

COMPANION VOLUME • STAGE 12 OF THE ENGINEER'S SYLLABUS

The Engineer's Watch

Code that isn't running isn't doing anything. This volume covers the distance between your laptop and a system that stays up: the machine underneath, the pipeline that ships to it, the signals that say how it's doing, and the habits that get you through the night. Engineers who can operate what they build are worth disproportionately more than those who can only write it.

CHAPTERS

20

CODE EXAMPLES

29

EXERCISES

60

STAGE BUDGET

220 h

Contents

How to use this volume	4
Part I · The machine	5
Chapter 1 · Linux in production	6
Chapter 2 · Diagnosing a sick box	13
Chapter 3 · Performance and capacity	19
Part II · Packaging and platform	25
Chapter 4 · Containