same metrics for both groups.

Watch three families of number: technical (latency, errors, cost per request), model (score distribution, confidence, refusal or fallback rate), and product (the thing you actually care about — clicks, completions, reports, purchases). Automate the rollback: if error rate or latency exceeds a threshold for five minutes, revert without waiting for a human.

Split by a stable hash of the user ID, not per request. A user who gets the new model on one page and the old one on the next has an incoherent experience and pollutes your comparison.

Stage three: the honest experiment

If the decision matters, run it as a proper A/B test with the metric and duration fixed in advance. Two warnings specific to models:

- **Peeking.** Checking the result daily and stopping when it looks good manufactures significance out of noise. Fix the sample size before you start, or

use a method designed for continuous monitoring.

- **Novelty and learning effects.** A changed recommendation feed produces a short-term bump because it is different, then settles. A week of data on a two-week effect is a wrong answer delivered confidently.

And for many small products the sober truth is that traffic is too low to detect the difference you are arguing about. When that is the case, say so, choose on other grounds — cost, latency, simplicity — and stop pretending the experiment settled it.

Keep a holdback

When the new model wins and goes to everybody, keep a small slice — 1 to 5% — on the old one for a while. It costs almost nothing and it is the only way to answer the question that always arrives in month three: is this system actually better than what we had, or has everything just drifted?

Holdbacks are also how you notice slow degradation that affects both groups equally, because the comparison stays live.

Rollback is different for models

Code rolls back cleanly. A model has three complications:

- **The old artifact must still exist**, with its preprocessing, at a version you can deploy. Keep the last two or three, and their evaluation numbers.
- **Data it produced does not roll back.** Predictions written to rows, embeddings stored in an index, moderation decisions applied to content — those persist. If the model wrote to a shared index, reverting the code does not revert the index.
- **The feedback it created stays in your training data.** Which is the subject of the feedback-loop lesson later in this module.

So the deployment note for a model is the same as for a schema change: separate the reversible part from the irreversible one, and know which is which before you press the button.

BEFORE YOU MOVE ON

A team shadows a new ranking model for a day. Latency, errors and cost all look fine, and it disagrees with the current model on 0.3% of requests. What is the reasonable conclusion?

A. It is operationally safe, but a 0.3% disagreement rate means any online effect will be small and may be undetectable at this traffic
B. The new model is safe to deploy to everybody, since shadowing showed no problems
C. The shadow test is invalid because the new model's answers were discarded
D. A low disagreement rate proves the offline improvement was measurement error

55 · 10 min

The model that quietly stopped fitting

THE ONE THING TO KEEP

A model degrades without any deploy, so watch what you can see immediately — input distributions, missingness and the prediction distribution — collect a small stream of real labels every week, and make retraining a triggered decision that passes the same gate as any other model change.

Nothing broke, and it got worse

A deployed model has a failure mode that no other part of your system has: it degrades with nobody deploying anything. The code is unchanged, the tests pass, latency is flat, error rate is zero, and the predictions are getting worse — because the world moved and the model did not.

A spam classifier trained on last year's spam meets this year's. A demand forecast trained before a festival season meets the season. A model trained on one

country's users meets an influx from another. In every case the machinery is healthy and the output is wrong.

Three different changes, often confused

- **Data drift** (covariate shift): the inputs change distribution. More mobile traffic, longer messages, a new language appearing in the text field.
- **Concept drift**: the relationship between inputs and the correct answer changes. The same message that meant "legitimate" now means "scam", because the scammers adapted.
- **Label delay**: nothing has drifted, but you will not know for weeks, because ground truth arrives late — a fraud chargeback at 60 days, a churn label at 90.

They need different responses. Data drift may be harmless if the model handles the new region of input space. Concept drift always hurts. Label delay is what makes both hard to detect quickly, and it is the reason input monitoring exists at all.

Monitor the inputs, because they are free

You can measure input distributions today, without waiting for labels. For each important feature, compute a summary on a recent window and compare it with the training distribution:

- Numeric features: mean, standard deviation, and the share falling outside the training range.
- Categorical features: the proportion of unseen categories, and the shift in the top few.
- Text or embeddings: mean embedding distance from the training centroid, or the share of inputs beyond a distance threshold.
- Missingness: the share of nulls per field. A field that suddenly becomes 30% null is usually an upstream change and it is one of the highest-yield checks in this list.

The standard statistics are the Population Stability Index and the Kolmogorov–Smirnov test. PSI has widely used rules of thumb — under 0.1 stable, 0.1 to 0.25 worth investigating, above 0.25 significant — and those thresholds are conventions, not laws; calibrate them against your own history before you page anybody.

One caution repeated by everybody who has run this in anger: with enough features and daily tests, statistical drift alarms fire constantly on differences that do not matter. Test a handful of features you have reason to care about, use a window long enough to be stable, and treat a drift signal as a prompt to look, never as an automatic retrain trigger.

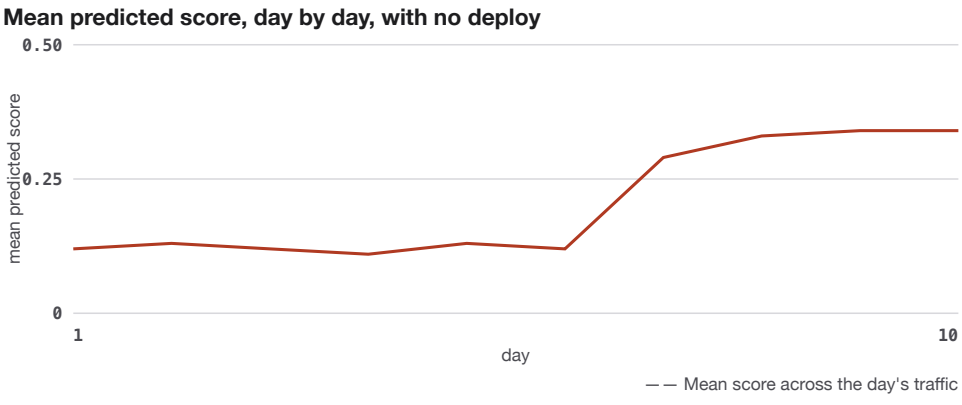
Monitor the outputs, which is cheaper still

The prediction distribution is available immediately and it moves before the labels do.

- The mean predicted score, day over day. A classifier whose average score jumps from 0.12 to 0.34 has met something new.
- The share of predictions near the decision threshold, which measures how much of your traffic is a coin flip.
- The rate of fallbacks, refusals, empty results or low-confidence answers.
- For a language model: output length, refusal rate, and the rate at which structured output fails to parse. That last one is a beautifully sensitive canary for a provider changing something under you.

Then get labels, slowly and deliberately

Input and output monitoring tell you something changed. Only labels tell you whether quality fell.



Nothing in the code changed. On day seven the classifier met something new, and its average score nearly tripled. The prediction distribution is available immediately; the labels that say whether quality fell arrive weeks later.

Three affordable sources: user actions that imply a judgement (a report, a correction, a deletion, an undo), a small stratified sample reviewed by a person each week, and delayed ground truth when it eventually arrives. Fifty labelled examples a week, chosen to over-sample the uncertain region, is a modest, sustainable practice that will detect real degradation within a month.

Keep those labelled examples forever. They become your regression set, and a set built from real production failures is worth more than a much larger synthetic one.

Retraining is a decision, not a schedule

"Retrain nightly" sounds diligent and is often a way to ship an unevaluated model every day. Retraining on a cadence is fine only if every retrained model passes the same evaluation gate, and is shipped through the same shadow-and-canary path, as a hand-made one. If it is not gated, a data pipeline bug on Tuesday becomes a production model on Wednesday.

Retrain when the evidence says to: measured quality has fallen, drift is sustained rather than a spike, or enough new labels exist to be worth learning from. Write the trigger down before you need it, and record which data snapshot each retrain used so the version lineage from the earlier lesson stays intact.

The free path

Evidently is a free, open-source library that computes drift reports and quality metrics and produces an HTML page you can look at. NannyML does the harder job of estimating performance when labels have not arrived yet. Neither is required: a weekly SQL query producing ten numbers, stored in a table and drawn on your dashboard, is a working drift monitor, and it is the version that actually gets built.

BEFORE YOU MOVE ON

A fraud model shows a stable input distribution and unchanged prediction scores, but chargebacks from two months ago show precision has fallen sharply. Which situation does this best describe?

A. Concept drift revealed by delayed labels, where the same inputs now correspond to different outcomes

B. Data drift, since the inputs must have changed for performance to fall

C. Training-serving skew introduced by a recent change to the feature pipeline

D. A monitoring failure, since input and output monitoring should always detect a real quality drop

56 · 9 min

The model that trains on its own decisions

THE ONE THING TO KEEP

A deployed model selects, censors and generates the data it will later be trained on, so log impressions and interventions, mark machine-generated content at write time, keep a small randomised slice and an untouched human-labelled anchor set — or your metrics will measure agreement with your own past decisions.

Your model changes the data it will learn from

A recommendation system shows ten items. Users click one. Tomorrow you train on clicks. But users could only click what was shown, so tomorrow's model learns that the items yesterday's model favoured are the good ones — and the items never shown accumulate no evidence at all, forever.

This is not a bug in the code. It is a property of deploying a model into the system that produces its training data, and it is one of the most reliably documented failure patterns in production machine learning. Once you see the shape you find it everywhere: fraud rules that only ever see the transactions they allowed, moderation classifiers trained on the content that was not removed, search rankers trained on the results they placed at the top, and pricing models trained on the sales their own prices caused.

Three loops worth naming

1. The selection loop. The model chooses what gets observed. The unobserved region is never corrected. Popular items get more popular; a good item ranked 400th stays there.

2. The censoring loop. The model prevents the outcome it predicts. A fraud model blocks transactions it believes are fraudulent, so those never generate a chargeback label, so the model's own successes look like a category with no

positive examples. Over time the model learns that its interventions were unnecessary.

3. The content loop. Model output becomes future input. Your assistant's answers get saved into your knowledge base; next month retrieval finds them and cites them as evidence. At industry scale the same problem appears as models trained on text generated by models, and the effect on quality is an active area of research rather than a settled question — what is not in dispute is that treating machine-generated content as independent evidence overstates how much you know.

Cheap countermeasures a small team can actually apply

- **Reserve a slice of randomness.** Show a small share of users an unranked or randomly ordered set, or allow a small share of blocked cases through when doing so is safe. That slice is unbiased data, and it is the only honest measurement of what your model is missing. This is the same idea as the holdback from the previous lesson, used for learning rather than for comparison.
- **Log what was shown, not only what was chosen.** Impressions plus position plus the model version. Without impressions you cannot correct for position bias later; with them, you can.
- **Record the intervention.** Every row should say whether the model acted on it, and how. A training set that cannot distinguish "this was allowed and was fine" from "this was blocked so we never found out" will teach the next model the wrong lesson.
- **Mark machine-generated content in your own data.** One boolean column, set at write time, forever. It costs nothing now and it is unrecoverable later — you cannot reliably tell after the fact whether a paragraph in your knowledge base was written by a person, and detectors are not accurate enough to rely on.
- **Keep a human-labelled anchor set** that is never influenced by the model, and evaluate against it. If your only ground truth comes from the deployed system, your metrics can improve while the product gets worse.

The loop that goes through your users

The subtlest version is behavioural rather than statistical. Recommendations change what people watch; the model then learns that people wanted what it showed them. Autocomplete changes what people write; the model learns that this is how people write. You cannot fully separate what users prefer from what your system taught them to prefer, and no amount of logging resolves it.

What you can do is be honest about it in how you measure. A metric that only measures engagement with what you chose to show will always look healthy. Pair it with something the loop cannot inflate: a direct question to users, a complaint rate, a retention figure over months, the rate at which people undo what the system did for them.

The operational summary

Before you train on production data, ask three questions and write the answers down:

1. Which rows exist only because a previous model chose them?
2. Which outcomes are missing because a previous model prevented them?
3. Which inputs were generated by a model, including ours?

A training set that cannot answer those three questions is not a sample of the world. It is a photograph of your own past decisions, and a model trained on it will be extremely good at agreeing with them.

BEFORE YOU MOVE ON

A fraud model blocks suspicious transactions. After a year of retraining on chargeback data, it flags fewer and fewer transactions, and the team concludes fraud has declined. What is the flaw in that conclusion?

A. Chargebacks are delayed, so the most recent months are undercounted

B. Blocked transactions never produce chargebacks, so the model's own successes appear in the data as cases with no fraud label

C. The input distribution drifted, so the model's thresholds no longer apply

D. Fraudsters adapted, making the remaining fraud harder to detect

57 · 9 min

What the page says when the model is not there

THE ONE THING TO KEEP

Model calls fail slowly, partially and expensively, so set idle timeouts on streams, validate the shape of every answer, define one cheaper fallback per feature and write down in advance whether each gate fails open or closed.

Model dependencies fail differently

Everything the failing-well lesson said about timeouts and breakers applies here, with four differences that change the settings.

1. **Normal latency is seconds, not milliseconds.** A 2-second timeout is aggressive for a database and far too tight for generation. Timeouts must be set per operation: a classification call might be 3 seconds, a long generation 60.
2. **Streaming complicates the timeout.** A stream that has produced tokens is alive even after 30 seconds. What you want is an idle timeout — no

bytes for 10 seconds — rather than a total deadline, plus a maximum total as a backstop.

3. **Failure is often partial.** The call succeeded and returned something unusable: truncated output, invalid JSON, a refusal, an empty string. HTTP 200 is not success for a model. Validate the shape of the answer before you use it, and count schema failures as errors in your metrics.
4. **Retries are expensive.** Retrying a database query costs a millisecond; retrying a generation costs cents and seconds. Retry once on a clear transient signal, and prefer falling back to something cheaper over trying the same expensive thing again.

Decide the degraded answer before you need it

For every model-backed feature, write down what happens when the model is unavailable. The options, roughly in order of how well they preserve the product:

- **Serve a cached answer.** Exact-match cache from the cost lesson doubles as an outage buffer, and `stale-if-error` semantics apply to your own cache too.
- **Fall back to a cheaper or local model.** A small model on your own hardware, or a smaller model from the same provider, is worse and available. For classification this is often nearly as good.
- **Fall back to rules.** A keyword filter, a heuristic ranking, most-recent order. Unfashionable, deterministic, and it keeps the feature alive.
- **Fall back to a second provider.** Model APIs are mostly interchangeable behind one interface, and this is one of the cheapest redundancies available anywhere in this course. Keep the prompt and the parsing shared, and test the second path monthly — an untested fallback is not a fallback.
- **Queue it.** If the answer is not needed instantly, accept the request and process later.
- **Turn the feature off** behind a flag and say so plainly on the page.

What should not happen is a 500 on a page whose main content does not need the model. The header, the article, the form: all of those work without it.

Fail open or fail closed, and why it is not a preference

When the model gates something, its unavailability forces a choice.

- A **spam classifier** that is down: failing open lets spam through; failing closed blocks legitimate users. Most products fail open for trusted accounts and closed for new ones, which is a defensible split rather than a single rule.
- A **safety check** in front of published content: failing open publishes unreviewed material to everybody. That is usually the worse outcome, so the honest answer is to hold the content and tell the author it is pending.
- A **payment risk score** that is down: failing open takes fraud, failing closed refuses revenue. This one is a business decision and should be made by whoever owns the money, in advance, in writing.

The pattern worth taking away: the answer depends on which error is more expensive and on who is affected, and it should be decided calmly and written into the runbook, not improvised at midnight by whoever is on call.

Guardrails cost latency too

If you check the model's output with another model — a safety classifier, a schema validator, a second opinion — you have added a dependency to the critical path. Two consequences: your latency is now the sum, and your availability is now the product. Run guardrails in parallel with the main call where the design allows, keep them small and fast, and give them their own timeout with a defined behaviour when they time out.

Test it deliberately

Add a drill to the list from the practising-failure lesson: block the model provider at the firewall for five minutes during working hours and watch what your product does. The first time, something will be wrong — usually an unset timeout, a fallback that was never wired, or an error page where a degraded

page should be. That is exactly the finding you want, on a Tuesday, rather than during the provider's next incident.

BEFORE YOU MOVE ON

Your generation endpoint uses a 30-second total timeout. During a provider incident, users see requests hang for the full 30 seconds and then fail, even though the stream stopped producing tokens after 2 seconds. What change addresses this directly?

A. Reduce the total timeout to 5 seconds

B. Add an idle timeout that fails when no bytes arrive for a few seconds, keeping the total as a backstop

C. Retry the request automatically when it times out

D. Disable streaming so the response either arrives whole or fails immediately

CASE STUDY · MODULE 6

Point eight nine offline, a coin toss in the office

A college in Coimbatore that scores students for dropout risk so three counsellors can spend their week on the right forty names.

The college has about four thousand students and three counsellors. Before the model there was a list, maintained by a clerk, of students who had missed a lot of classes or failed papers. The clerk was reassigned in June and the list stopped.

The IT department built a replacement. Five years of records, a gradient-boosted tree, an area under the curve of 0.89 on a held-out set that had been kept properly separate. It produces a weekly roster of forty names, ranked. It went live in July.

By the end of August the head of counselling had a complaint that was hard to argue with. The roster kept naming students who, on meeting them, were plainly fine — a topper, twice. And three students had left in that period, none of whom had appeared on any roster.

Nothing in the system was broken. There were no errors, latency was flat, the model file was the one that had been evaluated, and the tests passed. The head of IT spent a day on it and found that the numbers going into the model in production were not the numbers it had been trained on. Three of them, specifically.

`days_since_last_login` was computed in the training notebook from timestamps in UTC and in the web application from timestamps in IST. For any student whose last activity was in the evening — which is most of them — the two paths differed by one. A feature the model leaned on heavily was systematically shifted.

`attendance_pct` came, in training, from a warehouse table that a nightly job cleans and back-fills. At serving time it came from the live row, where first-year students have a null until that job has run. Serving replaced the null with zero. Training had replaced it with the column mean. Zero means "never at-

tended", which is the strongest signal the model has, and every first-year student was being handed it.

`programme` was normalised in the training pipeline — trimmed and upper-cased — and not in the serving path. So "B.Sc " and "b.sc" arrived as categories the encoder had never seen, and the encoder mapped them to index zero, which is the most common category. Not an error. A confident, quiet lie.

The decision that had to be made was not technical, and it belonged to the head of counselling. The fix would take two to three weeks.

Suspend the model. Then the counsellors need a list from somewhere, and the clerk who used to make one is in another department. Rebuilding it by hand costs about a fortnight of the counsellors' own time, which is the exact resource the model was bought to protect, in a term that has already started.

Keep using the roster while it is fixed. It is cheap and it is already in everybody's calendar. It is also close to random, and a name on that list receives a counsellor's hour that a different student does not receive. Three weeks of that is three weeks of misallocated attention with a false claim of rigour attached to it.

> STOP HERE
>
> Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They did neither, quite. The ranked roster was switched off the same afternoon, and the same page was refilled with a rule-based list the counsellors already trusted and could explain: attendance below sixty per cent, two or more failed papers, or no login for twenty-one days. That took an afternoon, it is not a model, and it carried the term.

The rebuild took four weeks rather than three. One module, `features.py`, is now imported by both the training job and the web application, so each fea-

ture is computed by exactly one piece of code. A scheduled job takes a hundred real rows, scores them through both paths, and diffs the feature vectors — not the final predictions, the features — failing the build on any mismatch. It found two more divergences in its first fortnight, both introduced by somebody editing one path. The fitted encoder, the scaler and the imputation values are serialised into the same directory as the model and loaded together, with a checksum logged at start-up beside the model version.

The rebuilt model scored 0.81 on the same held-out set. Lower than 0.89, and the first number anybody believed, because it was computed on the features the model would actually be given. It ran in shadow for a day, then as a canary on a quarter of the roster with the counsellors blind to which names came from which list, and went to everybody in November.

The test suite passed throughout. Why could tests not have caught this?

Because no piece of code was wrong. The training pipeline computed its features correctly and the serving path computed its features correctly; they simply disagreed. Skew lives in the gap between two correct implementations, so a test of either one in isolation passes. What catches it is a comparison across the two paths on the same real rows — score a sample through both, diff feature by feature, and treat any difference as a failure.

Of the three divergences, which is the most dangerous, and why?

The unseen category mapped to index zero. The timezone shift and the null handling are wrong by a measurable amount and would show up in a distribution comparison; the encoder default silently asserts that an unknown value is the most common category, produces a confident prediction, reports nothing, and gets worse over time as new programmes, plans, devices or country codes appear. A wrong number is recoverable; a wrong number that claims to be an ordinary one is not visible at all.

Why did a score of 0.81 count for more than the earlier 0.89?

Because 0.89 was measured on features the model would never actually receive, so it was a statement about a different program. 0.81 was measured on the features produced by the shared module that also runs in production. An offline number is only worth what its inputs share with the serving path, which is why the diff test and the shared feature code come before any claim about quality.

MODULE 6 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A prediction endpoint calls `joblib.load("model.pkl")` inside the handler. Beyond the wasted time, what else must change when you move it to start-up?

A. The model file has to be copied into the container image at build time
B. Readiness must stay false until the weights have finished loading
C. The handler must take a lock, since the model is now shared between requests
D. The process needs a larger thread pool to make use of the loaded model

2 Why does quantising a language model from 8-bit to 4-bit roughly double the tokens per second during generation?

A. Fewer bits means fewer arithmetic operations per multiply-accumulate
B. The KV cache shrinks, so more of the context fits in fast memory
C. Decoding is limited by memory bandwidth, and there is half as much to read
D. Smaller weights allow a larger batch, and larger batches finish sooner

3 A rented GPU costs $0.80 an hour and your product serves one request every ten seconds. What does the arithmetic say?

A. Self-hosting is cheaper, because the marginal cost of a request is zero
B. The two are close, so the decision should turn on latency instead
C. Self-hosting is dearer, because the card is paid for while it is idle
D. Self-hosting is cheaper once you include the API's per-token output charge

4 A user complains about an answer from last Tuesday. Which record makes the complaint investigable?

A. The dated model snapshot, prompt version, decoding parameters and retrieved document IDs
B. The full text of the prompt and the answer, stored against the user
C. The commit SHA of the deploy that was live at the time of the request
D. The token counts and latency, recorded per request in your own table

5 You are canarying a new model at five per cent. Why split by a stable hash of the user ID rather than per request?

A. It guarantees each group receives exactly five per cent of the traffic
B. It lets you roll back one group without redeploying the other
C. Per-request splitting doubles the inference cost during the comparison
D. A user switching between models has an incoherent experience and pollutes the comparison

6 You train tomorrow's ranking model on today's clicks. Which logging habit keeps that training set honest?

A. Recording the model version alongside each click event
B. Logging what was shown and in which position, not only what was clicked
C. Storing the full feature vector for every click that occurred
D. Keeping click events for longer than the model's retraining interval

MODULE 7

Security after the short list

The short list earlier in this course covers what compromises most small sites. This module is what comes after it, and it is mostly about authority: who may do what, which key proves it, what happens when a session or a key must be taken back, and what you do on the day something is genuinely wrong. It ends with the two situations nobody plans for — a breach, and a legal demand about one of your users.

> BY THE END OF THIS MODULE YOU CAN
>
> Enforce authorisation per object rather than per page, revoke a session or rotate a key without an outage, and run the first day of a breach or an abuse report with evidence preserved and users told the truth

58 · 9 min

Authentication is not authorisation

THE ONE THING TO KEEP

Being logged in is not permission to touch a particular row, so scope the query by the owner rather than checking the route, name each rule as a testable function, fail closed, and remember that an unguessable ID is a secret rather than an authorisation check.

Two questions that get answered in one place, badly

Authentication asks who you are. Authorisation asks whether you may do this, to this thing, right now. Most small applications answer the first carefully and the second by accident — a middleware that checks somebody is logged in, and then handlers that trust whatever ID arrives in the URL.

That gap has a name in the security literature — broken object-level authorisation — and it is consistently reported as the most common serious flaw in web APIs. It is also the easiest to demonstrate and the least likely to be caught by a test suite, because the request is perfectly valid.

```
GET /api/invoices/1041     -> your invoice
GET /api/invoices/1042     -> somebody else's invoice, 200 OK
```

No attack tool is required. Somebody changed a number.

The rule: authorise the object, not the route

A check at the router level answers "is this person allowed to use this feature". It cannot answer "is this person allowed to touch *this row*", because at routing time the row has not been fetched.

So the check belongs where the object is loaded, and the most reliable way to make that unavoidable is to put the ownership into the query itself:

```
-- not: SELECT * FROM invoices WHERE id = $1
SELECT * FROM invoices WHERE id = $1 AND account_id = $2;
```

A missing row and a forbidden row now produce the same code path, which is both safer and simpler. The alternative — fetch, then compare, then decide — works, but every handler is a place to forget it, and handlers get copied.

The same reasoning applies to writes, to nested resources (`/orders/12/items/98` needs both checks), to bulk endpoints (authorise every ID in the list, not the first one), and to anything that accepts an ID inside a JSON body where it is easier to overlook.

Guessable identifiers make it worse, and are not the fix

Sequential integer IDs let somebody enumerate your whole dataset with a loop. UUIDs make that impractical. But an unguessable ID is not an authorisation check — it is a secret that leaks through referrer headers, browser history, screenshots, shared links and support tickets. Use UUIDs for anything user-facing, and still check ownership. One is defence in depth; neither is sufficient alone.

Roles, and the thing roles cannot express

A role — admin, editor, viewer — answers a question about a person. Most real rules are about a person *and* an object: this user may edit posts *in rooms they moderate*, may read invoices *for their own account*, may delete a comment *they wrote, within an hour*.

Write those as a small number of named functions, in one file, and call them from every handler:

```
def can_edit_post(user, post):
    return post.author_id == user.identity_id or
user.moderates(post.room_id)
```

Two benefits beyond correctness. The rules become readable by a person who is not the author, and they become testable without a browser — a table of user, object, expected answer, run in CI. A permissions test file is one of the highest-value tests a small application can have, because the failure it prevents is silent.

Fail closed, and be careful what the error reveals

Default deny: an unknown role, a missing field, an unhandled case must refuse. Code that reads `if user.role == "banned": return 403` allows anyone whose role is an empty string.

On the response, there is a genuine trade-off. Returning 403 for a resource that exists tells the caller it exists — which is itself information, and for private objects it can matter. Returning 404 for both hides it, at the cost of a confusing message for a legitimate user who has lost access. Administrative surfaces in particular are usually better hidden entirely: an admin path that answers 404 to everyone unauthorised does not advertise that it is there.

Pick per surface, deliberately, and be consistent within it. The client needs to distinguish the cases it must act on — this course treats 401 as not signed in, 403 as signed in and refused, 428 as needing to agree to something first — and that only works if the codes are used honestly.

What to check on your own application this week

1. Open a resource you own, note the ID, and request the neighbouring ID from a second account. Repeat for every resource type you have.
2. Grep for `findById`, `get(id)`, `where id =` and read what happens next in each one.
3. Try the same on write and delete endpoints, and on any endpoint that takes an array of IDs.
4. Check that a user removed from a team immediately loses access to that team's objects — stale membership cached in a session is a common and unpleasant version of this bug.

Every one of those is fifteen minutes with a browser and a second account, and they find real holes in real applications with dispiriting regularity.

BEFORE YOU MOVE ON

An API loads a record by UUID and returns it to any authenticated user, on the reasoning that UUIDs cannot be guessed. Which failure does this design permit?

A. Attackers brute-forcing version-4 UUIDs across the keyspace

B. Any user who obtains a URL — from a shared link, a referrer header, a screenshot or a support ticket — gains permanent access to the object

C. UUID collisions returning another account's record

D. The database cannot index UUIDs efficiently, so the endpoint becomes slow under load

---

59 · 9 min

Sessions, and taking one back

THE ONE THING TO KEEP

Design sessions around revocation — an opaque ID looked up server-side revokes instantly, while a stateless token cannot until it expires — and rotate the ID on every privilege change, hash it at rest, and re-authenticate in front of dangerous actions.

The test of a session system is revocation

Anybody can issue a session. The question that separates designs is what happens when you need to end one immediately: a stolen laptop, a shared password, a member removed from a team, a support agent who has left, an account you have just banned.

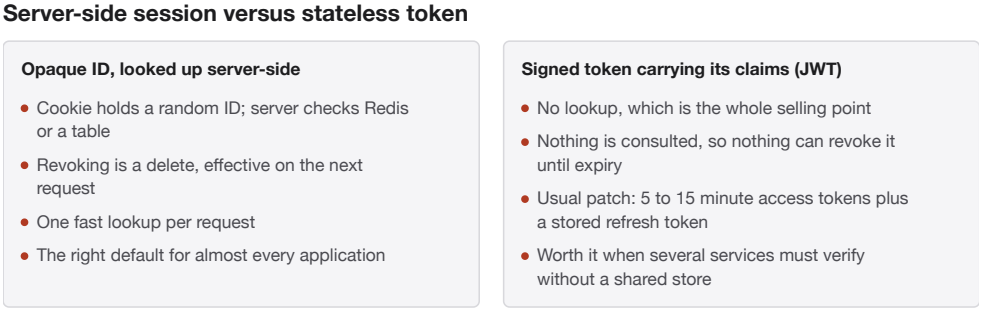
If your answer is "they stay logged in until it expires", you do not have a session system, you have a countdown.

Two designs, and the trade nobody explains clearly

Server-side sessions. The cookie holds a random opaque ID; the server looks it up in Redis or a table. Revoking is a delete. Reading costs one fast lookup per request. This is the boring design and it is the right default for almost every application, including large ones.

Stateless tokens (JWTs). The token itself carries the claims, signed. No lookup — which is the entire selling point — and therefore no way to revoke, because nothing is consulted. The usual patch is short-lived access tokens (5 to 15 minutes) plus a long-lived refresh token that *is* stored server-side, so revocation takes effect at the next refresh. That is a reasonable design, and notice what it has done: reintroduced the database lookup you were avoiding, and added a window during which a revoked user is still working.

Choose stateless tokens when you genuinely have several services that must verify without a shared store. Choose server-side sessions otherwise. "JWTs scale better" is a claim about an architecture most small products do not have, and the cost is paid in the incident where you cannot log somebody out.



The test of a session design is revocation. The stateless token's patch, short access tokens plus a stored refresh token, reintroduces the lookup it was avoiding and adds a window in which a revoked user keeps working.

The cookie attributes, and what each one prevents

```
Set-Cookie: sid=<random>; HttpOnly; Secure; SameSite=Lax;
Path=/; Max-Age=1209600
```

- **HttpOnly** keeps JavaScript from reading it, so one cross-site scripting bug does not hand over every session.
- **Secure** keeps it off plain HTTP.
- **SameSite=Lax** stops the cookie being sent on most cross-site requests, which removes the majority of cross-site request forgery. `Strict` breaks inbound links from other sites; `None` requires `Secure` and re-opens the risk, so use it only for a genuine cross-site flow.
- **A random, long ID** — at least 128 bits from a cryptographic generator. Not a counter, not a hash of the user ID, not anything derived from time.

Store a hash of the session ID rather than the ID itself, so a leaked database is not a set of working sessions. This is the same argument as hashing passwords, applied to the thing that substitutes for one.

Rotate on privilege change

Issue a new session ID at login, and again whenever authority changes — password change, email change, second factor enabled, role granted. Two reasons: it defeats session fixation, where an attacker plants a known ID before you log in, and it ensures a password change actually ends the attacker's session rather than leaving it alongside yours.

A password reset should, by default, end every other session. If you offer "log out everywhere", make sure it reaches API tokens and mobile apps too, or it is a comforting button that does nothing.

Show people their sessions

A list of active sessions — device, approximate location, last seen, with a revoke button — is perhaps two hundred lines of work and it does three things at once: it lets a user end a session you know nothing about, it makes a com-

promise visible to the only person who can recognise it, and it gives your support process something to point at. Every large service has one because they are the first thing a worried user asks for.

Expiry, honestly

There is a real tension. Short sessions are safer and annoy people into choosing worse passwords and staying signed in on shared machines. Long sessions are kind and leave more time for a stolen cookie to be useful.

A defensible middle: a sliding two-week session for ordinary use, an absolute maximum of a few months regardless of activity, and a re-authentication prompt in front of dangerous actions — changing email, deleting the account, viewing payment details — regardless of how fresh the session is. That last one is what protects the user whose unlocked laptop somebody borrowed for a minute.

The failure to plan for

Credential stuffing is the common attack and it is not a breach of your system: somebody takes a list of passwords leaked elsewhere and tries them against your login. The defences are rate limits per account rather than only per IP, a check against known-breached password lists (the free range-query API from Have I Been Pwned does this without sending you the password), and a second factor for accounts that can do damage.

When it succeeds, everything above is what you use: revoke the sessions, force a reset, show the user their session list, and tell them plainly what happened.

BEFORE YOU MOVE ON

A service issues 24-hour JWTs with no server-side store. A user's laptop is stolen and they ask to be logged out everywhere. What can the service actually do?

A. Delete the session record, ending access immediately

B. Rotate the signing key, invalidating that token and every other token in circulation

C. Nothing that takes effect before the token expires, unless a deny-list or key rotation is introduced

D. Change the user's password, which invalidates previously issued tokens

60 · 9 min

The code you did not write

THE ONE THING TO KEEP

Your dependencies run with your application's permissions, so install from a committed lockfile, delay adoption of brand-new versions, pin CI actions by commit hash and triage scanner output by whether the vulnerable path is reachable — and accept that a patient social attack defeats all of it.

Most of your application is somebody else's

A small web project pulls in hundreds to low thousands of packages once transitive dependencies are counted. Every one of them runs with your application's permissions, in your process, with access to your environment variables — which is where your database password and your API keys live.

The threat is not theoretical and it is not only about old versions with known bugs. There is a second, sharper category: packages that are compromised on purpose, published deliberately to be installed.

The four ways this actually happens

1. A known vulnerability in an old version. The ordinary case, and the one scanners find. Fixed by updating.

2. A typosquat. A package named one character away from a popular one — a hyphen moved, a letter doubled — published with an install script that reads your environment and posts it somewhere. These appear in the thousands on the large registries and are usually removed within days, which is plenty of time.

3. A maintainer account taken over. A popular package gets a new version containing something extra. This has happened repeatedly to widely used JavaScript packages, and the malicious version is often live for hours before anyone notices — hours in which every unpinned build pulls it.

4. A long social attack. The most sobering example remains the xz backdoor found in 2024: a contributor built trust in a small, critical compression library over roughly two years, became a maintainer, and inserted a backdoor targeting SSH that shipped into pre-release distributions. It was caught by an engineer investigating a 500-millisecond delay in a login benchmark. No scanner found it, and nothing about the maintainer's behaviour looked wrong until it did.

Draw the honest conclusion: tools reduce the first category well, the second and third partly, and the fourth not at all. What protects you against the fourth is a smaller dependency count and a build that cannot silently take new code.

The controls that are worth the effort

- **Commit the lockfile and install from it.** `npm ci`, `pip install -r requirements.txt` with hashes, `uv sync --frozen`, `bundle install --deployment`. A build that resolves versions freshly can pull a package published ten minutes ago.
- **Delay adoption.** A cooling-off period — do not install any version less than a few days old — removes most of the window in which a compromised release is live. Some registries and tools support this directly; a

scheduled update job that only accepts versions older than a week gets you the same thing.

- **Turn off install scripts where you can.** Much registry malware runs in a package's post-install hook. `npm ci --ignore-scripts` breaks some packages and is worth trying; know which of your dependencies genuinely need a script to run.
- **Pin CI actions by commit hash, not by tag.** A tag can be moved. In 2025 a widely used GitHub Action was modified to dump secrets from build logs, affecting thousands of repositories, and hash-pinned users were unaffected.
- **Read what you add.** For a small package, open the source before you depend on it. The number of dependencies whose entire content is thirty lines you could have written is larger than people admit.
- **Prefer fewer, larger, better-maintained dependencies** to many small ones. Every package is a maintainer you are trusting, and a package with one maintainer and no funding is a risk regardless of anybody's intentions.

Scanners, and the noise problem

`npm audit`, `pip-audit`, Trivy, Grype and Dependabot are free and worth running in CI. They will also produce a lot of findings that do not matter — a vulnerability in a code path you never call, a denial-of-service issue in a build-time tool, a critical rating for something exploitable only with local access.

Unactioned alerts train people to ignore alerts, which is the same failure as noisy paging. Triage on two questions: does the vulnerable code run in production, and can untrusted input reach it. Fix what passes both immediately, batch the rest into a monthly update, and write down the ones you deliberately accepted with a date to revisit. A short honest list beats a long ignored one.

Where your secrets sit during all this

Build systems are attractive targets precisely because they hold credentials with the power to publish. Give CI the narrowest tokens that work, scope them per repository, prefer short-lived cloud credentials over long-lived keys,

and never expose deployment secrets to a workflow triggered by a pull request from a fork. That last one has been the entry point in several public incidents.

The five-minute version

Commit the lockfile, install with the frozen command, pin your CI actions by hash, run one scanner weekly, and update deliberately on a schedule instead of reactively at midnight. That is most of the available protection, and it costs a morning to set up once.

BEFORE YOU MOVE ON

A team pins every dependency version exactly in `package.json` but runs `npm install` in CI without committing the lockfile. Which risk remains open?

A. None — exact versions in the manifest fully determine what is installed

B. Transitive dependencies still resolve to their newest permitted versions, so a freshly compromised sub-dependency enters the build

C. Exact pinning prevents security patches, so old vulnerable versions persist indefinitely

D. `npm install` ignores the manifest in CI environments and uses the registry's latest tags

61 · 8 min

Changing a key while everything is using it

THE ONE THING TO KEEP

Rotate by overlapping — both keys valid, deploy the new one, watch the old one go quiet, disable before deleting — and give keys an identifier so you can tell who is still using the old one instead of guessing.

Rotation is a procedure, not a value

Everybody knows keys should be rotated. Far fewer teams have done it, because the naive version — replace the value, restart, hope — takes the site down when something you had forgotten was still using the old one.

The result is that keys quietly become permanent. A credential shared with a contractor in 2023 still works. The database password is in four places, one of which is a cron job nobody has opened. And when a leak forces a rotation, it happens under pressure, badly.

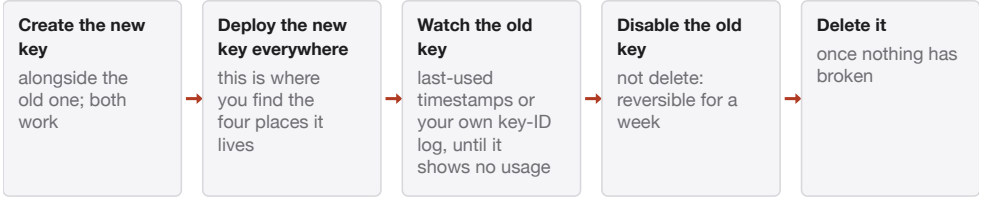
The fix is to make rotation a rehearsed, dull procedure. It has one core idea: **for a period, both keys are valid.**

The overlap window

1. **Create the new key** alongside the old one. Both work.
2. **Deploy the new key** to everything that uses it. This is where you discover the four places, and the discovery is safe because the old key still works.
3. **Watch** until the old key shows no usage. Most providers expose last-used timestamps or usage-per-key metrics; if yours does not, log which key ID your own code used.
4. **Disable, do not delete**, the old key. Reversible for a week.
5. **Delete it** once nothing has broken.

Step 3 is the one people skip, and it is the only one that turns rotation from a hope into a fact.

Rotation with an overlap window



For a period both keys are valid, so the forgotten cron job is discovered safely rather than by an outage. Watching the old key go quiet is the step that turns rotation from a hope into a fact.

Give your own keys an identifier

For keys you issue — session signing keys, webhook signatures, API keys for your customers — put a key ID in the artifact itself. A JWT header carries `kid`; a webhook header can carry a key version; your own API keys can be prefixed, as in `ak_live_2_xxxxx`.

With an ID, verification becomes: look up this key by ID, in a small set of currently valid keys. Rotation becomes: add a key, start signing with it, keep verifying with both, remove the old one when nothing presents it. Without an ID, you must try every key on every request, which works but tells you nothing about who is still on the old one.

The different things people mean by rotate

- **A provider's API key.** Usually easy: most providers let you have two active keys precisely so you can overlap. Check before you assume.
- **A database password.** Create a second user with the same grants, migrate, drop the first. Changing the password on a single user in place has no overlap window and will drop live connections.
- **A signing key.** Needs the key ID design above, plus a verification period long enough to cover the longest-lived token you have issued.
- **A cloud provider's long-lived credentials.** The real answer is to stop having them: short-lived credentials issued to a workload identity expire on their own, and rotation stops being an event. This is available on every

major cloud and in GitHub Actions via OIDC, and it removes the most dangerous class of secret entirely.

- **A key you have just leaked.** Different procedure: no overlap, revoke first, apologise second. The leak lesson earlier in this course covers why scrubbing history is not a substitute.

Write the runbook while it is calm

One page per credential type, in the repository, listing: where the key lives, everything that consumes it, the exact commands, how to verify the new one works, and how to roll back. Ten lines each.

Then run one rotation per quarter, deliberately, on a Tuesday morning. Not because the key is compromised, but because the run finds the forgotten consumer while nothing is wrong. This is the practising-failure lesson applied to credentials, and it is the cheapest drill on that list.

What good looks like

- Every secret has an owner, an expiry date and a documented rotation procedure.
- No credential is more than a year old, and anything that touched a laptop that left the company is already rotated.
- Rotation is a fifteen-minute task somebody other than the author can perform from the runbook.
- Nothing important depends on a key that cannot be overlapped — and where one does, you know which one it is, and that outage is planned rather than discovered.

BEFORE YOU MOVE ON

A team rotates a webhook signing key by replacing the secret in both the sender and receiver during a deploy. Some webhooks are rejected for several minutes afterwards. What explains it?

A. Clock skew between sender and receiver invalidated the signature timestamps

B. The receiver cached the old key and needed a restart to pick up the new one

C. Messages signed with the old key were still in flight or being retried when the receiver stopped accepting it

D. The new key was shorter, so signatures failed length validation

62 · 9 min

Bots, scrapers and the flood you cannot absorb

THE ONE THING TO KEEP

Volumetric floods are handled upstream by a CDN you did not write, but application-layer abuse is yours: cache the public pages, put quotas and challenges in front of the expensive ones, limit by account rather than address, and log every block because the people you wrongly refuse cannot tell you.

Most of your traffic is not people

Publish anything and automated traffic arrives. Measurements of the web as a whole put the automated share somewhere near or above half, and for a small site the proportion is often higher, because you have no audience yet and the crawlers do not care.

It is worth separating the kinds, because the response differs:

- **Search crawlers.** Wanted. Obey `robots.txt`, identify themselves, crawl politely.

- **Dataset and AI crawlers.** A newer, heavier category. Some obey `robots.txt`, some do not, and several have been observed requesting the same pages repeatedly. Whether you want them is a decision, not a given.
- **Scrapers of your content.** They want your data, and they will rotate addresses and pretend to be browsers.
- **Credential stuffers and signup abuse.** Covered by the earlier rate-limit and session lessons.
- **Volumetric floods.** Someone trying to make you unavailable, or a badly written client doing it accidentally.

The three layers, and which is yours to fix

Layer 3 and 4 — packet floods. Gigabits of traffic aimed at your address. You cannot absorb this on an origin server; the pipe fills before your code runs. It is handled upstream, by your provider or a CDN, and every major CDN including free tiers does it automatically. Nothing you write matters at this layer.

Layer 7 — application floods. Requests that look legitimate but hit your most expensive endpoint. Search, report generation, a model call. This is where a small site actually falls over, and this layer is yours: cache aggressively, rate-limit per account and per endpoint, and make sure your most expensive path is not the most exposed one.

Application logic abuse. Signup floods, coupon farming, comment spam, free-tier exhaustion. Not a volume problem — a rules problem. Fixed with quotas, verification tiers and cost controls, which is why the identity ladder in this project's own design exists.

Three layers of unwanted traffic, and whose job each is

Application logic abuse	signup floods, coupon farming, free-tier exhaustion: a rules problem, fixed with quotas and identity tiers
Layer 7: application floods	legitimate-looking requests to search, reports or a model call: yours to fix with caching and per-account limits
Layer 3 and 4: packet floods	gigabits at your address, absorbed by the provider or CDN before your code runs

The pipe-filling floods are handled upstream by a CDN you did not write. The layer that actually takes a small site down is the one that hits your most expensive endpoint, and that layer is yours.

What actually works, ordered by effort

1. **Put a CDN in front.** The single biggest step, free at the entry tier from several providers, and it removes both the volumetric layer and most of the cacheable load.
2. **Cache the expensive public pages.** A scraper hitting a cached page costs you nothing. A scraper hitting an uncached search endpoint costs you a database query each time.
3. **Rate-limit by account and by endpoint, not only by IP.** Addresses are cheap and shared; a single mobile network address may carry thousands of legitimate users, which is exactly why blocking by IP produces collateral damage.
4. **Make the expensive thing require an account.** Anonymous access to a cheap, cached page; anything that costs real money behind a login and a quota.
5. **Challenge rather than block.** A proof-of-work or interactive challenge — Cloudflare Turnstile, hCaptcha, mCaptcha — costs a real user a moment and a bulk scraper a great deal. It is also the honest place to say that these degrade accessibility for some users, so put them in front of abuse-prone actions, not in front of reading.

robots.txt is a request, not a fence

`robots.txt` and `noindex` are conventions honoured by well-behaved crawlers. They stop nothing that does not want to be stopped, and they have

no legal force in themselves. If content must not be taken, the control is authentication, not a text file.

That said, the file is still worth writing carefully: name the paths you do not want crawled, and add a `Crawl-delay` or explicit rules for the heavy agents. Several CDNs now offer a one-click block for known AI crawlers. Whether to use it is a product decision — a site that wants to be found may want to be in those datasets, and a site whose content is its product may not.

Blocking, and its costs

Every blocking mechanism has a false-positive rate, and the people it blocks cannot tell you, because they are blocked. Three habits keep this honest:

- **Log every block** with the reason, and review a sample weekly. If you find real users, your rules are too tight.
- **Prefer degradation to denial.** Serve a cached copy, slow the response, or require a challenge rather than returning 403.
- **Give a way back.** A support email on the block page, an appeal route, and an unblock that a human can perform in a minute.

Blocking whole countries, ranges or ISPs is tempting during an incident and is usually a mistake left in place for years. If you do it under pressure, put an expiry on the rule the same day.

During an actual flood

The order that works: turn on your CDN's attack mode or equivalent, cache everything cacheable including HTML for the duration, disable the most expensive endpoints behind a flag, raise your rate limits' strictness, and only then start analysing patterns. Diagnosis is much easier from a site that is up.

And afterwards, the useful question is not "who did this" — you will rarely know — but "which endpoint let a hundred requests a second hurt us". That endpoint is your finding, and caching or a quota is your fix.

BEFORE YOU MOVE ON

A site adds a strict per-IP rate limit and reports of "too many requests" arrive from an entire university and one mobile network. What does this reveal about the control?

A. The limit is set too low and simply needs a higher threshold

B. Many legitimate users share a single public address through NAT, so an address is a poor proxy for a person

C. The users are behind a proxy that strips the forwarded header, so all requests appear to come from the proxy

D. Rate limits should never be applied to anonymous traffic

63 · 9 min

Taking money without holding card numbers

THE ONE THING TO KEEP

Let the processor collect the card so raw numbers never reach you, grant access on a signature-verified webhook rather than a redirect, deduplicate by event ID, and send an idempotency key on every charge so a dropped connection cannot bill somebody twice.

The rule that decides everything else

Do not let card numbers touch your servers.

Every other decision about payments follows from that one. Use a processor — Stripe, Razorpay, Adyen, PayPal, Paddle, whichever fits your country — and let their hosted checkout or their JavaScript element collect the card. The card details go from the user's browser to the processor. Your server receives a token, and the token is useless to anyone who steals it.

The practical reason is scope. Card industry rules impose a large compliance burden on anybody whose systems handle raw card data, and a very small one on merchants who fully outsource collection. Building your own card form does not just add security risk; it adds an audit obligation that a small team cannot carry. There is no version of this where writing the form yourself is the sensible choice.

What you may store: the token, the last four digits, the card brand, the expiry month for display, the processor's customer ID. What you must never store: the full number, the CVV — which may not be retained after authorisation under any circumstances — or the magnetic stripe data.

The webhook is the part that goes wrong

Payment flows are asynchronous. The user is redirected back to your site before the payment is confirmed, and the confirmation arrives separately as a webhook. Three consequences that cause most payment bugs:

1. Never grant access on the redirect alone. The return URL is under the user's control. Grant on the webhook, or on a server-side check of the payment's status.

2. Verify the signature. Every processor signs its webhooks with a shared secret. An unverified webhook endpoint is an unauthenticated "give this user a subscription" API, and it is on the public internet.

```
event = stripe.Webhook.construct_event(
    payload=request.data,
    sig_header=request.headers["Stripe-Signature"],
    secret=os.environ["STRIPE_WEBHOOK_SECRET"])   # raises on
tampering
```

Use the raw request body for verification. A framework that parses and re-serialises the JSON changes the bytes and every signature fails — a very common half-day of confusion.

3. Expect duplicates and reordering. Processors retry until they get a 2xx, so the same event will arrive twice. Store the event ID and ignore ones

you have already processed. Events can also arrive out of order, so make each handler decide from the current state rather than assuming the previous event has been seen.

Respond quickly — acknowledge, then do the work in a queue. A slow webhook endpoint gets retried, which produces the duplicates you are now handling.

Idempotency, in the other direction

When you call the processor to charge, send an idempotency key: a value you generate per logical operation. If the network drops after the charge is made but before you get the response, retrying with the same key returns the original charge rather than making a second one.

Without it, the retry lesson's advice — do not retry non-idempotent operations — means you must not retry a charge at all, and you are left with a payment in an unknown state and a customer who has been debited.

The states nobody designs for

Build the schema so these are representable, because they will happen: pending (initiated, no outcome yet), failed, succeeded, refunded, partially refunded, disputed, and expired. A boolean `paid` column will not survive contact with a chargeback.

Record the processor's own IDs for every object, and reconcile: a nightly job comparing your records against the processor's list of transactions for the day, reporting anything that exists on one side only. This catches missed webhooks, duplicated grants and, occasionally, a genuine bug that has been quietly giving away subscriptions.

Testing, and one uncomfortable truth

Every processor has test cards for success, decline, insufficient funds and the authentication challenge. Use them, and use the CLI tools that replay webhooks to your local machine so you can develop the asynchronous half properly.

The uncomfortable truth is that the flows you cannot fully test are the ones that hurt: a dispute months later, a refund crossing a subscription renewal, a currency conversion, a failed renewal on an expired card. Write down what your system should do in each case before you launch, even if you cannot rehearse them.

Regional reality, stated plainly

Payment rules differ by country in ways that change your design and cannot be papered over: strong customer authentication requirements in Europe, mandated tokenisation and specific recurring-payment rules in India, sales tax and VAT registration thresholds that vary by jurisdiction. A merchant of record — a service that sells on your behalf and handles tax — costs a higher percentage and removes a genuine administrative burden for a small team selling internationally.

This is a real trade-off with money on both sides, and the right answer depends on where you and your customers are. It is also the point where the honest advice is to read your processor's country documentation and, past a certain revenue, ask an accountant. Nothing in this lesson is legal or tax advice.

BEFORE YOU MOVE ON

An app grants a subscription when the user is redirected to `/success?session_id=...` after checkout. What is the flaw?

A. The session ID may expire before the redirect completes, causing lost grants

B. The redirect happens in the user's browser, so anybody can request that URL and receive a subscription without paying

C. Redirect URLs are logged by the CDN, exposing session IDs to anybody with log access

D. Webhooks arrive before the redirect, so the grant would be applied twice

64 · 8 min

The tools that can do anything

THE ONE THING TO KEEP

The admin panel is unlimited authority with the fewest tests, so gate it with an explicit allowlist that answers 404, require a reason on every action, write an append-only audit log in the same transaction, and hold destructive powers as break-glass rather than by default.

The most dangerous page you will write

Your admin panel can read every message, change any record, refund any payment and delete any account. It was built quickly, it has fewer tests than the rest of the application, and it usually has no audit trail. It is also the page an attacker most wants, and the page from which an internal mistake does the most damage.

A useful way to hold it: an admin surface is not a feature, it is a set of powers granted to people, and every power needs a reason, a record and a limit.

Fail closed, and return nothing

Access should be an explicit allowlist — a list of accounts, checked on every request, defaulting to refusal. Not a role field that could be empty, not a check on an email domain, not the absence of a flag.

And for an unauthorised caller, answer 404 rather than 403. A 403 confirms the path exists, which is exactly what somebody probing wants to learn. This is a rare case where hiding is worth the confusion, because the audience for the truthful answer is nobody legitimate.

Put a second factor in front of it. If your identity provider supports it, require re-authentication for the admin session specifically, so a stolen ordinary session does not carry admin power.

Log every action, permanently

An admin audit log is a different thing from application logs: append-only, retained far longer, and containing who did what to which object at what time, with the before and after values where practical.

```
{"ts":"2026-09-
06T11:20:04Z","actor":"handle:priya","action":"account.suspend"
,
 "target":"identity:8812","reason":"report #4471","before":
{"status":"active"},
 "after":{"status":"suspended"},"request_id":"9ab3f1"}
```

The `reason` field is worth insisting on, and worth making mandatory. It changes behaviour: an action that must be justified in a sentence is an action somebody thinks about first. It is also the difference between a defensible moderation decision and an unexplained one when the affected person asks.

Write the log before performing the action, or in the same transaction. A log written afterwards is missing exactly the entries where something went wrong.

Impersonation, which you will eventually want

"View as this user" is genuinely useful for support and genuinely dangerous. If you build it:

- Log the start and end, with the reason.
- Make it visible in the interface, to both parties where possible, so nobody forgets which account they are in.
- Keep it read-only unless there is a specific reason not to, and log every write separately if there is.
- Time-limit the session.
- Never impersonate to read private content without a documented reason — a support ticket from that user, or a legal obligation. "Curiosity" is how staff-access scandals begin at every company that has had one.

Least privilege for the humans, too

Not everybody who needs admin needs all of it. Three tiers usually cover a small team: read-only support, moderation actions, and destructive or financial operations. The third tier should be a short list.

For genuinely dangerous operations — bulk deletion, database access, refunds above a threshold — the pattern is break-glass: the power exists, it is not held by default, obtaining it is a deliberate act that is logged and announced to the team, and it expires. This is the same idea as a rotating key, applied to authority.

Do not run destructive operations from a text box

An admin page with an arbitrary SQL box is a shell on your production database with a nicer font. If people need to run queries, give them a read-only database user, a saved-query interface, or a proper tool — Metabase is free and does this well — and keep writes in named operations with a confirmation and a log entry.

For bulk actions, three habits: preview what will change and show the count before executing, require typing the count or the object name to confirm, and make the operation reversible or at least recoverable from the audit log.

Review who has access, on a schedule

Access accumulates. A contractor, a former colleague, a personal account added during an incident. Once a quarter, print the list of admins and confirm each one line by line, then remove the rest. Add the same review for API tokens, database users and cloud accounts.

This is dull and it consistently finds something, which is why it belongs on the same quarterly calendar as the restore drill and the key rotation.

BEFORE YOU MOVE ON

An admin panel writes its audit entry after successfully performing each action. Which class of event is systematically missing from the log?

A. Read-only views, which are not actions and therefore not logged

B. Actions by accounts whose access was later revoked

C. Actions that partially applied and then failed, which are exactly the ones an investigation needs

D. Bulk actions, since one entry covers many objects

65 · 10 min

The first day of a security incident

THE ONE THING TO KEEP

In a security incident, preserve evidence before you clean anything, contain narrowly and rotate credentials, establish scope from logs rather than assumptions, and find out which notification clock applies to you before the day you need to know.

It does not look like the films

A security incident usually starts as something small and confusing. A user says their account posted something they did not write. A cloud bill has an unfamiliar region on it. A support agent notices two accounts sharing one email. A researcher emails you a screenshot.

The outage runbook from earlier in this course does not fit, because the priorities are different in one specific way: **you must preserve evidence while you stop the damage**, and the instinct to "clean it up" destroys the information you will need for every decision afterwards.

The first hour

1. **Write down the time and what you saw.** Start a document. Timestamp everything from here.
2. **Preserve before you change.** Snapshot the machine, export the relevant logs to somewhere the incident cannot reach, and take a copy of the affected database. Do not rebuild the server yet. Once logs roll over — often in days — the answer to "how did they get in" is gone permanently.
3. **Contain, narrowly first.** Revoke the compromised credential, disable the affected account, block the specific access path. Wholesale shutdown is available and expensive; use it when narrow containment is not working.
4. **Rotate credentials that could plausibly be exposed**, in the order the earlier rotation lesson gave — new key, deploy, watch, disable.
5. **Get a second person involved.** Not for expertise; for the decisions. Alone at 2am is when people delete things they needed.

What not to do: log into the compromised machine and start poking, which contaminates timestamps; announce a cause in the first hour; or promise users that nothing was taken before you know.

Establish scope with evidence, not with hope

The question everybody wants answered is "what did they get". Answer it from logs, not from reasoning about what should have been possible.

- Which credential was used, and what could that credential reach?
- What queries or requests were made with it, and how much data left?
- When did it start? The gap between the first sign and your detection is the number that matters most, and it is usually longer than people expect.
- Is the access still live? Assume yes until you have positive evidence otherwise.

If you cannot establish scope, say so. "We do not yet know" is a legitimate finding and a far better public statement than a confident claim you later withdraw.

The obligations, in shape rather than in detail

This is where the honest answer is that the rules differ by jurisdiction and by the kind of data, and this course cannot tell you what applies to you.

The shape, roughly, in several regimes: if personal data is exposed, a regulator may have to be told within a short window — 72 hours from becoming aware is the well-known figure under the European regime, and other countries have their own periods, some shorter for particular sectors. Affected individuals often have to be told directly when the risk to them is high. Some sectors — health, finance, children's data — carry stricter duties. Contracts with business customers frequently impose faster notification than the law does, and people forget to read those.

What follows for you, practically:

- Find out **before an incident** which regimes you are under and what the clock is. Write it in the runbook with the actual deadline.
- Keep a record of when you became aware, because the clock starts there and you will be asked.
- If personal data of real people may be exposed, take advice from somebody qualified in your jurisdiction. Nothing here is legal advice, and this is one of the few places in this course where the stakes justify professional help.

Telling people, without making it worse

When you notify, say: what happened, what data was involved, what it means for the reader, what you have done, what they should do, and when you will update again. Plain language, no minimising verbs, no "sophisticated attack" unless it genuinely was.

The reputational damage in published breach cases is rarely from the breach itself. It comes from the delay, the initial denial, or the phrase that turns out to be untrue. A short, early, accurate notice that says "we do not yet know the full scope" ages far better than a confident one that ages badly.

If passwords were involved, force a reset and end all sessions — the session lesson's revocation design is what makes that possible in minutes rather than days.

Not every incident is a breach

Keep the categories separate, because they need different responses and different words:

- **Credential stuffing** — attackers using passwords leaked from another service. Your system was not breached; individual accounts were. Say that precisely, help the affected users, and add the defences from the sessions lesson.
- **A vulnerability report with no evidence of exploitation.** Fix it, thank the reporter, and check your logs for use before deciding whether anybody needs to be told.
- **An internal mistake** — a public bucket, an over-broad query in an export. Still a data exposure, still possibly notifiable, and the containment steps are the same.

Afterwards

Run the postmortem as usual, with two additions: the detection gap in hours, and an honest account of what evidence you did not have and now will. Most security post-incident actions are unglamorous — turn on logging you did not have, remove a credential that should not have existed, restrict a permission that was broader than the job needed. Those are the ones that make the next one smaller.

BEFORE YOU MOVE ON

You discover an attacker has been using a leaked API key. The instinct is to rebuild the compromised machine immediately from a clean image. Why is that a poor first move?

A. Rebuilding does not revoke the key, so access continues

B. The clean image may contain the same vulnerability, so the attacker returns

C. It destroys the logs and system state needed to establish how they got in and what they reached

D. It causes downtime that must be reported to the regulator

66 · 9 min

The email about one of your users

THE ONE THING TO KEEP

Have one monitored address, preserve the evidence before you act on any serious report, verify who is really asking and give only what is demanded, and learn which copyright and disclosure regime applies to you before the first notice arrives rather than during it.

An inbox you have not prepared for

At some point a message arrives that is not about your code. Somebody says a user is impersonating them. A company says your user posted their copyrighted material. A parent says a child is being harassed. A police force or a court asks for data about an account. Somebody wants their own data deleted, and somebody else wants that same data preserved.

These arrive to whichever address is on your site, usually at an awkward moment, and the cost of improvising is high — for you and for the person the message is about. The preparation is small and it is mostly done in advance.

Have a place for it to land

One published address for abuse reports and legal notices, monitored by more than one person, with an auto-acknowledgement giving a reference number and a realistic timeframe. Do not use a personal address, and do not make it the same as support, or serious notices will queue behind password resets.

Write one page describing what you do when a report arrives: who reviews it, what evidence you collect, what actions are possible, and how the person reported is told. Publish the summary. A process nobody can see looks arbitrary from both sides.

Preserve first, decide second

The first action on any serious report is the same as in a security incident: keep the evidence. Snapshot the content, the account state, the timestamps and the relevant logs, before any moderation action changes them, and store that snapshot outside the normal deletion path.

This matters because your ordinary retention rules and a user's deletion request will otherwise remove the exact material the complaint concerns. This project's own design handles that with a legal-hold record: content and identifying details preserved as a separate row that survives deletion of the account and the content, held under a legal obligation rather than kept for convenience. The general principle transfers — a preservation obligation is a documented exception to your retention rules, applied narrowly, not a reason to keep everything.

Copyright notices, where the law genuinely differs

A notice claiming your user has posted somebody else's work is common, and the correct handling depends entirely on where you are.

The shape of the disagreement, stated fairly: the United States regime offers platforms a safe harbour conditioned on removing material expeditiously on a valid notice, with a counter-notice procedure that can restore it. The European regime places different obligations on some platforms, including efforts to prevent re-upload for certain services, and its implementation varies

by member state. India's rules impose their own timelines and intermediary conditions. Several countries are actively revising these rules, and how any of them apply to material generated by a model is unsettled — courts in multiple jurisdictions are hearing the question now, and the answers are not in yet.

What that means for you: identify the regime you operate under before you receive your first notice, follow it, and do not invent a policy on the day. Nothing in this course is legal advice, and this is a subject where a wrong assumption is expensive.

What you can do regardless of jurisdiction: require notices to be specific, keep a record of every one, tell the affected user what was removed and why unless you are legally barred from doing so, and offer a route to dispute it. Automatic removal on any unverified claim is how honest users get silenced by whoever complains loudest.

Requests from authorities

The rules differ by country and the specifics matter, so the general habits are what to take away:

- **Verify the request.** Confirm it came from the body it claims, through a channel you can check. Impersonation of law enforcement to obtain user data is a documented technique and has succeeded against large platforms.
- **Read what is actually demanded.** A request to preserve data is not a request to hand it over. Preservation is usually the immediate obligation; disclosure typically needs its own legal instrument.
- **Give only what is asked for.** Over-disclosure is the common mistake and it is your user's privacy being spent.
- **Get advice** when the request is broad, unusual, or from another country. This is the second place in this module where a professional is the right answer.
- **Tell the user** if you are permitted to. Some orders prohibit it; where they do not, notifying is generally the right default and many platforms commit to it publicly.

Emergencies are different

A credible threat to life — a suicide note, a specific threat of violence, material involving a child — is not a process question. Have the emergency route written down: which authority you contact in your country, what you preserve, and who can act at three in the morning. For the specific case of child sexual abuse material, most jurisdictions impose reporting duties on platforms and there are national bodies that receive those reports; find yours before you need it.

Keep the count

Once a quarter, count what arrived: reports by type, actions taken, appeals, reversals. Two reasons. It tells you whether your moderation is accurate, since a high reversal rate means the first decision is wrong too often. And several regimes now require transparency reporting past a size threshold, so the habit of counting is easier to start early than to reconstruct later.

BEFORE YOU MOVE ON

A message arrives claiming a user posted copyrighted material. The user has meanwhile requested account deletion under your privacy policy. What is the correct first action?

A. Complete the deletion, since a privacy request takes precedence over a third-party claim

B. Snapshot and preserve the relevant content and records outside the deletion path, then handle both requests

C. Remove the content immediately and close the report as resolved

D. Suspend the deletion request until the complainant provides a court order

CASE STUDY · MODULE 7

Receipt number 4412

A school in Pune during fee week, after a class-nine student changes one digit in a URL and finds another family's receipt.

The school takes fees online from about nineteen hundred families. Fee week runs Monday to Friday, with a late fee after that, and the portal is the only online route — the alternative is a counter in the office that manages perhaps two hundred families a day.

On Tuesday afternoon a student in class nine looked at the address of his own receipt, `/receipt/4411`, and typed `4412`. He got a receipt belonging to a family he did not know: the parent's name, their mobile number, the amount paid, the date, and the last four digits of the card. He showed his class teacher, who took him to the office, who telephoned the vendor who built the portal.

The portal checks that somebody is logged in. A middleware does it, on every route, correctly. What it does not check is whether the logged-in parent has anything to do with the receipt being requested. The handler loads the row by ID and renders it. This is broken object-level authorisation, and it is consistently reported as the most common serious flaw in web APIs, largely because nothing about the request is invalid. No tool is required. Somebody changed a number.

The fix is small and everybody could see it: put the ownership into the query, so the row is fetched by ID and account together and a forbidden row and a missing row take the same code path. But the same pattern appears in four places — receipts, transport records, the term invoice, and a bulk endpoint that takes a list of IDs and, in the version written last year, checks the first one.

The vendor found the fourth of those by doing what the lesson recommends and nobody does: opening one object he owned, noting the identifier, and requesting the neighbouring identifier from a second account. Fifteen minutes with a browser and two logins found three of the four holes. The fourth need-

ed a grep for every place a record is loaded by identifier, which produced eleven call sites, of which four went straight to rendering.

One more thing was true and made the week worse. Nineteen hundred receipts is nineteen hundred parents' names, mobile numbers and payment amounts, walkable in a loop by anybody with an account — and any parent at the school can get an account, because that is what the portal is for. The exposure was not one receipt. It was the fee register of the whole school, available to every family in it.

That is where the decision sat, on Tuesday evening, with the late-fee deadline on Friday.

Take the portal offline until all four are fixed and tested. Nothing further leaks. Nineteen hundred families lose the online route for at least a day, the counter cannot clear the backlog before Friday, and the school would have to waive the late fee for everybody, which the trustees would have to approve and which sets a precedent they had refused twice before.

Patch it live, during fee week. The change is a few lines per handler. It is also a rushed change to the code path that takes money, in the week that takes most of the year's money, and a mistake there is worse than the leak.

A second decision was harder and nobody wanted it. The school did not know whether anybody had done this before the student did. The access logs would say — and they roll after fourteen days.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

Before any code was changed, the vendor exported the access logs for the receipt routes to a separate account, because the answer to "did anybody else do this" exists only in evidence and the retention clock was running. That took

twenty minutes and it turned out to matter: six weeks earlier, forty-one sequential requests from one address in under a minute, walking upwards through receipt numbers. A scraper, not a parent. No other pattern in the window.

They did not take the portal down. One change went live at half past nine that night: every receipt query became `WHERE id = $1 AND account_id = $2`, shipped behind the existing kill switch so it could be reverted in seconds, and verified before and after with two real parent accounts on two phones. The other three object types were patched over the following two days in the same shape, one per deploy. The bulk endpoint was changed to authorise every ID in the list rather than the first.

The school then wrote to all nineteen hundred families. What happened, what was visible, that they had found one earlier instance and what it appeared to be, what had been changed, and an address that a named person reads. They did not write "sophisticated attack", because it was not one. Two families replied angrily and both were telephoned.

Afterwards, three things stayed. A permissions test file — a table of user, object and expected answer, run in CI — which started at twenty-two rows and found one more hole on the day it was written. New URLs use UUIDs, and the ownership check was kept, because an unguessable identifier is a secret that leaks through referrer headers, screenshots and shared links rather than an authorisation check. And the admin surface moved behind an explicit allowlist that answers 404 to everybody else.

Would switching receipt IDs from sequential numbers to UUIDs have fixed this?

No. It would have stopped somebody enumerating the whole set with a loop, which is worth having, but an unguessable identifier is a secret and not a permission. It leaks through referrer headers, browser history, screenshots, forwarded links and support tickets, and once it has leaked the request is still authorised. The two are defence in depth together: use opaque identifiers and still scope every query by the owner.

Why export the logs before patching anything?

Because the patch changes nothing about the past, but the retention window keeps expiring — these logs rolled at fourteen days. Scope has to be established from evidence rather than from reasoning about what should have been possible, and "did anybody else use this" is answerable only while the records exist. Preserving first also mirrors the security-incident sequence: keep the evidence somewhere the incident cannot reach, then contain.

Why is a permissions test file unusually valuable for this class of bug?

Because the failure is completely silent. The request is well formed, the session is valid, the query succeeds, the page renders, the status is 200, and nothing appears in any error feed. There is no exception for a monitor to catch and no crash for a supervisor to restart. A table of user, object and expected answer, run in CI, is the only thing that asserts the rule at all — and it runs without a browser, so it is cheap enough to extend every time a new object type is added.

MODULE 7 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 `GET /api/invoices/1042` returns somebody else's invoice to a signed-in user. Which change removes the whole class of bug most reliably?

A. Check the invoice's owner after loading it and return 403 when it differs
B. Move the check into middleware that runs before every invoice route
C. Fetch the row by its ID and the caller's account together, in one query
D. Replace the sequential identifiers with UUIDs that cannot be guessed

2 A team chooses stateless tokens, then adds short-lived access tokens with a stored refresh token so they can revoke. What has that achieved?

A. Revocation without any server-side state, as originally intended
B. The database lookup is back, plus a window where a revoked user still works
C. Immediate revocation at the cost of one extra request per page load
D. Nothing, because refresh tokens cannot be revoked any more than access tokens

3 Why pin third-party CI actions by commit hash rather than by version tag?

A. Hashes resolve faster, so the pipeline spends less time on checkout
B. Tags are resolved at runtime and may fail if the network is unavailable
C. A tag can be moved to point at different code without changing its name
D. Hash-pinned actions are cached by the runner and reused between jobs

4 In a key rotation with an overlap window, which step do people skip, and what does skipping it cost?

A. Watching the old key's usage fall to zero before disabling it
B. Creating the new key before deploying it anywhere
C. Disabling the old key before deleting it permanently
D. Recording the rotation in the runbook once it has finished

5 Your payment webhook verifies its signature, but every event fails verification in production and passes locally. What is the usual cause?

A. The webhook secret was copied from the test environment by mistake
B. The events arrive out of order, so the signature covers a different payload
C. A middleware parses and re-serialises the JSON, changing the bytes signed
D. The processor rotated its signing key without publishing the new one

6 Why should an admin path answer 404 rather than 403 to an unauthorised caller?

A. A 403 tells the caller the path exists, which is what probing is for
B. A 404 is not logged by most proxies, so it produces less log noise
C. A 403 invites the client to retry with credentials, creating a loop
D. A 404 is cached by CDNs, so repeated probes never reach your origin

MODULE 8

Running it for years

Everything so far gets a thing launched and keeps it alive through a bad week. This module is the longer horizon: what you chose to run yourself and what that costs in attention, the upgrades that arrive whether or not you want them, what a user costs you, how to move house, what the law expects of a public site, and how one person keeps this going without burning out.

> BY THE END OF THIS MODULE YOU CAN
>
> Decide what to run yourself and what to buy using a cost per user and a maintenance budget you have actually calculated, keep a system upgraded and legally presentable over several years, and move it to another provider without losing a URL

67 · 9 min

What to run yourself, and what to pay for

THE ONE THING TO KEEP

Compare money against hours of attention, not money against money: buy the stateful things whose backup, restore and upgrade obligations would eat your weekends, run the stateless ones that recover by restarting, and check how you would leave before you join.

The comparison people make is the wrong one

A managed Postgres costs perhaps $25 a month at small size. The same database on a VPS you already rent costs nothing extra. So self-hosting saves $300 a year, and people take that deal without writing down the other side of it.

The other side is a list, and it is the same list every time: installing and configuring it, taking backups, testing a restore, applying security patches, upgrading major versions, monitoring disk and connections, tuning when it slows down, and being the person who fixes it at 2am. Put an hour a month against that — which is optimistic — and at any realistic hourly value the managed service was cheaper before you finished reading this paragraph.

That is not an argument for buying everything. It is an argument for comparing the same two columns each time: **money** and **hours of your attention**, where attention is the scarcer one for a small team.

The three questions that decide it

1. **Is this thing stateful?** Stateless services are easy to run yourself: if one dies, start another. Stateful ones — databases, queues with durable state, object storage, search indexes — carry backup, restore, replication and upgrade obligations that dominate the work. **Buy state, run stateless** is the single most useful heuristic here.

2. **What happens at 3am if it breaks?** If the answer is "I do not know how to fix it", either learn before you depend on it, or pay somebody whose job that is.
3. **Would running it teach you something you need?** A genuine reason. Running Postgres yourself for a year teaches you more about databases than any course. Do it on a side project, not on the thing with users.

What is usually worth paying for, at small scale

- **The database.** Backups, point-in-time recovery, failover and version upgrades, for the price of two coffees a week. This is the best value on the list.
- **Email delivery.** The deliverability lesson explained why running your own mail server is a losing fight.
- **TLS certificates and DNS.** Free or nearly, and automated.
- **Error tracking and uptime checks.** Free tiers are generous, and self-hosting your monitoring on the machine being monitored is a known trap.
- **Object storage.** Durability you cannot easily reproduce.

What is usually fine to run yourself

- **Your own application.** Obviously.
- **A cache.** Redis or Valkey on the same box, treated as disposable, is genuinely low-maintenance because losing it is survivable by design.
- **A reverse proxy.** Caddy or nginx. One config file, rarely touched.
- **Batch jobs and workers.** Stateless, restartable, cheap.
- **Small internal tools.** Metabase, a status page, a wiki.

The middle ground people forget

Between "my own machine" and "a hyperscaler" is a large and underrated space: a €5 to €20 VPS from Hetzner, OVH, Contabo or DigitalOcean, with generous bundled bandwidth and predictable pricing. A single 4-core box with 8 GB of memory serves a surprising amount of traffic — tens of thousands of

daily visitors for an ordinary application — and costs less per month than the free tier's overage on a serverless platform.

The honest trade is that you own the operating system. Given the first module of this course, that is now within your reach, and for many projects it is both the cheapest and the most understandable option.

Read the exit before the entry

Before adopting anything managed, ask three questions:

- **Can I get my data out, in a usable format, without asking permission?** A database with a standard dump command, yes. A proprietary store with an export queue and a support ticket, be careful.
- **What is the egress cost of leaving?** Sometimes thousands of dollars, entirely by design.
- **Is there a compatible alternative?** S3-compatible storage, standard Postgres and standard Redis protocols mean the switch is configuration. A vendor-specific service means a rewrite.

Lock-in is not automatically bad — a service that saves you a hundred hours a year has earned some — but it should be entered knowingly, and written down in the decisions document from the handover lesson.

The rule of thumb

Buy the thing whose failure would ruin your weekend. Run the thing whose failure means restarting a process. Revisit once a year, because both your traffic and the prices move, and a decision that was right at fifty users is not automatically right at fifty thousand.

BEFORE YOU MOVE ON

A solo maintainer moves Postgres from a managed service to their own VPS, saving $30 a month. Which cost most reliably appears afterwards, regardless of skill?

A. Higher latency, since the database is no longer on optimised hardware

B. The recurring obligation to take backups, test restores, patch and perform major-version upgrades

C. Loss of the ability to scale, since a VPS cannot be resized

D. Higher egress charges, since traffic now leaves a different network

68 · 9 min

The upgrade you keep postponing

THE ONE THING TO KEEP

End-of-life dates are published years ahead and arrive anyway, so keep dependencies moving weekly, take one major upgrade a quarter in small steps, rehearse database upgrades because they do not roll back, and write down any version you are deliberately staying on.

Software has an expiry date, and it is published

Every language runtime, database and framework you depend on has a documented end-of-life date. After it, security fixes stop. The dates are public years in advance, and they arrive anyway, because nobody's calendar has them in it.

As a shape rather than a specific list: Node.js major versions get roughly three years, with a long-term-support line; Python versions get five; Ubuntu's long-term-support releases get five years of standard support; PostgreSQL major versions get five; Django and Rails maintain a small number of recent versions with a long-term line. The exact months change — look them up, and record the ones you depend on.

The fact that matters: an unsupported runtime does not stop working. It stops receiving patches. Your site runs exactly as well as before, which is precisely why it stays there.

The debt compounds, quietly

A framework upgrade skipped once is a week of work. Skipped four times, it is a rewrite — because the upgrade guides assume you are coming from the previous version, the migration tooling has been removed, the packages you depend on have moved on, and nobody remembers why the custom patch in the middle exists.

So the practical rule is counter-intuitive to how it feels: **upgrade more often, in smaller steps**, and never skip a major version if the project's guide says not to.

A cadence that survives contact with real work

- **Weekly, automatic:** patch and minor version updates for dependencies, applied by a bot, merged when CI is green. Dependabot and Renovate are free and do this. Group them so it is one pull request, not thirty.
- **Monthly, 30 minutes:** read the failures. A minor update that breaks a test is a signal, not an inconvenience.
- **Quarterly, half a day:** one major upgrade — the runtime, the framework, or the database — chosen from the list, done properly with a staging run and a rollback plan.
- **Annually, one hour:** update the end-of-life dates for everything you depend on and put the next two into the calendar with a name against them.

That annual hour is the whole trick. The upgrade is not hard; being surprised by it is.

Database major versions are the one to plan

Application upgrades roll back. A database major version upgrade often does not, because the data directory format changes. Three honest options:

- **A managed provider's in-place upgrade.** Simplest, and it needs a maintenance window measured in minutes, plus a tested backup taken immediately before.
- **Dump and restore.** Reliable, slow, and the downtime scales with your data size. Fine at a few gigabytes, painful at a hundred.
- **Logical replication to a new instance, then a cutover.** Near-zero downtime, more moving parts, and the approach worth learning once you are past the size where a dump is quick.

Whichever you choose, rehearse on a copy. Time it. Write the number in the runbook, because that number is your maintenance window.

Deprecations from providers arrive on their schedule

API versions get retired. Model snapshots get withdrawn. TLS versions and cipher suites get disabled by browsers. Payment providers change required flows to meet new regulation. Cloud instance families get deprecated.

Most of these come with months of notice, sent by email, to an address somebody set up in year one. Two habits: make sure provider notices reach more than one person and get read, and put every announced retirement date into the same calendar as your expiry list. This is the same discipline as the expiry lesson, applied to other people's deadlines rather than your own.

Upgrading is also how you keep a site fast and cheap

It is easy to frame upgrades purely as risk avoidance, which makes them feel like a chore with no upside. In practice, runtime and database major versions routinely bring real performance and memory improvements, deprecate a workaround you were carrying, or add something that lets you delete code. A quarterly upgrade slot is one of the few maintenance activities that regularly makes the system smaller.

When you deliberately do not upgrade

Sometimes staying is correct: a library that is finished and unmaintained but does its job, a version whose successor removed something you need, a system

nearing retirement anyway. That is a legitimate decision when it is a decision.

Write it down where the handover documents live: what you are staying on, why, what the risk is, and when you will look again. The difference between technical debt and technical negligence is whether somebody chose it on purpose and left a note.

BEFORE YOU MOVE ON

A team skips three consecutive major versions of their web framework, then attempts the upgrade in one jump. Why is this usually harder than three separate upgrades?

A. The final version contains more code, so more of the application must change

B. Upgrade guides and migration tooling target adjacent versions, and deprecation shims from intermediate releases have already been removed

C. Skipping versions invalidates the lockfile, forcing every dependency to resolve afresh

D. Security patches from the skipped versions must be applied individually first

---

69 · 9 min

What one user costs you

THE ONE THING TO KEEP

Divide the bill by active users, split it into fixed and variable, and price the one action that dominates the variable half — then test whether ten times the users would still be affordable, because the answer is nearly always one line item rather than the architecture.

One number that settles a lot of arguments

Take last month's infrastructure bill and divide it by the number of monthly active users. That is your cost per user, and it is the number that tells you

whether a free tier is generous or reckless, whether a feature is affordable, and whether your architecture has to change.

Most small projects have never computed it, and are surprised in both directions. A content site with a CDN and a small database can run at a fraction of a cent per user per month. A product where every visit triggers model calls and image processing can be tens of cents — a hundred times more, for something that looks similar from the outside.

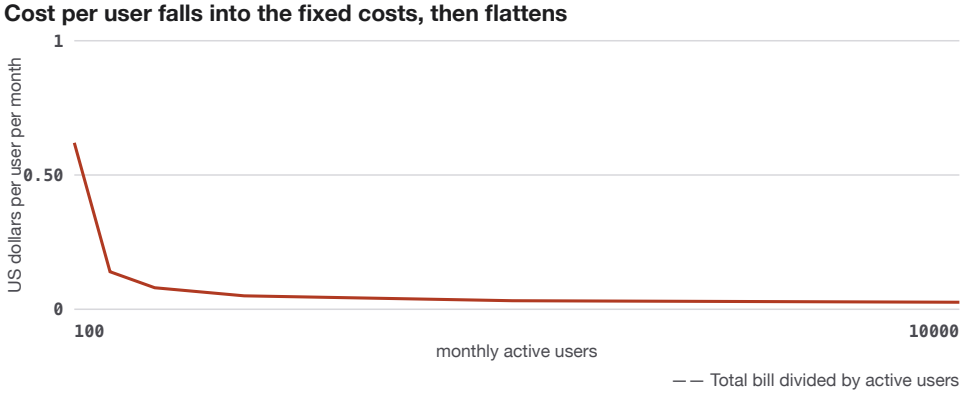
Compute it properly, once

Split the bill into fixed and variable:

- **Fixed:** the machines, the database, the monitoring, the domain. These do not move with usage. At small scale this is nearly all of your bill, which is why cost per user falls sharply as you grow — the first hundred users are extremely expensive each.
- **Variable:** bandwidth, model tokens, email, storage, per-request charges. These scale with usage and eventually dominate.

Then compute the variable cost of one *action*, not one user, because that is where the design decisions are. From the earlier lessons: a model call at $0.0008, an email at a fraction of a cent, a megabyte of egress at a tenth of a cent on a metered provider and free behind some CDNs, a stored photo at roughly two cents per gigabyte-month plus its derivatives.

Now multiply by how often an average user does each thing per month. The answer is usually dominated by one line, and that line is where all the useful work is.



A site with 60 dollars a month of fixed cost and two cents of variable cost per user. The first hundred users cost sixty-two cents each; at ten thousand the fixed half has vanished and the two-cent line is all that is left. That variable line is where the design decisions are.

The three shapes of a per-user cost problem

1. A heavy free tier. Anonymous or free users doing something expensive — generation, transcoding, search over a large index. The fix is a quota by identity tier, exactly as this project's own ladder does: more capability as somebody proves they are a person, with the expensive operations reserved for accounts.

2. Storage that only grows. Every upload stays forever, along with three derivative sizes nobody looks at. Storage is cheap per gigabyte and infinite over time. The fix is a lifecycle policy and a retention decision per class of data, which the deletion lesson already required you to have for other reasons.

3. A per-request external cost. Any external API in the request path makes your marginal cost non-zero, and abuse becomes directly expensive rather than merely annoying. Cache, quota, and put the expensive call behind an account.

When the architecture has to change

A useful test: multiply your current cost per user by ten times the users. If the answer is fine, your architecture is fine and you should not redesign anything. If the answer is alarming, find the one line item responsible — it will be one — and fix that, not the architecture.

Almost every "we need to rearchitect for scale" conversation in a small product is actually one uncached endpoint, one missing index, one image not resized, or one model call that could be a cheaper model. Rewrites are the most expensive way to reduce a bill.

Free forever, honestly

Some projects — this one included — commit to being free permanently. That is a real commitment with a real arithmetic consequence: cost per user must stay low enough that the funding source, whatever it is, can carry the whole user base. It is achievable, and it constrains design in specific ways:

- Reading must be cheap. Static, cached, served from an edge.
- Expensive operations need a per-person daily budget, enforced in code before the call, not a hope that nobody abuses it.
- Storage growth must have a policy, or it becomes a permanent rising cost with no ceiling.
- Anything with a per-token or per-request price needs a hard monthly cap at the provider, so a bad week cannot become an unpayable invoice.

Saying "free" and meaning it is an engineering constraint, not just a promise on a page.

Track it monthly, in one row

A table with five columns — month, total bill, active users, cost per user, and the largest line item — takes five minutes to fill in and makes the trend visible. A cost per user that is falling means you are growing into your fixed costs. One that is rising means a variable line has started to dominate, and you have found it early enough to do something cheap about it.

BEFORE YOU MOVE ON

A project's monthly bill is 90% fixed. Users double and the bill rises by 4%. What does this indicate about the next growth decision?

A. The architecture is nearly free to scale and no cost work is needed at any size

B. Growth is currently cheap because fixed costs dominate, so the useful question is which variable line grows fastest as that changes

C. The bill is mostly waste, since costs did not track usage

D. Cost per user is meaningless while fixed costs dominate

70 · 9 min

Moving to another provider without losing a URL

THE ONE THING TO KEEP

Migrate in parallel and in order — lower the DNS TTL days ahead, replicate the data continuously, cut over at the quiet hour with the old system still alive, and serve 301s for every old URL — because the one cost you cannot recover is a link that stops working.

Everything moves eventually

Prices change, a free tier ends, a provider is acquired, a region closes, or the thing you chose at the start no longer fits. Migration is not a failure; it is a normal event in the life of a system that survives.

It is also where small projects lose data and break links, because the work is done in the wrong order.

The order that works

1. **Stand the new thing up in parallel.** Nothing is switched. Deploy the application to the new provider, point it at a copy of the data, and use it yourself on a temporary hostname.
2. **Copy the data, then keep it in sync.** For a database, a dump and restore establishes the baseline; logical replication or change-data-capture keeps it current so the final cutover moves minutes of data rather than hours.
3. **Lower the DNS TTL first.** Days in advance. A record with a 24-hour TTL takes up to a day for the last resolver to notice a change; a 60-second TTL makes the cutover reversible in a minute. This step must happen before everything else, and it is the one most often forgotten.
4. **Run read traffic against the new system** while writes still go to the old one, if your architecture allows it. Any difference in behaviour appears here, without risk.
5. **Cut over during your quietest hour**, with the old system still running and still receiving data.
6. **Watch for a day, then a week**, before decommissioning anything.
7. **Keep the old system's data**, offline, for a month after you turn it off.

The stateless half is easy; the state is the migration

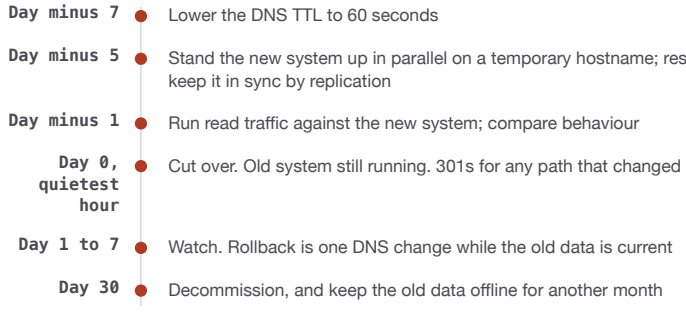
Moving an application is a deploy to a different place. Moving a database is the actual project, and there are three approaches:

- **Downtime window.** Stop writes, dump, restore, start. Simple, honest, and fine if you can afford ten minutes at 4am and your data is small. Rehearse it and time it first.
- **Logical replication.** The new database subscribes to the old one's changes and stays current; you cut over when the lag is near zero. Standard in Postgres and MySQL, and the right answer above a few tens of gigabytes.
- **Dual write.** The application writes to both, reads from one. Powerful, and the most error-prone of the three, because partial failures produce di-

vergence. If you use it, add a comparison job that reads from both and reports differences, or you will find the divergence months later.

File storage has a fourth trick worth knowing: copy everything in advance, then keep serving reads from the old bucket while new writes go to the new one, and run a background copy for the remainder. S3-compatible tooling — `rclone` is free and excellent — makes this a single long-running command.

A migration with an undo at every step



The order is the whole method. Lower the TTL before anything else, cut over with the old system still alive and still receiving data, and decommission nothing for a month, because a same-day decommission is a migration with no undo.

The costs that appear at the door

Egress is charged on the way out. Moving 5 TB off a provider that charges 9 cents a gigabyte is about $450, payable before you have benefited from the move. Check that number early; occasionally it changes the plan, and occasionally the destination provider will cover it, which is worth asking.

There are also regulatory rules in some places designed to reduce switching costs for cloud customers, and the picture varies by jurisdiction. Do not rely on it; budget for the bill.

URLs are the thing you must not break

Every link somebody saved, every search result, every citation of your content is a URL. Losing them is the one migration cost you never recover, because the traffic simply stops and nobody tells you.

- Keep the same paths if you possibly can.

- Where paths must change, serve **301 redirects** from every old URL to its new one, and keep them for years, not months.
- Check redirects with a crawler before and after — free tools will fetch your sitemap and report every non-200.
- Do not move the domain and the platform at the same time. One change at a time, a week apart, so you know which one caused the problem.

Write the rollback down first

Before the cutover, one page: what "go back" means at each step, who decides, and what the deadline is. In practice it is usually "change the DNS record back and stop writes to the new system" — which only works if the old system is still running and still has current data, which is why steps 5 to 7 above are in that order.

A migration where the old system was decommissioned on the same day is a migration with no undo, and it is the reason most horror stories about this exist.

BEFORE YOU MOVE ON

A team plans a provider migration for Saturday. On Friday they are still deciding details, and the DNS record has a 24-hour TTL. What does this imply for Saturday's cutover?

A. Nothing, provided they update the record before starting the migration

B. Some resolvers will keep sending users to the old system for up to a day, and reverting is equally slow

C. The cutover will fail, since DNS changes cannot take effect while a long TTL is set

D. They should switch to a different DNS provider so the change takes effect immediately

71 · 8 min

The hour a week that keeps it alive

THE ONE THING TO KEEP

Everything that keeps a system healthy is recurring and forgettable, so put it in a calendar with intervals, one owner per item and a written note per run — and when you fall behind, restore a backup and review access before anything else.

Maintenance is not a mood

Everything in this course that keeps a system healthy is recurring: restores get tested, keys get rotated, dependencies get updated, access gets reviewed, costs get checked, capacity gets re-measured. None of it is hard. All of it is forgettable.

The teams that stay out of trouble are not more disciplined by temperament. They have a written calendar with a named owner per item, and the items are small enough to actually do. The expiry lesson gave you the list of things with dates on them; this is the list of things with *intervals*.

The calendar

Weekly — 20 minutes

- Merge the grouped dependency update pull request, once CI is green.
- Skim errors from the last seven days: any new group, any big jump.
- Glance at the cost dashboard for anything unexpected.

Monthly — one hour

- Perform a deploy and a rollback, deliberately, and time the rollback.
- Read the alerts that fired: which were actionable, which were noise, delete or retune the noise.

- Check backup jobs actually ran, and check the size of the most recent backup is plausible.
- Review the open security findings from your scanner and either fix or explicitly accept, with a date.

Quarterly — half a day

- Restore a backup into a scratch environment and record the time it took.
- Rotate one credential, end to end, using its runbook.
- Review who has admin, database and cloud access; remove what is no longer needed.
- Take one major upgrade from the list.
- Re-run the load test and compare the ceiling with last quarter's.
- Read your own runbook as though you had never seen it, and fix what it assumes.

Annually — one hour

- Refresh the expiry list: domains, certificates, cards, tokens, contracts, end-of-life dates.
- Recompute cost per user and the ten-times-users test.
- Re-read the decisions document and mark anything that is no longer true.

Make it a real appointment

Three things that decide whether this survives its first busy month:

1. **It is in the calendar with a time**, not on a list. An hour on Friday morning, recurring.
2. **Each item has one name against it.** Shared ownership on a small team means nobody.
3. **The output is written down.** A one-line note per item in a file: date, what was done, what was found. Six months of those notes is how you notice that the restore has been getting slower.

Automate the parts that are honest to automate

Much of the weekly list should be a bot. Dependency pull requests, certificate expiry checks, backup success reports, cost anomaly alerts, a scheduled job that fails the build when something is expiring within 30 days. Free tooling covers all of it.

What should *not* be automated is anything whose value comes from a person looking: the alert review, the access review, the runbook read-through. A green tick from a script that nobody reads is the same as no check at all — this course has made that point about health checks, about verification scripts and about backups, and it is the same failure each time.

When you are behind

Everybody falls behind. The recovery is not to do six months of maintenance in one weekend; it is to pick the two items with the worst consequences and do those.

For almost every project, those two are: **restore a backup**, and **review who has access**. One protects the data, the other protects it from people who should no longer reach it. Everything else can wait a fortnight.

The measure that tells you it is working

Count, over a year, how many of your incidents were caused by something on this list going undone — an expired certificate, an unpatched dependency, a full disk, a backup that had been failing silently. In an unmaintained system that is most of them. In a maintained one it approaches zero, and the incidents you do have are genuinely novel.

That is the return on an hour a week: not the absence of problems, but the absence of the boring ones you already knew how to prevent.

BEFORE YOU MOVE ON

A team automates backup verification with a script that emails a success message nightly. A year later a restore fails because the backups contained only one of two databases. What does this illustrate?

A. Backups should be taken more frequently than nightly

B. The email was not being read, which is the real failure

C. A check that verifies the job ran is not a check that a restore works, so the quarterly restore drill is the one that would have caught it

D. Automation is unreliable and this work should be done manually

72 · 8 min

Removing a feature, and retiring a site

THE ONE THING TO KEEP

Removal is a shipping skill with a sequence: count real usage, announce with a date, offer an export, dark-launch the switch-off, remove code before data, and answer old URLs with a 301 or an honest 410 rather than a redirect to the homepage.

Deleting is a shipping skill

Everything you keep has a cost: code to maintain, a database table to migrate, a page to test, a dependency to upgrade, a surface to secure. Features that nobody uses charge you rent forever, and the only way to stop paying is to remove them.

Most people find this harder than building. The reason is usually a fear of the small number of users who will notice — which is a real consideration, and it is best handled with a process rather than with avoidance.

Find out who actually uses it

Before any argument about whether to remove something, count. Add one log line or one counter to the feature and leave it for a month.

The result is often decisive: a feature you assumed had a small loyal following has eleven users, four of whom are your own test accounts. Occasionally it goes the other way and the thing you thought was dead is quietly load-bearing for a group nobody spoke to. Either way you are now arguing about a number.

The removal sequence

1. **Announce**, with a date. Where the users are — in the interface, not only in an email nobody reads.
2. **Give an export**, if the feature holds anybody's data. This is the part that makes removal legitimate rather than merely convenient.
3. **Stop new use.** Hide the entry point, keep it working for existing users. Numbers fall immediately and you learn who the real users are, because they complain.
4. **Dark-launch the removal.** Turn it off behind a flag for an hour, then a day, and watch what breaks. This is exactly the technique from the kill-switch lesson, used deliberately.
5. **Remove the code and the routes**, but leave the data.
6. **Delete the data** after your retention period, not on the same day.

That gap between step 5 and step 6 is where recoverable mistakes live. Code you can restore from git; data you cannot.

Keep the URLs alive

When a page goes away, the links to it do not. A removed page should return a **301** to its nearest replacement where one exists, or a **410 Gone** where it genuinely does not — 410 tells search engines and other machines that this is deliberate and permanent, which is more honest than a 404 and gets the page removed from indexes faster.

What should not happen is a redirect from every removed page to the home-page. It looks tidy and it is a small lie: the person clicked a specific link and got a shrug. A short page saying what was there, when it went, and where the nearest equivalent is costs an hour and is remembered kindly.

Retiring the whole thing

Projects end. Doing it well takes about a day and preserves both your users' work and your own reputation.

- **Say it early**, with a real date, and say why in one honest paragraph.
- **Give everybody a full export** in a format that opens elsewhere: JSON, CSV, Markdown, standard files. Not a proprietary archive that only your dead product can read.
- **Keep the domain** for as long as you can afford. Expired domains are bought and repointed, and a site that was yours becomes somebody else's advertising with your reputation attached to it.
- **Consider a static archive.** Most database-backed sites can be crawled into flat HTML with `wget --mirror` or a static-site crawler and hosted for nothing on any static host. The content stays readable, the links keep working, and the running cost goes to zero. This is by far the best outcome for anything with public writing in it.
- **Turn off the money.** Cancel the subscriptions, remove the card, revoke the keys, and delete the cloud resources — in that order, after the archive is confirmed working.
- **Delete the personal data** you no longer have a reason to hold, per the retention lesson. A dead product still has data obligations until the data is gone.

Say what happened

A final post explaining what the project was, what it did, what it cost, and why it stopped is worth writing. It closes the thing for the people who used it, and it is the most useful document you will ever produce for the next person considering the same idea — including, quite often, yourself.

BEFORE YOU MOVE ON

A team removes a section of their site and redirects every old URL to the homepage, reasoning that visitors will find what they need. What is the concrete downside?

A. Search engines treat mass redirects to an unrelated page much like a not-found, so the pages leave the index anyway while visitors lose the context they clicked for

B. 301 redirects pass no signal at all, so the pages should have returned 404

C. Browsers cache 301s permanently, so users can never reach the homepage again

D. The redirects will overload the server, since every old URL now hits one page

73 · 9 min

The pages every public site needs

THE ONE THING TO KEEP

A public site needs terms, a truthful privacy policy written from your own schema, a reachable contact, visible rules and an honest tracking position — and the cleanest answer to consent banners is to set nothing that needs consent.

The part engineers skip

A site that accepts users is a small institution, and institutions have paperwork. Skipping it is common and it is a bet: that nobody asks, that nothing goes wrong, and that no obligation applies to you. That bet is fine at zero users and worse every month.

This lesson is a map of what exists. It is emphatically **not legal advice** — the rules differ by where you are, where your users are, what data you hold and what you charge for, and several of the questions below are actively unsettled

in courts and parliaments right now. What it will do is stop you being surprised.

The five documents

1. Terms of service. The agreement between you and the user. What the service is, what people may not do, what happens when they break the rules, who owns what they post, that you may change or end the service, and where disputes are handled. Its main practical function on a small site is to give you a stated basis for suspending an abusive account.

2. Privacy policy. What personal data you collect, why, on what basis, who you share it with, how long you keep it, and how somebody exercises their rights. In many jurisdictions this is not optional if you have users there. It must describe what your system actually does — a policy copied from another site that names services you do not use is worse than none, because it is a written statement that is false.

3. Contact and identity. Some jurisdictions require a public business identity and contact address. Even where they do not, a reachable address is the difference between a complaint and a legal notice.

4. Community rules, if users can post. Short, specific, and written in the language of your users rather than of lawyers. This is the document moderators actually apply, and the one users are entitled to see before they are judged by it.

5. A cookie or tracking notice, if you set anything beyond what is strictly necessary.

Cookie banners, honestly

The rule in the European regime is roughly: cookies strictly necessary for the service need no consent; anything for analytics or advertising needs freely given, informed, opt-in consent, and refusing must be as easy as accepting. Most banners on the web do not meet that standard, and regulators have said so repeatedly.

There is a cleaner answer available to a small site: **do not set any non-essential cookies**, and you need no banner at all. Privacy-respecting analytics — Plausible, Umami, GoatCounter, or your own server logs — measure page views without per-visitor identifiers, and several are free to self-host. This is the rare compliance question with a genuinely better technical answer than the legal one.

Where it is unsettled, and stays unsettled

Be honest about the live disputes rather than pretending they are resolved:

- **Training data and copyright.** Whether using copyrighted material to train a model is permitted, and under what conditions, is being litigated in several countries with different answers emerging. Anybody who tells you this is settled is describing one jurisdiction's current position as though it were universal.
- **Ownership of model output.** Whether machine-generated work attracts copyright, and to whom, varies — some offices require meaningful human authorship, and the boundary is being drawn case by case.
- **Cross-border data transfer.** The legal mechanisms for moving personal data between regions have been struck down and rebuilt more than once. Anything you build on top of one should be able to survive the next change.
- **Platform liability for user content.** Different regimes, actively being amended, with different duties by platform size.

The engineering response to all four is the same: keep records of what you collected and from where, keep the ability to delete or move it, and do not build a product whose viability depends on one contested legal question going your way.

Accessibility is also an obligation

In a growing number of jurisdictions, public digital services must meet an accessibility standard, and the direction of travel is towards more sectors rather

than fewer. Separately from any law, it is the difference between your site being usable by somebody with a screen reader or not.

The basics are not expensive: semantic HTML, real labels on form fields, keyboard operability, sufficient colour contrast, alt text, and not conveying meaning by colour alone. Free checkers — axe, Lighthouse, WAVE — find most of the mechanical failures in one pass.

What to do this week

1. Write the privacy policy from your own schema: open the tables, list the personal data, and describe it truthfully.
2. Publish a contact address that a human reads.
3. Remove any tracking you are not actually using — most sites are carrying a script somebody added in year one and nobody reads.
4. Run one accessibility check and fix the automated findings.
5. Note the two or three legal questions that genuinely apply to your situation, and get advice on those rather than reading forums about all of them.

BEFORE YOU MOVE ON

A team copies a privacy policy from a larger company. It mentions advertising partners and analytics providers they do not use, and omits the model provider they send user text to. Why is this worse than having no policy at all in most regimes?

A. Copying the text infringes the other company's copyright

B. It is a published statement about their data handling that is inaccurate in both directions, and the omitted processor is the one users would care about

C. Policies must be written by a lawyer to have any effect

D. Naming providers you do not use exposes you to their terms of service

74 · 9 min

Keeping it going without burning out

THE ONE THING TO KEEP

Attention is the scarce resource in a small project, so decide in advance what is worth waking for, automate the frequent and write runbooks for the rare, refuse features you cannot maintain, and judge the system by how long it runs without you and how quickly you can understand it when you return.

The constraint nobody puts in the architecture diagram

Everything in this course assumes somebody will do it. On a small project that somebody is one or two people who also write the features, answer the emails and have a life. Attention, not money or compute, is the scarce resource, and a system that consumes more of it than you have will decay regardless of how well it was built.

So the last lesson is about designing the operation around a real person's capacity, which is a legitimate engineering constraint and not a personal failing.

Decide what "down" costs before you promise anything

A one-person project cannot offer 24-hour response and should not pretend to. The honest version is to decide, in advance, three tiers:

- **Wake me:** the site is down for everybody, data is at risk, money is being lost, or a security incident is live. This should be a very short list, and every item should have a runbook.
- **Same day:** a feature is broken, a subset of users is affected, an alert that means something is degrading.
- **This week:** everything else, including most of what currently pages people.

Write it in the runbook, configure the alerts to match, and publish your actual support hours. A stated 24-hour response time that you meet is worth more than an implied instant one that you do not.

Automate the recurring, accept the rare

A useful test for any piece of operational work: how often does it happen, and how long does it take?

- **Frequent and short:** automate it. Dependency updates, certificate renewal, backup verification, deploys.
- **Frequent and long:** this is where you are losing your life. Fix the underlying cause, or the automation, or the feature.
- **Rare and short:** do it by hand, from a runbook.
- **Rare and long:** write the runbook and accept it. Building automation for something that happens once a year takes longer than the task and will be broken by the time you need it.

That last quadrant is where a lot of over-engineering lives, usually justified as saving time.

Reduce the surface deliberately

Every feature, integration, dependency and environment is a standing claim on your attention. The maintenance calendar's length is a direct function of how many things exist.

So saying no is an operational decision, not a personality trait. Fewer features, fewer services, one language where possible, one database, boring dependencies with many users and slow release cycles, and managed services for anything stateful. Each of those choices buys back hours per month, and the previous lessons in this module are all versions of the same idea.

The strongest version: if you cannot maintain it, you cannot ship it. A feature that will need weekly attention forever is not free because the code is written.

Build for the version of you who is tired

The person who fixes the outage is you at 2am, on a phone, on holiday, six months after you last touched this code. Everything that helps them is worth more than it looks:

- A runbook with the actual commands, copy-pasteable.
- A rollback that is one command and needs no thought.
- Alerts that link to the runbook.
- A `/version` route so you know what is live.
- A kill switch per risky feature.
- Notes in the decisions file explaining why something strange is the way it is.

None of that is glamorous. All of it converts a bad night into twenty minutes.

Take real breaks, and design for them

The things that make time off possible are specific and buildable: a system that survives a week unattended, spend caps so a runaway cannot become an unpayable bill, an auto-reply with realistic expectations, and one other person who has access and knows where the runbook is — even if their only job is to turn the site off if it starts doing harm.

If nobody else can act, write down the one command that stops everything and make sure a trusted person has it. That is not a governance nicety; it is what stands between a bug and a week of damage while you are unreachable.

The measure of a well-run small system

Not uptime. Not the number of features. It is this: **how long can it run without you, and how quickly can you understand it when you come back.**

A system that runs itself for a fortnight, tells you the truth when something is wrong, and can be understood from four short documents is a well-engineered

one, whatever it is built on. That is what everything in this course has been for — and it is what you can now build, deploy, watch, defend and hand over.

BEFORE YOU MOVE ON

A solo maintainer spends four hours a month manually reconciling a data import that fails in a predictable way. Which quadrant is this, and what follows?

A. Rare and long — write a runbook and accept the cost

B. Frequent and long — the underlying failure should be fixed, since this is where a maintainer's capacity is actually consumed

C. Frequent and short — automate the reconciliation script

D. Rare and short — no action needed beyond documentation

CASE STUDY · MODULE 8

The volunteer left, and the free tier ended

A children's library trust in Kochi with a lending site nobody has deployed for eleven months, and thirty days of notice.

The trust runs a lending library for about nine hundred active borrowers, with three thousand one hundred borrower records and nine thousand titles. The site was built in 2023 by a volunteer, Anoop, who did it well and then took a job in Bengaluru in January. Nobody has deployed anything since.

In November two emails arrived within a week. The managed database provider was discontinuing the free tier the site runs on, in thirty days. And the language runtime the application uses reaches end of life in February, after which it stops receiving security patches — a date that had been published three years in advance.

The trust's position when it looked at what it had:

- The repository exists and is complete. The README has four steps. Three of them fail on a machine that is not Anoop's laptop.
- Nobody at the trust knows how to deploy it. There is a deploy script; it references an environment variable nobody can find the value of.
- The domain renews on a card belonging to Anoop's father. Auto-renew is on. Nobody at the trust has access to the registrar account.
- There is a nightly database backup. Nobody has ever restored one.
- The librarian uses the lending screen for every issue and return, about seventy a day.

The trustees had thirty days and three options, each with a cost they could name.

Pay for the managed database and buy Anoop's time. Roughly two thousand rupees a month for the database, plus a weekend or two of a former volunteer's paid time for the runtime upgrade and a handover. It is money the trust does not have spare, and it puts them back where they started — dependent on one person who lives elsewhere.

Move everything onto a small virtual machine themselves. About five hundred rupees a month, a quarter of the price. It also means somebody at the trust owns an operating system: patches, firewall, backups, restores, a database to upgrade, and a person to be woken. There is no such person. The librarian is a librarian.

Retire the lending half and keep the catalogue as a static archive. Crawl the catalogue into flat HTML, host it free, serve 301s from the old catalogue URLs, export the lending records to CSV, and go back to the paper register the library used before 2023. The running cost falls to almost nothing and the links keep working. It also removes the one screen the librarian uses seventy times a day.

Before choosing, one trustee asked the question that should have been asked in 2023 and had not been: if we take the paid database plan, how do we leave it later. The answer was reassuring and it was not luck — the database is ordinary Postgres, the dump command is standard, and there is no export queue or support ticket between the trust and its own data. Had the volunteer chosen a proprietary store with a vendor-specific query language, the same thirty days would have been a rewrite rather than a bill.

The other quiet fact in the room was that the lending screen is the only part anybody uses daily, and it is also the part that holds state. The catalogue is a read-only view of a table and could live anywhere for nothing. Almost the whole of the trust's operating burden came from one feature, which is a useful thing to know before deciding what to keep.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

The trustees bought two weekends of Anoop's time, which turned out to be the cheapest of the three once they compared money against hours of attention

rather than money against money. Buy the stateful thing whose backup, restore and upgrade obligations would eat somebody's weekends; run the stateless things that recover by restarting.

The first weekend was infrastructure. The runtime upgrade was taken in two steps rather than one, with a staging run each time. The database moved to a paid plan on the same provider, with a dump and restore rehearsed on a scratch instance first and timed at twenty-six minutes — a number now written in the runbook, because that number is the trust's real recovery time. The domain moved into an account owned by a role address at the trust, with auto-renew on a trust card and a transfer lock enabled. A restore was performed into an empty database and a page opened against it.

The second weekend was documents. A README that a trustee followed on a borrowed laptop while Anoop watched and did not help: it took three attempts and eleven corrections, and every correction was a step he does without noticing. A RUNBOOK with the deploy command, the rollback command, the restore procedure and the date it was last actually tested. A RENEWALS file listing the domain, the card, the database plan, the certificate and both end-of-life dates. And a DECISIONS file with six entries explaining why things are as they are.

They also computed cost per user for the first time: about two thousand five hundred rupees a month across nine hundred active borrowers, roughly two rupees eighty each, almost all of it fixed. Ten times the borrowers came out at about forty paise each, which ended a two-year argument about whether the system would cope with growth.

The librarian now has an hour in the calendar on Friday morning. Eleven months later the site had had one incident: a certificate that failed to renew because a firewall change had closed port 80, caught by the expiry check at nineteen days remaining.

The virtual machine was a quarter of the monthly price. Why was paying more the cheaper decision?

Because the comparison is money against hours of attention, and attention was the resource the trust did not have. The saving of fifteen hundred rupees a month buys

an obligation to patch, back up, restore, monitor and upgrade a database, with nobody able to do any of it at three in the morning. Buy the stateful things whose failure would ruin a weekend; run the stateless things whose failure means restarting a process.

What made the README test honest, and why can the author not perform it alone?

Somebody else followed it on a clean machine while the author watched and refused to help. The author cannot find the gaps because the missing steps are precisely the ones he performs without noticing — a global install from years ago, an environment variable exported in a shell profile, a directory that already exists. Every question the other person has to ask is a defect in the document, and the count of questions is the measure.

What did the ten-times-users test settle, and why is that the right question to ask?

It settled whether the design needed to change, and the answer was no: at nine thousand borrowers the cost per user falls, because nearly the whole bill is fixed. The test is right because it separates a cost problem from an architecture problem. When the answer is alarming it is almost always one line item — an uncached endpoint, a missing index, an unresized image, an expensive model call — and fixing that line is far cheaper than the rewrite people reach for instead.

MODULE 8 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 You are deciding what to self-host on a small team. Which heuristic does most of the work?

A. Run anything with a free tier; buy anything metered per request
B. Buy the stateful things; run the ones that recover by restarting
C. Run whatever you understand; buy whatever you have not used before
D. Buy anything on the critical path; run everything users do not see

2 Why does a database major-version upgrade need rehearsing in a way an application upgrade does not?

A. It requires a maintenance window that has to be announced to users
B. The client libraries change, so application code must be updated too
C. It does not roll back, so the only unknown left is how long it takes
D. Backups taken before the upgrade cannot be restored into the new version

3 Your cost per user is fine today. You multiply it by ten times the users and the answer is alarming. What is the usual correct response?

A. Redesign the architecture before the growth arrives
B. Introduce a paid tier to cover the variable half of the bill
C. Recompute with the fixed costs excluded, since they do not scale
D. Find the single line item responsible and fix that

4 You are moving to another provider next weekend. Which step has to happen days in advance rather than on the day?

A. Lowering the DNS TTL so the cutover can be undone in a minute
B. Taking the final database dump and restoring it on the new host
C. Setting up 301 redirects from every old URL to its replacement
D. Decommissioning the old machines to avoid paying for both

5 A feature is removed and its pages have no replacement. What should those URLs return?

A. A 302 to the homepage, so nobody meets an error page
B. A 404, which is what the server would return anyway
C. A 410, saying the page is deliberately and permanently gone
D. A 301 to the nearest section, so link value is preserved

6 A maintenance task happens roughly once a year and takes half a day. What should you do with it?

A. Automate it, since half a day is the most expensive kind of task
B. Write the runbook and accept doing it by hand
C. Delete the requirement, since annual work is rarely load-bearing
D. Split it into monthly pieces so it becomes routine and automatable

EXAMINATION

Shipping It — final examination

This paper covers the whole course: how a request reaches your process, making a deploy boring and reversible, knowing what your system is doing, load and cost and the traffic that is not customers, staying alive on the bad day and the day after, shipping a model rather than a web app, security past the short list, and running the thing for years.

Twenty-five questions are drawn at random from a larger pool, so no two papers are the same. The pass mark is 70 per cent, and passing opens the next course in the track.

There is no timer. Read each question as slowly as you like, in whatever language you think in, and come back to it later if you want to. Every question asks about a mechanism the course explained rather than a definition to recall, and after you submit you are shown why the right answer is right — including for the questions you got right.

If you do not pass, nothing is locked. You may retake after thirty minutes, and the retake draws fresh questions from the pool rather than showing you the same paper again. A teacher can also open the next course for you at any time, which is recorded as a teacher's decision and is never presented as a pass.

On the website a sitting is 25 questions drawn from this pool and the pass mark is 70%. There is no time limit, there and here. Passing opens the next course in the track.

1 `1. How the internet reaches your process`

You move an always-on application to a per-request serverless platform. Which of your existing habits silently stops working?

A. Reading configuration from environment variables at start-up

B. A nightly job scheduled with setInterval inside the process

C. Writing one structured log line per request to stdout

D. Returning a JSON body with an explicit status code

2 1. How the internet reaches your process

On a new machine, `ss -tlnp` shows Postgres listening on `0.0.0.0:5432`. What have you just learned?

A. Postgres is reachable from the whole internet unless something else stops it
B. Postgres is bound to the loopback interface and is safe from outside
C. The port is open locally but blocked by the default firewall policy
D. Postgres is running as root and should be moved to its own user

3 1. How the internet reaches your process

Your process keeps vanishing. There is nothing in its own logs before each disappearance — no exception, no shutdown line. Where do you look?

A. The supervisor's restart-limit settings, which suppress the final error
B. The proxy's error log, which records upstream connection failures
C. The kernel log, for an out-of-memory kill
D. The deploy history, for a release that overlapped the crash

4 1. How the internet reaches your process

Why do hosting instructions always say to use a CNAME for `www` and an A record for the bare domain?

A. A CNAME cannot point at an address outside your own zone
B. The specification forbids a CNAME beside other records at the apex
C. Apex records are answered by the registrar rather than the nameservers
D. CNAME lookups add a second round trip that the apex cannot afford

5 1. How the internet reaches your process

Why must a destructive action never sit behind a plain link such as `/posts/12/delete`?

A. Browsers cache GET responses, so the deletion appears not to happen
B. Search engines penalise sites whose links change server state
C. The URL is visible in logs, which leaks the identifier being deleted
D. Prefetchers, crawlers and link scanners will follow it with no human involved

6 1. How the internet reaches your process

You ship `Strict-Transport-Security: max-age=31536000; includeSubDomains` and one of your subdomains is HTTP-only. What is the position?

A. That subdomain is unreachable, and you cannot reach the browsers to undo it
B. That subdomain is unreachable until you remove the header from responses
C. Only browsers that visit the subdomain directly are affected by the policy
D. Nothing breaks, because the policy applies to the exact host that sent it

7 1. How the internet reaches your process

Your service creates a fresh HTTP client for every outbound call to a payment API. What does that cost under load?

A. Extra memory, because each client allocates its own buffers
B. Nothing measurable, since connection setup happens in the background
C. A TCP and TLS handshake per call, and sockets piling up in TIME_WAIT
D. Rate limiting at the provider, which counts connections rather than requests

8 2. Making a deploy boring

A build tool says an API key is undefined in the browser. Somebody adds the `NEXT_PUBLIC_` prefix and the error goes away. What has happened?

A. The key is now read at runtime rather than at build time
B. The key has been substituted into a bundle every visitor downloads
C. The key is now encrypted by the platform before it reaches the browser
D. The key is only exposed in development builds, not in production ones

9 2. Making a deploy boring

You build a container image on an Apple Silicon laptop, push it, and the server reports `exec format error`. What happened?

A. The image layers were corrupted in transit and must be pushed again
B. The base image tag was updated between your build and the deploy
C. The entrypoint was written in shell form and the shell is missing
D. You built for arm64 and the server runs x86

10 2. Making a deploy boring

An outside contributor opens a pull request and your pipeline's deploy step fails on a missing secret. What is going on?

A. The fork's workflow file is out of date and must be rebased first
B. Secrets are scoped to branches, and the fork's branch is not covered
C. Pull requests from forks are deliberately denied your secrets
D. The runner masks secrets in forked builds, which breaks the value

11 2. Making a deploy boring

Two people run `apply` against the same infrastructure at the same moment. What is the danger, and what prevents it?

A. Duplicate resources; prevented by naming resources with unique suffixes
B. A rate limit at the provider; prevented by retrying with backoff
C. Drift from the console; prevented by running plan on a schedule
D. A corrupted state file; prevented by a remote backend with locking

12 2. Making a deploy boring

Which rule most reliably prevents the classic staging incident, where forty thousand customers receive a test email?

A. Non-production may hold no production credentials for anything that writes
B. Staging data must be anonymised before any test run is started
C. Test accounts must use addresses on a domain you control
D. Every outbound message must be reviewed before a test is run

13 2. Making a deploy boring

Why backfill ten million rows in batches of a few thousand rather than in one `UPDATE`?

A. Batches let you run the migration on a replica and promote it afterwards
B. One statement holds a transaction open for minutes, blocks vacuum and bloats the table
C. One statement exceeds the query size limit that most drivers impose
D. Batching allows the planner to choose an index scan instead of a sequential one

14 2. Making a deploy boring

You roll the code back and the errors continue. Which of these explains it?

A. The load balancer is still routing to the new instances
B. The rollback restored the artifact but not the lockfile
C. The cache is full of objects serialised in the new shape
D. The previous artifact was rebuilt rather than reused

15 3. Knowing what it is doing

Your load balancer reports 502s for a ten-minute window. Your application logs show nothing at all for that window — no errors and no successful requests either. What does that mean?

A. The logging agent stopped shipping, so the evidence is elsewhere
B. The requests never reached your code at all
C. The errors were logged at debug level, which is off in production
D. The balancer timed out before your handlers could finish

16 3. Knowing what it is doing

Your log bill has tripled. Which control cuts it most without losing your ability to investigate?

A. Shorten every message and remove fields you rarely read
B. Move from JSON back to plain text lines, which compress better
C. Sample the successful noise and keep every error and warning
D. Reduce retention from fourteen days to three across the board

17 3. Knowing what it is doing

A page makes twenty backend calls and each has a one per cent chance of being slow. What share of page loads contains at least one slow call?

A. About one per cent, since the calls are independent

B. About five per cent, once retries are included

C. About eighteen per cent

D. About twenty per cent, one per cent for each call

18 3. Knowing what it is doing

Why is a stacked area chart the wrong shape for p50, p95 and p99 on one panel?

A. Stacking hides the threshold line that makes the panel readable

B. The three series update at different intervals and cannot be aligned

C. Percentiles do not add up, and stacking claims that they do

D. Areas obscure the gaps in data when a scrape is missed

19 3. Knowing what it is doing

Why is catching an unhandled exception and keeping the process running usually worse than letting it crash?

A. The exception is lost, so your error tracker never groups it

B. A supervisor restarts a crashed process; nothing restarts a wedged one

C. The stack trace becomes useless once the frame has been unwound

D. Restarting clears the memory leak that produced the exception

20 3. Knowing what it is doing

Why must at least one health check run on infrastructure you do not control?

A. External checks measure latency as a user experiences it

B. Providers require an independent check before they will honour credits

C. A check inside the failure goes silent, and silence is not an alert

D. Internal checks cannot see DNS or certificate problems at all

21 3. Knowing what it is doing

A request spends eight hundred milliseconds apparently in the database, but the database itself is idle and every query is fast. What is most likely?

A. The request was waiting to acquire a connection from the pool
B. The query planner chose a sequential scan over an index
C. The result set was large and serialisation dominated the time
D. Replication lag delayed the read until the replica caught up

22 4. Load, cost and the traffic that is not customers

A function writes a file to a bucket, and writes to that bucket trigger the function. What have you built?

A. A durable pipeline, since each write is processed exactly once
B. A loop that bills per invocation until something stops it
C. A queue with unbounded depth but a fixed monthly cost
D. A race between the function and the bucket's consistency model

23 4. Load, cost and the traffic that is not customers

Which `Cache-Control` value means "you may store this, but check with me before reusing it"?

A. no-store
B. no-cache
C. private
D. max-age=0, immutable

24 4. Load, cost and the traffic that is not customers

Your site is fully dynamic and nothing can be cached. Why put a CDN in front anyway?

A. It compresses responses that your origin would otherwise send uncompressed
B. It absorbs bot traffic before it reaches your rate limiter
C. It rewrites your headers so responses become cacheable automatically
D. Handshakes complete nearby, and the edge reuses a warm connection to your origin

25 4. Load, cost and the traffic that is not customers

You run a load test on a small application. Which resource is most likely to give out first?

A. Database connections
B. CPU on the application instances
C. Network bandwidth at the origin
D. The operating system's file descriptor limit

26 4. Load, cost and the traffic that is not customers

Sticky sessions make your in-memory session problem disappear without a code change. What have you actually bought?

A. A slow memory leak, because sessions are never evicted from any instance
B. Everyone pinned to an instance losing their state whenever it restarts
C. A limit on how many instances the load balancer will use at once
D. An extra lookup per request, because the balancer must consult a table

27 4. Load, cost and the traffic that is not customers

Which number tells you whether your workers are keeping up with the queue?

A. Jobs completed per minute
B. The average duration of a job
C. The age of the oldest message
D. The number of workers currently running

28 4. Load, cost and the traffic that is not customers

Why publish DMARC at `p=none` before moving to quarantine or reject?

A. Because mailbox providers ignore a policy that appears without warning
B. Because reject requires DKIM alignment, which takes weeks to establish
C. Because none is the only policy that survives message forwarding
D. Because the reports name every system sending as your domain, including the ones you forgot

29 5. Staying alive: the bad day and the day after

Why must at least one backup copy live in a different account or at a different provider?

A. One compromised key or one destroy command must not reach both
B. Providers do not guarantee durability for data written by their own tools
C. Cross-provider copies restore faster, because they are read in parallel
D. Regulations in most countries require a copy outside the primary region

30 5. Staying alive: the bad day and the day after

You have five minutes to spend on account security for a small site. Which two accounts matter most?

A. The cloud console and the managed database
B. The domain registrar and the email account
C. The code host and the CI provider
D. The error tracker and the log platform

31 5. Staying alive: the bad day and the day after

Somebody asks to be erased. What is the honest position on your backups?

A. Backups are exempt, so nothing needs to be said about them
B. The archives must be restored, edited and rewritten without that person
C. Deletion is immediate in live systems and complete once backups age out, after a stated number of days
D. Encryption of the backup counts as erasure once the key is destroyed

32 5. Staying alive: the bad day and the day after

Which failure is characteristic of an expiring OAuth client secret?

A. Requests fail intermittently as cached tokens expire one by one
B. Existing sessions are invalidated, while new sign-ins continue to work
C. Only newly registered users are affected, until they reset a password
D. Sign-in stops working for everybody at once, on a known date

33 5. Staying alive: the bad day and the day after

Three retries in the browser, three in your gateway and three in the service. What has that produced?

A. Nine requests, because the layers share a retry budget

B. Twenty-seven requests for one user action

C. Three requests, since only the innermost layer actually retries

D. An unbounded number, because each retry restarts the outer ones

34 5. Staying alive: the bad day and the day after

Which form of redundancy is genuinely worth it for a one-person project?

A. An active-active deployment across two regions

B. A standby database replicated to a second provider

C. A second mail or model provider behind one interface

D. A duplicate cluster in a second cloud account, kept warm

35 5. Staying alive: the bad day and the day after

A new maintainer removes a guard clause during a tidy-up and the site breaks in a way that takes a day to trace. Which document would have prevented it?

A. A README that explains how to run the project locally

B. A runbook listing the deploy and rollback commands

C. A short decisions file saying why the guard exists

D. An architecture diagram showing where the guard sits

36 6. Shipping a model, not just a web app

You run four worker processes for CPU inference on a four-core box, and each starts four maths threads. What happens?

A. Throughput roughly quadruples, at the cost of higher memory use

B. Sixteen threads contend for four cores and it is slower than one worker

C. The operating system pins one thread per core and the rest idle

D. Latency improves while throughput stays where it was

37 6. Shipping a model, not just a web app

Which lever reduces both latency and cost the most for a language-model feature?

A. Shortening the system prompt
B. Raising the batch size at the serving layer
C. Generating fewer output tokens
D. Lowering the temperature towards zero

38 6. Shipping a model, not just a web app

Every request sends the same three-thousand-token system prompt. What is the cheapest available change?

A. Use the provider's prompt caching for the repeated prefix
B. Move the whole feature to the provider's batch endpoint
C. Split the request into several smaller calls in parallel
D. Fine-tune a model so the instructions are no longer needed

39 6. Shipping a model, not just a web app

A model call returns HTTP 200 with truncated, unparseable JSON. Why must that count as an error in your metrics?

A. Because the provider will not bill for a malformed response
B. Because the retry logic only triggers on recorded error rates
C. Because model failure is often partial, and a 200 is not success
D. Because parse failures indicate a version mismatch in the client library

40 6. Shipping a model, not just a web app

Why serialise the scaler, encoder and vocabulary into the same artifact as the model?

A. It reduces cold-start time by loading one file instead of several
B. Otherwise a mismatched preprocessing artifact makes every prediction quietly wrong
C. It allows the model to be converted to ONNX without a separate export
D. Serving frameworks refuse to load a model whose transforms are missing

41 6. Shipping a model, not just a web app

The new model has won and gone to everybody. Why keep one to five per cent on the old one?

A. To keep the old serving path warm in case a rollback is needed
B. To satisfy the sample size the original experiment required
C. To split the cost of inference between two providers
D. To answer, in month three, whether the system is better or everything drifted

42 6. Shipping a model, not just a web app

Labels for your classifier arrive sixty days late. Which signal can tell you something today, for nothing?

A. The distribution of the model's own predictions, day over day
B. The offline evaluation score on the original held-out set
C. The weekly sample your reviewer has not yet finished labelling
D. The rate at which delayed ground truth eventually disagrees

43 7. Security after the short list

A handler reads `if user.role == "banned": return 403` and otherwise continues. What is wrong with it?

A. It leaks the existence of the resource to a banned account
B. It anticipates one bad case and admits every unlisted one
C. It returns 403 where a banned user should receive 401
D. It checks the role before the session has been validated

44 7. Security after the short list

Why issue a fresh session identifier at login and again on a password change?

A. To reset the sliding expiry so the user is not logged out mid-task
B. To keep the session store from accumulating stale rows over time
C. To rotate the signing key without invalidating other users' sessions
D. To defeat session fixation and to end an attacker's parallel session

45 7. Security after the short list

Why is a cooling-off period — refusing package versions less than a few days old — worth having?

A. New versions are usually reverted, so waiting avoids churn
B. Most compromised releases are removed within hours or days
C. Registries publish security metadata only after a delay
D. It gives your scanner time to build a database entry for the version

46 7. Security after the short list

A volumetric flood of several gigabits per second is aimed at your address. What can your application code do about it?

A. Nothing; the pipe fills before your code runs
B. Shed load with 503 responses and a short Retry-After
C. Rate-limit by account, which the flood cannot forge
D. Serve everything from cache until the flood subsides

47 7. Security after the short list

Why not grant the subscription when the user is redirected back to your success URL?

A. The redirect may arrive before the processor has captured the payment
B. Browsers strip query parameters from third-party redirects
C. The return URL is under the user's control
D. The redirect carries no event ID, so it cannot be deduplicated

48 7. Security after the short list

Why write an admin audit entry in the same transaction as the action rather than after it?

A. It halves the number of round trips to the database
B. It keeps the timestamps in the log strictly ordered
C. Otherwise the log is missing exactly the entries where something went wrong
D. Otherwise the entry can be written by a different connection and lost

49 7. Security after the short list

You suspect a compromise. What comes before cleaning anything up?

A. Snapshot the machine and export the logs somewhere the incident cannot reach
B. Rebuild the server from a known-good image to remove any persistence
C. Publish a notice, since the clock on disclosure has already started
D. Change every password in the organisation, beginning with your own

50 8. Running it for years

Before adopting a managed service, which question is most often skipped?

A. Whether its free tier will still exist in a year's time
B. Whether it is available in the region your database is in
C. Whether its status page is hosted on its own infrastructure
D. How you would get your data out, and what leaving would cost

51 8. Running it for years

Why does skipping four framework major versions turn a week of work into a rewrite?

A. Upgrade guides assume the previous version, and the migration tooling is gone
B. Licences change between majors, so old versions cannot legally be run
C. The package registry removes versions once they leave support
D. Test suites written against an old major cannot be run at all

52 8. Running it for years

Your cost per user is falling month on month while the bill rises. What does that mean?

A. A variable line item has started to dominate the total
B. You are growing into fixed costs that do not move with usage
C. Your active-user count is being measured over too long a window
D. A provider has changed its pricing in your favour

53 8. Running it for years

Why not move the domain and the hosting platform in the same week?

A. Registrars lock transfers for sixty days after any nameserver change

B. Certificates cannot be issued while a transfer is in progress

C. One change at a time tells you which one caused the problem

D. Search engines treat two simultaneous changes as a site relaunch

54 8. Running it for years

You have fallen six months behind on maintenance. Which two items do you do first?

A. Restore a backup, and review who has access

B. Update dependencies, and re-run the load test

C. Rotate every credential, and read the runbook through

D. Take the pending major upgrade, and check the expiry list

55 8. Running it for years

Before arguing about whether to remove a feature, what do you do?

A. Ask the users who have contacted support about it

B. Check whether its code has been touched in the last year

C. Add a counter and leave it running for a month

D. Estimate the maintenance hours it consumes each quarter

56 8. Running it for years

What is the cleanest way for a small site to be done with consent banners?

A. Show a banner that remembers the refusal for a year

B. Set nothing beyond what is strictly necessary

C. Ask for consent only from visitors in regulated jurisdictions

D. Anonymise the identifiers your analytics provider stores

Answers

Each entry gives the correct option, then what each wrong one reveals about how somebody was thinking. The second part is worth more than the first: a wrong answer here is a named misreading, not a mistake.

Lesson checks

1. What a server actually is — A

`B` — Assumes the platform sandboxes away a language feature, rather than simply not keeping any one process alive between requests.

`C` — Assumes latency dominates everything, so an in-process win could not show up in the timings — yet locally the same code showed a clear difference.

`D` — Assumes a behaviour difference between dev and production must come from the build step, which is often true but not here.

2. The first ten minutes on a new machine — D

`A` — Assumes modern OpenSSH rejects Ed25519. It has been the recommended default for years; algorithm mismatch is a real failure, but it is rare and would show as a specific unsupported-type error.

`B` — Confuses a network failure with an authentication failure. A blocked port gives a timeout or connection-refused, not a "Permission denied" from the server.

`C` — Treats the two settings as coupled. Public-key authentication is on by default; the password setting only removes the password path.

3. What keeps your process running when you close the laptop — D

`A` — Plausible, but an unhandled exception normally prints a stack trace to stderr, which the journal captures even when the application's own logger does not.

`B` — The classic beginner failure, but it happens once, at the moment the terminal closes, not repeatedly on a service started by a supervisor.

`C` — Describes what happens after repeated fast restarts, not a cause of the first disappearance — and it would leave the service down, not running again.

4. Why your site is not loading — A

`B` — Treats DNS as a push system that copies records around the world, which is how the phrase 'propagation' makes it sound.

`C` — Blames the client cache, which genuinely does cause stale pages, while ignoring that dig bypasses the browser entirely and still disagrees.

`D` — Reads two different answers as deliberate round-robin load balancing rather than one of them being out of date.

5. What a status code is telling you — A

`B` — Treats the body as the real contract and the status line as decoration, which works only for callers you personally wrote and configured.

`C` — A genuinely common reading: the bytes were delivered fine. But HTTP status describes the outcome of the request, not the health of the pipe.

`D` — Focuses on the triggering event rather than on why three hours passed before anyone knew.

6. The padlock, and what it does not prove — C

`A` — Confuses a name mismatch with an expiry warning; both are red pages, but the browser names which one it is, and a mismatch would have been failing since day one.

`B` — Assumes browsers reliably perform live revocation checks. Most do not — revocation is the weakest part of the system, and it would not present as expiry.

`D` — A plausible cousin, but clock skew produces a "not yet valid" error and would also break outgoing TLS calls from the box, which you would have noticed elsewhere.

7. What sits in front of your app — A

`B` — Would produce a connection refused almost immediately, not a hang, and `docker ps` would show the mapping you could read back.

`C` — Imports HTTPS into a plain HTTP problem; nothing in this setup is terminating TLS, and a handshake failure is an error rather than a hang.

`D` — A real constraint, but port 3000 is above 1024, and it would fail loudly at startup rather than logging that it is listening.

8. One request, with a stopwatch — D

A — Queueing would show up in the monitor's own timings too, since the monitor's requests wait in the same queue as everyone else's.

B — Treats bandwidth as the only client-side variable and ends the investigation, when distance and cold starts are both yours to fix.

C — Misreads caching: a long TTL makes lookups faster for them, and a stale record would break the site rather than slow it.

9. The handshake you keep paying for — A

B — Descriptor exhaustion is real but produces accept failures and errors in the app's own log, not silent 502s from the layer above.

C — Resumption failure costs a round trip; it does not produce 502s, which are the balancer reporting a bad response from upstream.

D — A request timeout usually presents as 504 and correlates with load; these errors appear on idle connections, independent of traffic.

10. The key you must not commit — A

B — Assumes the remote is the only copy, when clones, forks, proxy caches and scrapers all keep their own.

C — Believes exposure can be disproven from access data, when automated scanners read the public events stream without ever cloning.

D — Correctly identifies a real cost of rewriting history and mistakes that inconvenience for the actual risk.

11. The same build, everywhere — B

A — Reasonable instinct, but a dependency mismatch would fail on a package name, not on a relative path to your own file.

C — Shallow clones limit history depth, not which files are present at the checked-out commit.

D — Would mean the file is not committed at all, which contradicts it being in the repository, and would show up as missing in a plain checkout too.

12. Which commit is live right now — C

A — Confuses two versions being live with two builds of one version; the SHAs differ because they came from different commits, not different builds.

B — A cache would return one stale answer consistently, not alternate between two, and a version route is normally uncacheable anyway.

D — Possible but far less likely than the ordinary case; a partial rollback and a deploy in progress look identical from outside, which is why the deploy log matters.

13. The checks that run without you — D

A — True and worth fixing, but minutes are cheap compared with what the habit does to the signal.

B — A real cost, and the ten-minute rule exists because of it, but delay is recoverable in a way that lost trust is not.

C — Often true, and worth investigating — but it describes the flake itself rather than what the re-run habit does to every other check.

14. The console clicks nobody wrote down — C

A — True for resources outside its state, but this rule belongs to a security group under management, so it falls inside the desired state Terraform enforces.

B — Assumes the tool refuses on drift. It does not: it treats the files as the truth and quietly plans the world back towards them.

D — Refresh does read reality, but only to compute the difference — the difference is then removed, not adopted.

15. Deploying for nearly nothing — A

B — Assumes a deploy is a single instantaneous switch, with only one version of the code ever running at a time.

C — Trusts the health check to catch it, but an old instance is genuinely healthy — the schema moved underneath a process that is running fine.

D — Imagines the database buffers around schema changes, when it simply answers that the column does not exist.

16. Ending a process without dropping a request — A

B — Would produce a clean exit at ten seconds with no cut-off requests; the dropped requests show the handler is not draining at all.

C — Invents a special case. Docker always sends SIGTERM first regardless of what the process is.

D — A real cause of dropped requests in general, but it would not explain why stopping the container takes the full grace period every time.

17. The copy you are allowed to break — A

B — A genuinely necessary control, and it would have limited this one — but it is a second line of defence; the sending path itself is still live and the next task might not read addresses from the dump.

C — Detection, not prevention. The question asks what stops it, and forty thousand emails leave in well under a minute.

D — Reduces the window in which nobody notices, but does nothing about whether the messages can be sent.

18. Changing the schema under a running site — C

A — True before Postgres 11 and still true for volatile defaults, but a plain nullable column addition is a metadata-only change regardless of table size.

B — Confuses schema changes with query planning; an unused new column does not change plans for queries that never reference it.

D — A real failure mode worth knowing, but here the statement is the thing waiting, not the thing holding — and cancelling it restored service immediately.

19. Undo, in under two minutes — B

A — An empty cache would produce a miss and a fresh read, which is the harmless case — the symptom here is data being read and rejected.

C — A genuine rollback hazard worth remembering, but a wrong Redis would be empty rather than full of unreadable objects.

D — Migrations do not run backwards during a code rollback, and the error described is happening at deserialisation, before the database is involved.

20. Reading logs at 2am — A

B — Log loss under pressure is a real phenomenon, so this reads as the cautious answer — but it explains the missing logs without explaining the 502s.

C — Assumes a log-level misconfiguration, forgetting that even the successful request lines are missing.

D — Mixes up 4xx and 5xx, treating 'the browser saw an error' as 'the error belonged to the browser'.

21. Where the logs go, and what they cost — D

A — Random strings do compress poorly, but one short field per line is a marginal cost, not the cause of a collapse.

B — Backwards: Loki is designed to scan chunks for text, which is exactly why a request ID belongs in the body rather than in a label.

C — Duplication of a single small field cannot explain a large change; the problem is the number of distinct series, not the bytes.

22. Four numbers that describe any service — D

A — Possible in general, and worth checking — but it abandons the server-side data before reading it properly, and the mean is specifically constructed to hide this case.

B — Real, but it would affect everyone consistently rather than producing intermittent hangs, and it does not explain why the metric looks healthy.

C — Close, and shortening the window helps a little — but even at one-minute resolution a mean over a long-tailed distribution still hides a small fraction of very slow requests.

23. One screen that answers the question — C

A — Resource panels describe the machine, not the outcome; a validation bug that rejects every form uses less CPU, not more.

B — A finer latency percentile finds slow requests, but these requests are fast — they are fast failures that return 200.

D — Useful once errors exist, but the failure here produces successful responses, so no error panel of any granularity shows it.

24. Errors that come to you — A

B — Discards the only view you have of what users actually experience; most user-visible breakage in a modern app happens in the browser.

C — Moves the noise somewhere more disruptive and converts a feed people ignore into notifications people mute.

D — Volume is not the same as importance — a critical bug on the checkout page may fire eleven times a day and would be filtered out.

25. The health check that lies — C

A — A real bug, but a hanging readiness probe removes instances from rotation; it does not order the platform to kill and restart them.

B — Genuinely makes an incident worse and is worth fixing, but extra queries do not cause restarts on their own.

D — Describes why the failure was simultaneous rather than why it became a restart loop, and the loop persisting after recovery points at the probe.

26. Alerts you would get out of bed for — B

A — Puts a label on the problem rather than removing it; people who have learned to dismiss alerts will learn to dismiss a severity too.

C — Better than nothing, and a reasonable second step — but the noisy channel becomes unread, so those conditions are now unmonitored while appearing to be covered.

D — Treats frequency as the defect. Some of those conditions genuinely occur weekly and simply do not need a human; a higher threshold hides them without answering the question.

27. Deciding how reliable it has to be — C

A — Changing the target because you missed it removes the signal; the target is a product decision, not something adjusted to match the current month's luck.

B — True arithmetically and misses the point of a budget: the remaining 40% has to cover 26 more days, which is the decision the policy exists to force.

D — Confuses the budget level with the burn rate; the burn happened during the incident and alerting has already served its purpose.

28. Finding the slow bit — D

A — Possible under write-heavy load, but the database is nearly idle and read queries do not block each other in Postgres.

B — Worth measuring, and occasionally true — but this would show as CPU on the app server rather than as a request that is mostly waiting.

C — A real and common cause of slow endpoints, but it contradicts the given fact that each query runs in under 10ms against the same data.

29. The bill, and the traffic that runs it up — A

B — Treats the alert as a control. Alerts are delayed and inform a person who may be asleep or on holiday; they never stop anything.

C — Caching is the right instinct for repeated work, but heavy users send distinct URLs, which is a cache miss on every single call.

D — A fixed compute price feels like a ceiling, but the model API bills per token no matter where the code that calls it runs.

30. The request you never serve — A

B — The visible symptom people plan for, and much less serious than the one that actually occurs first with a `public` directive on a session-varying response.

C — Many CDNs do apply that default, but the question is what to expect from the header as written — and relying on the CDN's default to save you is not a design.

D — Describes what happens with `Vary: Cookie`, which is not present here; without it the cookie is not part of the key at all.

31. Serving from near the user, and what bandwidth costs — C

A — Size limits exist for very large objects, but ordinary images are well under them; this would not affect the whole static set.

B — Each location does warm separately, which lowers the ratio briefly — but it converges quickly and cannot hold the ratio at 30% indefinitely.

D — Confuses transport protocol with caching; the protocol carrying the bytes has no bearing on whether a response is cacheable.

32. Finding your ceiling before your users do — C

A — Assumes the plateau is CPU-bound; at this shape the constraint is far more often a pool or a lock, and adding CPU changes nothing.

B — Would cap throughput by bandwidth rather than producing a clean plateau with linearly growing queue time.

D — A real caveat for very high loads, but the curve described is exactly the expected shape of a saturated service, not an artefact.

33. The day you run two copies — C

A — It works until an instance stops. The state is still inside a process, so every deploy, crash or scale-in logs out everybody pinned to that instance at once.

B — Stickiness does route reliably in normal operation; the weakness is what happens when the target disappears, not routing accuracy.

D — Distance is not the cost here; all instances are in the same place, and the price paid is fate-sharing with one process, not extra network time.

34. It is almost always the database — B

A — The instinct everyone has, and it trades one problem for a worse one: each connection is a process with real memory cost, and a thousand of them degrade the database itself.

C — Halves the symptom at best, doubles the infrastructure, and does nothing about a model where concurrency and connections are one-to-one.

D — Sounds right, but each serverless instance typically holds a single connection already; the count scales with instances, not with per-instance pool settings.

35. Work that does not belong in a request — D

A — Trades duplicates for silent losses: a crash after marking and before sending means the customer never gets the email and nothing records that.

B — Delays redelivery rather than preventing it, and a long timeout means a genuinely failed job sits unprocessed for that whole period.

C — Removes the safety net for every transient failure — most of which are recoverable — to work around a problem that has a direct fix.

36. The traffic that is not customers — A

B — Raising it helps the blocked users and helps the attacker equally; the shape of the limit is wrong, not its value.

C — A genuine defect worth fixing, but a factor of two does not explain either a blocked city or a sustained brute-force.

D — Possible, and worth checking — but it explains only half of what was observed and leaves the blocked users unaccounted for.

37. Files people send you — D

A — Version-4 UUIDs have enough entropy that enumeration is not the practical risk here; the danger is what happens when a legitimate URL is opened.

B — Expiry is a real operational detail, but a broken link is an availability annoyance, not the security flaw the design contains.

C — They can — the app proxies or presigns — and privateness is not what fails in this design.

38. Why your email went to spam — C

A — Attributes it to an external change that coincides suspiciously with your own, and abandons the investigation at the first plausible excuse.

B — A real and nasty failure mode, but it would cause SPF to fail — and the question states authentication still passes.

D — Volume alone does not push previously-delivering mail into spam, and it does not explain the timing.

39. The restore you have never tested — A

B — A genuine bug worth fixing with pipefail — but a file of plausible size appearing nightly is evidence against it, and it is not the deeper problem.

C — Correctly names the RPO cost, but that is a trade-off they chose knowingly, not the thing that turns an incident into a disaster.

D — Invents a specific mechanism for the failure instead of the plain one: a procedure that has never been run.

40. The short list that actually matters — B

A — Valuable and worth doing, but it defends against one class of bug, and enforced without a report-only period it is most likely to break your own site.

C — Produces a report, mostly of low-severity findings, and does nothing about the credential and exposure paths that account for most real compromises.

D — Consumes the most time for the least benefit, since most alerts are unreachable from your code, and it delays the launch without closing the likely doors.

41. Data you must keep, and data you must delete — D

A — Replicas apply the same deletion automatically through replication; they are the copy you do not have to think about.

B — A real distinction, but the question states the rows were deleted, and soft delete is a design choice you would already know you had made.

C — Expiring sessions do clear themselves, and they are among the least consequential copies.

42. Everything expires, and nothing tells you — A

B — A certificate expiry produces a browser warning on a site that still resolves and responds; here the name does not resolve at all.

C — Worth checking second, and it happens — but a provider outage is public within minutes and does not follow a two-week period of no changes as neatly as a billing date does.

D — Ruled out by the facts given: no change has been made for two weeks, and DNS answers do not degrade on their own after propagating.

43. Timeouts, retries, and the failure you made worse — D

A — Rate limiting would fail the calls to that API, but it does not explain why unrelated endpoints on your own service also stop responding.

B — A plausible secondary effect, but exhaustion of workers occurs long before descriptor limits on a normal service.

C — Synchronised waves are a real problem for the dependency, but the local collapse here is caused by workers held in waiting, not by scheduling.

44. The outage that is not yours — A

B — Adding an integration under pressure is slow and untested; redundancy has to exist before the outage to be usable during it.

C — Longer timeouts hold workers for longer against a dependency that is down, which makes the collapse worse rather than better.

D — Communication is necessary but not a fix; a page that could be serving 95% of its function should not stay at zero while you wait.

45. The first thirty minutes of an outage — C

A — Sounds diligent and is the most common way a short incident becomes a long one; the logs will still be there after the site is working.

B — Guesses at a cause, spends money, and leaves the broken version serving users while you wait to see whether the guess was right.

D — Communication matters and belongs in the first five minutes, but it takes seconds and does not compete with the rollback; delaying the fix in order to investigate is the error.

46. Writing down what happened — B

A — Useful context, but it describes the document's completeness rather than a defect in what it concluded.

C — A genuine omission from the write-up, and it affects prioritisation later — but it does not change what the next incident will look like.

D — The opposite of what a postmortem is for; it reduces what people report and makes future detection worse rather than better.

47. Breaking it on purpose, on a Tuesday — C

A — Failing closed would make things worse; the site correctly continued to serve, and the fault is in how long it waits, not in whether it proceeds.

B — Real outages very often do hang — packets dropped silently is the common case — which is exactly why this drill is representative.

D — Retries would multiply the delay, but the primary evidence is a uniform hang, which points at a single long wait before any retry logic matters.

48. Writing it down for whoever is next — D

A — Sometimes true, and still the wrong first conclusion; assumed knowledge that stops a competent person is precisely the thing the document should have named.

B — Splits the document rather than fixing it, and the reader who got stuck would not have known to look in the second file.

C — Explains the friction away instead of using it, and discards the only test of the document you will ever get for free.

49. A prediction endpoint is a web service with a large secret — A

B — Memory duplication is real and worth watching, but swapping produces far worse and more erratic latency than a threefold rise, and would show in the machine's swap counters.

C — Inference does not usually benefit from a per-worker input cache, and cache locality could not account for a systematic slowdown of this size.

D — Confuses process concurrency with batching; separate processes do not share a batch, and no batch limit is being exceeded.

50. Where inference time actually goes — C

A — The cache is built from the tokens actually processed; a shorter prompt makes a smaller cache, so this is not what keeps the time constant.

B — Prompt tokens are certainly processed — that is prefill — so the premise is wrong even though the conclusion about cost is partly right.

D — Quantisation reduces bytes read per pass across both phases; it does not eliminate prefill or make prompt length irrelevant.

51. What a prediction costs, worked out — C

A — Only true if the requests exist; the cost is fixed but the volume here is not, which is precisely the trap.

B — Assumes providers price against your utilisation. They price against theirs, which is far higher than a small product's.

D — Latency is a real consideration, but the costs here differ by nearly an order of magnitude, so it is not the deciding factor.

52. Which model, which prompt, which version — C

A — Useful for cost and for spotting context growth, but nothing changed in the prompt, so the counts would look normal.

B — Slower responses often accompany provider changes but do not establish that the model behind an alias was replaced.

D — A genuine candidate and worth checking second, but the described change happened with no action on your side, which points at something moving upstream of you.

53. The feature computed two different ways — A

B — Overfitting would usually show as a gap between training and held-out scores; the held-out score here is already good, which points outside the model.

C — Timeouts produce missing predictions and errors in logs, not confident predictions that happen to be wrong.

D — Drift is the right suspicion for a slow decline over months, not for a model that is wrong from the first day it serves.

54. Shadow, then canary, then everybody — A

B — Shadowing tests operational behaviour, not quality; no user outcome was observed, so nothing has been learnt about whether it is better.

C — Discarding the answers is the design, not a flaw; it is what makes shadowing risk-free.

D — Small disagreement is consistent with a real but narrow improvement; it bounds the effect size rather than disproving it.

55. The model that quietly stopped fitting — A

B — Inputs were measured and are stable, so the change is in what those inputs mean rather than in the inputs themselves.

C — Skew would show as a difference between offline and online feature values, and would typically appear as soon as the change shipped, not with a two-month lag.

D — Overstates what those monitors can do: they detect changes in distribution, and adversaries can change behaviour while keeping input statistics similar.

56. The model that trains on its own decisions — B

A — Label delay does bias the most recent window, but it does not explain a trend sustained across a year of retraining.

C — Drift is plausible in general, but the described mechanism is the model removing the evidence for its own decisions, which happens even with a stable distribution.

D — Adaptation is real and would lower recall, but it would not by itself create the systematic absence of labels for intervened cases.

57. What the page says when the model is not there — B

A — It shortens the hang but also cuts off legitimate long generations, trading one failure for another.

C — Doubles the wait and the cost during an incident, and retrying an unhealthy provider is the retry-storm pattern.

D — Without streaming the request still waits for the full timeout, and users lose the perceived-latency benefit for no gain.

58. Authentication is not authorisation — B

A — Random UUIDs have far too much entropy for that to be practical; the risk is disclosure of a valid identifier, not guessing one.

C — Collision probability for version-4 UUIDs is negligible and is not a security consideration at any realistic scale.

D — A real performance consideration for random primary keys, but a performance point rather than the authorisation flaw in question.

59. Sessions, and taking one back — C

A — There is no session record; the whole design avoids one, which is exactly why this option is unavailable.

B — It does work, and it logs out every user of the service at once — a blunt instrument that most teams find unacceptable, though it is sometimes the only lever available.

D — Only if the token verification consults something that changes with the password, which a purely stateless design does not do.

60. The code you did not write — B

A — Only the direct dependencies are fixed; everything they depend on is resolved fresh, which is most of the tree.

C — A genuine downside of pinning without a maintenance schedule, but it is a slow risk, not the immediate hole created by unlocked resolution.

D — It does honour declared ranges; the problem is what those ranges leave undetermined further down the tree.

61. Changing a key while everything is using it — C

A — Skew does break timestamped signatures, but it would affect deliveries before and after the change rather than starting precisely at the deploy.

B — A caching bug is possible, but the described failure occurs even with a correct restart because both sides cannot change at the same instant.

D — Key length would fail consistently rather than for a few minutes, and it would break every delivery, not some.

62. Bots, scrapers and the flood you cannot absorb — B

A — Raising the number helps those users while weakening the limit for everyone; the flaw is in the unit of measurement, not the value.

C — A real misconfiguration to check, but the described pattern — whole institutions affected — is the expected behaviour of per-IP limits even when headers are correct.

D — Overcorrects: anonymous traffic is exactly where abuse arrives, and the answer is a better key plus cheaper responses, not no limit.

63. Taking money without holding card numbers — B

A — Expiry causes a missed grant, which is an availability annoyance rather than the security hole the design contains.

C — Identifiers in URLs are a genuine hygiene problem, but the primary flaw here is trusting a client-controlled request as proof of payment.

D — Ordering varies and duplicate grants are a real concern, but the design's failure is granting on an unverified signal at all.

64. The tools that can do anything — C

A — Reads often go unlogged by design, but their absence follows from what is instrumented rather than from where the write is placed.

B — Revocation removes future access; it does not remove entries already written for past actions.

D — Coarse granularity for bulk operations is a real weakness, but it is a design choice about entry shape, not a consequence of ordering.

65. The first day of a security incident — C

A — True and important, but it argues for also revoking, not against preserving; a team could rebuild and revoke together and still have made the mistake.

B — Plausible if the entry point was a code flaw, but here the entry was a leaked credential, and it still would not be the primary objection.

D — Confuses availability with data protection; notification duties attach to exposure of personal data, not to a maintenance outage.

66. The email about one of your users — B

A — Treats deletion rights as absolute; every regime that grants them also recognises exceptions where another obligation requires retention.

C — Acting on an unverified claim without a record or a route to dispute it is how legitimate content and users get removed by whoever complains.

D — Sets an unrealistic bar and delays a right the user may hold, when the narrow answer is to preserve the specific material rather than freeze everything.

67. What to run yourself, and what to pay for — B

A — A local database on the same machine is often faster, not slower; performance is not the predictable cost here.

C — Most VPS providers resize in minutes; capacity is a solvable problem and not the standing cost of ownership.

D — Bandwidth on a VPS is usually cheaper and bundled; egress is a reason to move to one, not a consequence of doing so.

68. The upgrade you keep postponing — B

A — Size is not the issue; the difficulty comes from how changes are documented and tooled, not from how much code exists.

C — Dependencies do move, but a lockfile can be regenerated at any point; that is a mechanical step rather than the source of the difficulty.

D — Patches are cumulative within a line and are superseded by the newer major; there is no backlog to replay.

69. What one user costs you — B

A — Extrapolates a small step indefinitely; fixed costs eventually saturate and the variable share grows, so the conclusion holds only near current usage.

C — Fixed costs are not waste; a database and a machine must exist before the first user and are the reason early users appear expensive.

D — It is dominated by fixed costs, which is informative in itself — it tells you exactly where you are on the curve rather than making the metric useless.

70. Moving to another provider without losing a URL — B

A — A change to a record does not shorten the caching of the answers already handed out; the old value can persist for up to the previous TTL.

C — Overstates it: changes do take effect, gradually, as cached answers expire — the problem is the delay and its asymmetry, not failure.

D — Caching happens at resolvers everywhere, not at the authoritative provider, so changing provider does not shorten the wait.

71. The hour a week that keeps it alive — C

A — Frequency affects how much data you lose, not whether the backup contains what you need.

B — Unread notifications are a genuine problem, but reading them would not have helped: the script was reporting success truthfully about the wrong thing.

D — Overcorrects: automation is right for running the job, and the gap is which property was being verified, not who verified it.

72. Removing a feature, and retiring a site — A

B — Redirects do pass signals when the target is genuinely equivalent; the flaw is the mismatch between old and new content, not the mechanism.

C — Permanent redirects are cached aggressively, which is a reason for care, but it does not block access to the destination.

D — Traffic from removed pages is small and the homepage is usually the most cacheable page on the site.

73. The pages every public site needs — B

A — Possibly true and a separate problem, but not the reason the document is dangerous to the users it describes.

C — Plenty of adequate policies are written by the people who built the system; accuracy matters far more than authorship.

D — Mentioning a company in a document does not create a relationship with it; the harm is misdescribing your own processing.

74. Keeping it going without burning out — B

A — Monthly and four hours is neither rare nor cheap; accepting it means surrendering nearly a working week a year to one task.

C — Automating the cleanup preserves the fault and hides it; the work is long precisely because the import keeps failing.

D — Misjudges both axes, and documenting a recurring multi-hour task simply makes the loss repeatable by somebody else.

Module test — 1. How the internet reaches your process

1. B

The two answers separate the two possible faults. If the authoritative server returns the new address and a public resolver returns the old one, the change saved and you are waiting on a cached copy to expire. If the authoritative server returns the old address, the change never took effect there — often because the nameservers are somewhere other than the registrar whose form you edited.

2. B

401 means "I do not know who you are, send credentials", so a client that receives it does the only sensible thing and starts an authentication flow — for somebody who is already authenticated, that is a loop. The honest code is 403: I know exactly who you are, and no.

3. C

A web server reads the certificate at start-up and keeps it in memory. Renewal writes a new file; nothing tells the running process. The renewal hook has to reload the server, and the user running the cron needs permission to do it. Clock skew and rate limits are real failures, but they stop issuance rather than leaving a fresh file unserved.

4. D

X-Forwarded-For is an ordinary header that a client can set on its first request, and each hop appends to the list. The leftmost entry is therefore whatever the caller typed. Only an entry appended by a proxy you control is trustworthy, which in practice means reading from the right, or using a header your CDN overwrites rather than appends.

5. A

Roughly four hundred milliseconds of that request was DNS, the TCP handshake and the TLS handshake — round trips across thirteen thousand kilometres, which no amount of server tuning touches. Shortening the distance is the only lever that reaches them. Making a 236 ms handler faster is worth doing later, and it cannot win here.

6. C

If the balancer believes a pooled connection is still usable for longer than your application does, the application will close one at the moment a request is being handed to it. That request fails with nothing on your side to log. Making the application's idle timeout the longer of the two removes it. A wrong bind address fails every request rather than a few; Connection: close costs handshakes, not errors.

Module test — 2. Making a deploy boring

1. C

Automated scanners read the public events stream continuously, and keys posted to public repositories are used within minutes without anybody cloning anything. The old commit also survives in every clone and fork. Only revoking the credential stops it working. Cleaning history is optional tidying, and doing it first leaves the live key live for another ninety minutes.

2. B

The failure is the feature. `npm install` may quietly update the lockfile to satisfy the manifest, so CI can test a dependency tree nobody has committed. `npm ci` refuses to proceed when the two disagree, which turns a silent drift into a red build. Every ecosystem has the equivalent frozen-install command.

3. D

A rolling deploy starts new instances and retires old ones over thirty to ninety seconds, so for that window some requests are answered by each version. That is normal, and it is the single fact behind every hard part of schema change: any migration must be correct for both versions at once, which is why a rename takes three deploys rather than one.

4. A

Create, update and destroy read as what they are. Replace is a destroy followed by a create, which for a database means the data goes and a fresh empty one takes its place — usually because an immutable attribute was edited. This is why `plan` is a review step rather than a formality: a console asks before doing something irreversible, and the tool does not.

5. C

Ten seconds is the default grace period. A container that handles SIGTERM exits in a fraction of a second; one that reaches the deadline was killed. The usual cause is PID 1: the kernel gives PID 1 no default signal handling, and a shell started by `CMD npm start` does not forward signals to its child, so your handler exists and never runs.

6. C

Concurrent index builds are the way to add an index without locking out writes, and the price is that they cannot run inside a transaction — so your tool has to be told to run that migration outside one. There is a second trap worth remembering: if a concurrent build fails it leaves an INVALID index behind, which must be dropped by hand before you retry.

Module test — 3. Knowing what it is doing

1. A

A request ID stamped on every line written while handling that request turns an investigation into a search. With only a timestamp you are filtering four hundred requests in the same second and guessing which one was theirs. It is the cheapest habit in observability and the one with the largest return.

2. B

A percentile is a property of a distribution, not a quantity that can be summed or averaged. To get a fleet p99 you need the buckets from each server added together and the percentile computed from the total, which is exactly why metrics systems store latency as a histogram. If your monitoring only keeps a pre-computed average per minute, no percentile can be recovered from it later.

3. C

A label index is built for values from a small set — service, environment, level. A field that is unique per line makes as many streams as there are lines, which is the exact opposite of what the index is for. Keep request and user IDs in the message body, where full-text search finds them, and label only things with a handful of values.

4. A

A deep readiness check is right, and it must not become part of the problem. Probing every instance every second means constant extra load, and the moment the database is unwell the health checks pile on and help finish it. Cache the answer for a few seconds, put a timeout on every dependency in it, and return 503 rather than a 200 carrying a false field.

5. C

Page on symptoms a user would notice, not on resource numbers. Ninety per cent CPU with everybody happy is a well-used machine; thirty per cent CPU with an exhausted connection pool is an outage. The error rate is user-facing and sustained, which passes both tests: a human is needed, and needed now. The rest belong on a dashboard or in a ticket.

6. D

Burn rate asks how fast you are consuming the month's allowance. A fast rule — a couple of per cent of the budget in an hour — catches the total outage; a slow rule — a tenth of it over six hours — catches the drizzle that never crosses a fixed threshold but ends the month anyway. Both come from the same objective, with no hand-tuned numbers per endpoint.

Module test — 4. Load, cost and the traffic that is not customers

1. A

`public` tells shared caches they may store the response and hand it to anybody. If the response varied by session, the next visitor receives the previous visitor's page, including whatever it contained. A response that differs because of a session cookie is `private, no-store`, and the related habit is never to let a response carrying `Set-Cookie` be cached.

2. B

The stampede happens because many entries expire at the same instant, which is what a fixed TTL guarantees for anything written together. Jitter spreads the expiries so the misses arrive a few at a time. Single-flight — letting the first request compute while the others wait — is the other half, and a longer TTL only makes the same collision less frequent and larger.

3. A

Concurrency equals throughput multiplied by latency: 200 per second times 0.1 seconds is 20 in flight at any moment. So you need at least twenty database connections and enough workers to hold twenty concurrent requests. Run it the other way to find your ceiling — a pool of five with 200 ms latency caps you at twenty-five requests a second, whatever the CPU says.

4. C

Past the knee, extra concurrency adds queue wait rather than throughput. The user who waited thirty seconds has closed the tab, and your server is still holding a connection and doing their work, which makes the queue longer. Serving ninety per cent of people quickly and answering the rest with 503 and a Retry-After delivers far more value than doing all the work and none of the good.

5. D

Every concurrent function instance opens its own connection and there is no shared pool, so concurrency translates directly into backends. Raising max_connections buys a thousand processes competing to do very little. A pooler holds a small number of real connections and multiplexes clients over them — with the caveat that in transaction pooling mode, session-level state does not survive between statements.

6. A

Everything the client says about a file is attacker-controlled, including the content type. Serving attacker-authored HTML from your own origin gives it your cookies and your domain, which is stored cross-site scripting. Check the actual bytes, generate your own key rather than using the supplied filename, and serve user content from a separate domain or as an attachment.

Module test — 5. Staying alive: the bad day and the day after

1. C

A replica faithfully reproduces whatever the primary does, including the destructive statement, usually in under a second. Replication, RAID and provider redundancy all protect against hardware, and almost nothing that destroys a small project's data is hardware. It is a bad migration, a wrong `WHERE` clause, a confident `terraform destroy` or a compromised key.

2. B

The recovery point objective is how much data you lose: everything written since the last good copy, which here is thirteen and a half hours. The recovery time objective is separate — how long the restore itself takes, which for a large dump is one to three hours plus index rebuilds plus finding the file. Nightly dumps are a decision about both, and worth making on purpose.

3. C

Almost every avoidable outage in this module begins with a notice sent to somebody who left. A sunset email, a card expiry warning, a certificate problem, a change of terms — all of them are addressed to whatever mailbox created the account. A forwarding role address is the cheapest fix available, and it costs nothing to set up on the day you open the account.

4. B

Without half-open, a breaker is a switch that turns itself off and never back on. After a cooling period it lets exactly one request through: if it succeeds the breaker closes and normal service resumes, if it fails the breaker stays open and waits again. That single probe is what makes recovery automatic instead of manual.

5. D

Rolling back, disabling a flag or turning off an endpoint stops the damage without destroying anything you need — the logs, the metrics and the bad artifact are all still there afterwards, and diagnosis proceeds calmly. The common way a fifteen-minute incident becomes a two-hour one is somebody choosing to understand it first. End it, then be curious.

6. A

Split every incident into detect, decide and fix. Here the response was competent and the blindness was not: thirty-one minutes with nothing telling you. Fixing the bug prevents that one cause, while the detection gap applies to every future failure whatever it turns out to be. Teams reliably fix the bug and leave the gap, which is why the next unrelated incident also runs for half an hour.

Module test — 6. Shipping a model, not just a web app

1. B

Loading once is the obvious half. The consequence is a warm-up window of seconds to minutes during which the process is listening but cannot answer, and a load balancer that trusts a naive health check will send traffic into it. Readiness has to assert that the model object exists and can score one fixed example — which doubles as a smoke test for a corrupted or swapped file.

2. C

Decode produces one token at a time and each token requires a pass over the weights, so the binding constraint is how fast the weights can be read from memory rather than how fast the chip multiplies. Halve the bytes and you roughly halve the time per token. The same fact explains why a smaller model is faster and why batching raises throughput almost for free.

3. C

$0.80 an hour is $576 a month whether or not anything arrives. One request every ten seconds is about 260,000 a month, which is roughly $0.0022 each — several times a typical hosted API price for the same work. Self-hosting wins on sustained volume and loses on idleness, and small products are mostly idle. The engineering time to run it belongs on the same side of the ledger.

4. A

A model system has four versions that move independently of your code: the provider's snapshot behind an alias, the prompt, the decoding parameters, and the retrieval index. A commit SHA pins none of them. Recording all four with each response is what separates "the model hallucinated" from "the retriever returned the wrong document", which need completely different fixes.

5. D

Per-request assignment gives one person the new model on one page and the old one on the next, which is confusing to use and useless to measure — any behavioural metric is now averaged over a mixture within each user. A stable hash keeps each person in one arm for the whole test, so the product numbers mean something. Shadowing is the stage where you pay for double inference.

6. B

Users can only click what they were shown, so clicks alone teach the next model that yesterday's choices were the right ones and leave everything unshown with no evidence forever. Impressions plus position let you correct for that later; without them you cannot. The companion habits are a small randomised slice and an untouched human-labelled anchor set.

Module test — 7. Security after the short list

1. C

Router-level middleware cannot answer a question about a row that has not been fetched yet. Fetch-then-compare works and is a place every new handler can forget, and handlers get copied. Putting the ownership into the `WHERE` clause makes a forbidden row and a missing row the same code path, so forgetting the check means the query returns nothing rather than somebody else's data.

2. B

The selling point of a stateless token is that nothing is consulted, which is exactly why nothing can be revoked. The standard patch reintroduces a server-side store for the refresh token and leaves a gap of five to fifteen minutes in which a banned account keeps working. That is a defensible design for several services sharing no store, and an odd one for a single application.

3. C

A tag is a mutable label. Whoever controls the repository can repoint `v3` at new content, and every workflow trusting that tag runs it on the next build — which is precisely what happened when a widely used action was modified to dump secrets from build logs. A commit hash names immutable content, so an attacker cannot substitute code under a name you already trust.

4. A

Creating and deploying are the visible steps. Watching is the one that turns rotation from a hope into a fact: until the old key shows no usage you do not know whether the cron job nobody has opened is still using it. Giving your own keys an identifier is what makes that observable, because you can then see who is presenting which key rather than guessing.

5. C

The signature covers the exact bytes the processor sent. A body parser that turns the payload into an object and back into JSON reorders keys and changes whitespace, so the recomputed signature never matches. Verification has to run against the raw request body. It is a very common half-day, and it is why frameworks document a raw-body route specifically for webhooks.

6. A

Ordinarily an honest 403 is right, because the client has to distinguish refused from unauthenticated. The admin surface is the exception: nobody legitimate is guessing at it, and confirming that a path exists is free information for whoever is scanning. Combine that with an explicit allowlist checked on every request and a second factor for the admin session.

Module test — 8. Running it for years

1. B

Stateless services are cheap to operate because failure means starting another one. Stateful services carry backup, restore, replication and major-version upgrade obligations, and those are what consume weekends. Compare money against hours of attention rather than money against money, and the database is usually the best-value thing on the list to pay for.

2. C

An application rolls back to the previous artifact in a couple of minutes. A major database upgrade changes the on-disk format, so going backwards means restoring a backup and losing whatever was written since. What rehearsal on a copy buys you is the duration — and that number, written into the runbook, is your maintenance window rather than a guess.

3. D

The variable half of a bill is nearly always dominated by one thing: an uncached endpoint, an unresized image, a missing index, or a model call that could be a cheaper model. Almost every "we must rearchitect for scale" conversation in a small product is one of those four. Rewrites are the most expensive way to reduce a bill, and they usually leave the offending line untouched.

4. A

A record with a twenty-four hour TTL takes up to a day for the last resolver to notice a change, in both directions — so a cutover made without lowering it first cannot be reversed quickly either. Lower it days ahead, cut over at the quiet hour with the old system still running and still receiving data, and raise the TTL again a week later.

5. C

410 Gone tells machines that the removal was deliberate and permanent, which gets the page out of indexes faster than a 404 that merely looks like a mistake. A 301 is right when there is a genuine replacement. Redirecting everything to the homepage looks tidy and is a small lie: somebody clicked a specific link and received a shrug.

6. B

Sort operational work by frequency and duration. Frequent and short gets automated. Frequent and long is where your life goes, so fix the cause. Rare and short is done by hand from a runbook. Rare and long — this one — is where over-engineering lives: building automation takes longer than the task and will have rotted by the time it is next needed.

Shipping It — final examination

1. B *1. How the internet reaches your process*

A per-request process starts when a request arrives and may be gone before the next one, so a timer set inside it never reaches its deadline. Scheduled work has to move to the platform's own scheduler. The same shape explains the other two surprises: a module-level cache is usually empty, and every concurrent instance opens its own database connection.

2. A ***1. How the internet reaches your process***

`0.0.0.0` means every network interface; `127.0.0.1` would mean this machine only. Databases exposed this way are found and emptied within hours — there are search engines devoted to listing them. The fix is to bind to localhost or a private network, and to close everything except 22, 80 and 443 at the firewall as well.

3. C ***1. How the internet reaches your process***

When a Linux machine runs out of memory the kernel picks a process and kills it, usually the largest one, which is yours. It receives no signal it can catch and writes nothing, so the silence is the signature. `dmesg -T | grep -i "killed process"` finds it. The fix is memory — fewer workers, a smaller heap, a bigger box, a leak found — not a higher restart limit.

4. B ***1. How the internet reaches your process***

The bare domain must carry NS and usually MX records, and a CNAME may not co-exist with other records on the same name. Providers work around it with ALIAS, ANAME or CNAME flattening, which behave like a CNAME for you and resolve to an A record for everybody else — useful to recognise, because it explains why the same instruction differs between hosts.

5. D ***1. How the internet reaches your process***

GET and HEAD are defined as safe, meaning callers are entitled to assume they cause nothing. Browsers prefetch, chat apps fetch link previews, mail servers open every URL in a message to scan it, and crawlers walk everything. All of them will delete post 12. Destructive actions belong behind POST or DELETE.

6. A ***1. How the internet reaches your process***

Every browser that has seen the header will refuse plain HTTP for the domain and its subdomains for a year, and removing the header does not reach a browser that has already stored the policy. That is why you start at a max-age of a few minutes, confirm nothing breaks and raise it in steps — and why the preload list, which bakes the rule into the browsers themselves, is close to permanent.

7. C ***1. How the internet reaches your process***

A new connection costs one round trip for TCP and one or two more for TLS before any byte of your request moves — often eighty to a hundred and twenty milliseconds on top of a five-millisecond call. Under load you also exhaust ephemeral ports. One shared, pooled client created at module scope removes all of it, and it is where the surprise usually is.

8. B ***2. Making a deploy boring***

Those prefixes exist precisely to tell the build tool to inline the value into the JavaScript. Anyone can view source and find it. If the browser needs a paid API, the fix is a route on your own server that holds the key and calls the API on the user's behalf — which also gives you somewhere to put the rate limit you are going to need.

9. D ***2. Making a deploy boring***

The error reads like a corrupt binary and is an architecture mismatch. Build with an explicit platform flag, or build in CI, which is x86 by default. This is one of a family of laptop-specific faults — the other classic is case sensitivity, where an import that works forever on a case-insensitive macOS filesystem fails on Linux with a bare module-not-found.

10. C ***2. Making a deploy boring***

If forked pull requests received your secrets, anybody could open one whose workflow prints your production credentials. It looks broken the first time and it is the correct behaviour. The related trap is that masking only matches the literal value: base64 it, or pass it through a tool that reformats it, and the mask misses and your token is in a public log.

11. D ***2. Making a deploy boring***

The state file maps your code to real resources, and two concurrent writers can leave it inconsistent with the world — after which the tool's idea of what exists is wrong in both directions. A remote backend with locking makes the second apply wait. Two further facts about that file: it contains secrets in plain text, and deleting a resource from your code means deleting it in the world.

12. A ***2. Making a deploy boring***

Anonymising data is worth doing and does not stop a run that is pointed at the live mail provider. The controls that work are structural: a mail catcher in non-production so nothing leaves the machine, or an enforced allowlist of recipient domains — and the general rule that if staging can write to production, staging is production with worse discipline. The same applies to SMS, push and outbound webhooks.

13. B ***2. Making a deploy boring***

A single enormous update keeps one transaction open for the whole run, which stops autovacuum cleaning that table and leaves it bloated, and if it fails at ninety per cent everything rolls back and none of the work survives. Loop small batches with a pause until zero rows are affected, and run the backfill separately from any deploy.

14. C ***2. Making a deploy boring***

Old code reads a cached object it does not understand and throws, so the rollback poisons itself. Include a version in your cache keys and a rollback misses the cache instead. The same family of problems covers queues holding new-format messages, browsers still running the new client bundle, and anything you have already told a third party.

15. B ***3. Knowing what it is doing***

An empty access log during traffic is the loudest thing on the screen. No successful requests either means nothing arrived: the process crashed, failed to start, or is not listening on the port the platform expects. People lose an hour here by reading "no errors in the logs" as good news.

16. C ***3. Knowing what it is doing***

Most volume is 200s that nobody will ever read — health checks and static assets at fifty requests a second are gigabytes a day on their own. Keeping one in a hundred of those and all of the errors preserves every line you would actually search for. Retention is the second lever and it is worth setting on purpose rather than discovering on an invoice.

17. C ***3. Knowing what it is doing***

One minus 0.99 to the twentieth power is about 0.18. A p99 at the service level is close to a p82 at the page level, which is why tail latency matters far more than its name suggests and why "only one per cent of requests" is not the reassurance it sounds like. The naive twenty per cent is close by accident and wrong in method.

18. C ***3. Knowing what it is doing***

A stack means the parts sum to the whole, which is true of request counts by status and false of percentiles — p95 is not p50 plus something. Draw them as three lines on a shared axis with the number you consider bad marked on the chart, and name the panel as the question it answers.

19. B ***3. Knowing what it is doing***

After an unhandled exception the process is in a state nobody has reasoned about — half-written responses, an open transaction, a corrupted in-memory structure. A crash is visible and self-healing: the supervisor starts a fresh process in a second. A process that survives in an unknown state serves wrong answers until a human notices.

20. C ***3. Knowing what it is doing***

If the monitor runs on the same box, cluster or region, it stops reporting exactly when you need it, and nothing distinguishes "healthy" from "gone". A check from somewhere else entirely — a free uptime service, a cron job on another provider — fails loudly instead. It is also the only vantage point that sees DNS, TLS and load balancer failures, which are the worst outages.

21. A ***3. Knowing what it is doing***

Slowness is nearly always waiting rather than computing. If you instrument the query but not the wait to acquire a connection, that time is invisible and you will spend a day optimising SQL that was never slow. The same shape covers waiting for a lock and waiting behind a queue — measure the wait, not only the work.

22. B ***4. Load, cost and the traffic that is not customers***

This is one of the three classic shapes of a shock bill, alongside a retry storm and somebody else discovering an endpoint of yours. Per-invocation pricing with no ceiling means the arithmetic runs away overnight, and a budget alert is a notification rather than a stop. Maximum concurrency limits and hard spend caps are the controls that act without you.

23. B ***4. Load, cost and the traffic that is not customers***

The names are the wrong way round from most people's intuition. `no-cache` permits storage and requires revalidation before reuse; `no-store` is the one that means do not keep a copy at all. `private` means only the browser may store it, not a shared cache. Getting these two confused is how personalised pages end up in a CDN.

24. D ***4. Load, cost and the traffic that is not customers***

The expensive part of a first request is the TCP and TLS handshakes, and those complete against a point of presence near the user rather than across an ocean. The edge then talks to your origin over a connection it already holds open. Dynamic HTML that cannot be cached at all commonly improves by a hundred milliseconds or more, before any caching is considered.

25. A ***4. Load, cost and the traffic that is not customers***

Connections run out first far more often than anything else, followed by one unindexed query that was fine at five thousand rows, then memory, then file descriptors. CPU is genuinely last, and much later than people expect. Most side projects fall over around twenty requests a second, and it is almost never the language or the framework.

26. B ***4. Load, cost and the traffic that is not customers***

And an instance restarts on every deploy. Stickiness also stops load distributing evenly, because a busy long-lived user stays where they were, and it prevents autoscaling from draining an instance without dropping people. Use it deliberately for things that genuinely cannot move, such as an open websocket, not as a substitute for externalising state.

27. C ***4. Load, cost and the traffic that is not customers***

Throughput of five hundred jobs a minute sounds healthy and says nothing about whether arrivals exceed capacity. Oldest-message age says exactly that: if it is climbing, you are falling behind, and the slope predicts when it becomes a customer problem. Graph it and alert on it, along with the depth of the dead-letter queue at greater than zero.

28. D ***4. Load, cost and the traffic that is not customers***

`p=none` changes nothing about delivery and collects reports. Two weeks of them will show you the invoicing tool, the ticketing system and the newsletter platform that also send as you. Going straight to reject is how people stop their own systems from delivering — and the SPF ten-lookup limit means adding a fourth provider can break authentication silently anyway.

29. A ***5. Staying alive: the bad day and the day after***

The point of a backup is not to survive a failed disk — replication handles that. It is to survive you: a bad migration, a wrong `WHERE` clause, a confident destroy, a leaked credential. If the same credentials that can drop the database can also empty the bucket holding its backups, you have one failure away from nothing. Append-only access from the application side is the cheap version.

30. B ***5. Staying alive: the bad day and the day after***

With your email an attacker resets everything else, including the registrar. With DNS control they receive your traffic and can obtain a valid certificate for your domain, which defeats the padlock entirely. Multi-factor authentication on those two, then on DNS, cloud and code host, and recovery codes stored where a second person can reach them.

31. C ***5. Staying alive: the bad day and the day after***

You cannot surgically remove one person from a backup archive without destroying its integrity, and pretending otherwise in a policy is a public claim you are not honouring. Stating the retention window plainly, with a number, is both honest and defensible — and it forces you to have a retention window at all, enforced by a job rather than by a document.

32. D ***5. Staying alive: the bad day and the day after***

It fails cleanly, completely, and on a day that was published a year earlier. That is the whole character of this class of outage: nothing to do with load or bugs, everything to do with a date nobody owned. One file listing what expires, when, and on whose card, plus a calendar entry thirty days before each, retires most of them.

33. B ***5. Staying alive: the bad day and the day after***

Retries multiply through layers, so a dependency that is already struggling receives an order of magnitude more traffic than it did when it was healthy — and the recovery attempts become the second outage. Retry at one layer, deliberately chosen, usually the innermost one that knows whether the call is safe to repeat, with exponential backoff and jitter.

34. C ***5. Staying alive: the bad day and the day after***

Stateless HTTP APIs are interchangeable behind your own interface, so a second mail, SMS or model provider is cheap redundancy you can actually operate. Multi-region and multi-cloud multiply cost and, more importantly, complexity — and complexity causes outages. Single region with good backups and an honest recovery time is a defensible choice, said plainly.

35. C ***5. Staying alive: the bad day and the day after***

A decision with no written reason has no defence, and a reasonable person doing reasonable cleaning will remove it. Three lines are enough: what the rule is, what breaks without it, and where to read more. It is the shortest of the four handover documents and the one that saves the most damage, because the code cannot carry the reason on its own.

36. B ***6. Shipping a model, not just a web app***

The arithmetic library is already using every core, so multiplying workers by threads oversubscribes the machine and the scheduler spends its time swapping between them. This is the most common self-inflicted performance problem in model serving. Set the thread count to one per worker, or run one worker with all the threads, and measure both.

37. C ***6. Shipping a model, not just a web app***

Output length dominates, because each token requires its own pass over the weights during decode and output is priced several times higher than input. Ask for a shorter answer, set a maximum, and prefer structured output over prose. Shortening the prompt helps prefill and the bill, but has less effect on total time than people expect.

38. A ***6. Shipping a model, not just a web app***

Prompt caching reprices a repeated prefix at a fraction of the normal input rate when the same prefix is sent again within a short window, and it usually needs one parameter. Batch endpoints halve the price but only for work that can wait hours, so they suit nightly jobs rather than anything a user is watching. Fine-tuning is a project, not a change.

39. C ***6. Shipping a model, not just a web app***

Model dependencies fail in ways ordinary services do not: truncated output, invalid JSON, a refusal, an empty string, all delivered with a successful status. Validate the shape of every answer before using it and count schema failures as errors — the rate at which structured output stops parsing is also one of the most sensitive early signals that a provider has changed something.

40. B ***6. Shipping a model, not just a web app***

The fitted transforms are part of the model. Deploy the weights against an older encoder and the column order shifts by one, or an unseen category maps to a different index, and every score is wrong with no error message anywhere. Loading them together, plus a checksum of the preprocessing artifact logged at start-up next to the model version, makes that visible.

41. D ***6. Shipping a model, not just a web app***

Without a live comparison you have only a memory of a two-week test. A holdback costs almost nothing and keeps the question answerable — it is also how you notice slow degradation that affects both arms equally, which no absolute metric can show you because there is nothing to compare it against.

42. A ***6. Shipping a model, not just a web app***

Output monitoring is available immediately and moves before the labels do. A mean predicted score that jumps from 0.12 to 0.34 has met something new; so has a rise in the share of predictions sitting near the decision threshold, or in fallbacks and refusals. Input distributions and missingness are the other free signals. Labels then tell you whether quality actually fell.

43. B ***7. Security after the short list***

Default deny is the rule: an unknown role, a missing field, an empty string or an unhandled case must refuse. Code that enumerates the bad states allows everything it forgot, which grows every time somebody adds a role. Write the allow condition and refuse everything else, and keep the rules as small named functions you can test as a table.

44. D ***7. Security after the short list***

Session fixation is an attacker planting a known identifier before you sign in and then using it afterwards; rotating at login makes it worthless. Rotating on a password change is what makes the change meaningful — otherwise the attacker's session sits happily alongside yours. By default a reset should end every other session, including API tokens and mobile apps.

45. B ***7. Security after the short list***

Typosquats and hijacked maintainer accounts are usually caught quickly, and the whole window of exposure is the hours in which the bad version is live. A build that resolves versions freshly can pull a package published ten minutes ago; a frozen install plus a delay removes most of that. It does nothing against a patient two-year social attack, and being honest about that is part of the point.

46. A ***7. Security after the short list***

Layers three and four are handled upstream, by a provider or a CDN whose anycast network spreads the flood across hundreds of locations. Nothing you write matters there. What is yours is layer seven — requests that look legitimate and hit your most expensive endpoint — and application-logic abuse such as signup floods, which are quota problems rather than volume problems.

47. C ***7. Security after the short list***

Anybody can visit your success URL. Access is granted on a signature-verified webhook, or on a server-side check of the payment's status with the processor. The related habits: verify against the raw body, store the event ID and ignore repeats because processors retry until they get a 2xx, and send an idempotency key when you charge so a dropped connection cannot bill twice.

48. C ***7. Security after the short list***

A log written afterwards is skipped precisely when the process crashes, the request times out or the action half-succeeds — which are the cases you most need recorded. Write it before or within the same transaction. And make the reason field mandatory: an action that must be justified in one sentence is an action somebody thinks about first.

49. A ***7. Security after the short list***

The instinct to tidy destroys the evidence you need for every later decision — how they got in, what they reached, when it started, whether it is still live. Logs roll, often within days, and after that the answer is gone permanently. Preserve, then contain narrowly, then rotate credentials, then establish scope from evidence rather than from reasoning about what should have been possible.

50. D ***8. Running it for years***

Read the exit before the entry. A standard dump command and a compatible protocol mean a future switch is configuration; a proprietary store with an export queue and a support ticket means a rewrite, and egress on the way out can be thousands of dollars entirely by design. Lock-in is not automatically bad, and it should be entered knowingly and written into the decisions file.

51. A ***8. Running it for years***

Each upgrade guide is written for people coming from the version before it, the codemods and compatibility shims are removed after a release or two, your dependencies have moved on independently, and nobody remembers why the custom patch in the middle exists. The counter-intuitive rule follows: upgrade more often, in smaller steps, and take one major a quarter.

52. B ***8. Running it for years***

At small scale nearly all of a bill is fixed — machines, database, monitoring, domain — so the first hundred users are extremely expensive each and the number falls sharply as you grow. A cost per user that starts rising is the signal to look, because it means the variable half is taking over, and it is nearly always one line item.

53. C ***8. Running it for years***

Migration failures are rarely dramatic; they are a subset of URLs that stopped working and nobody telling you. With two changes in flight you cannot tell whether it was the DNS, the redirects, the new host's configuration or the transfer. Space them a week apart, lower the TTL days ahead, cut over with the old system still alive, and keep its data for a month.

54. A ***8. Running it for years***

The recovery is not six months of maintenance in one weekend. One of these protects the data and the other protects it from people who should no longer reach it, and both are cheap. Everything else — upgrades, load tests, alert reviews, rotations — can wait a fortnight without much changing, and will be easier once those two are known to be sound.

55. C ***8. Running it for years***

One log line turns an opinion into a number, and the number is usually decisive: a feature with an assumed loyal following has eleven users, four of them your own test accounts. Occasionally it goes the other way and something you thought was dead is load-bearing for a group nobody spoke to. Either way you are now arguing about evidence.

56. B ***8. Running it for years***

Cookies strictly necessary for the service need no consent; analytics and advertising do. Privacy-respecting analytics measure page views without a per-visitor identifier, several are free to self-host, and your own server logs answer most of the same questions. This is the rare compliance problem with a better technical answer than a legal one.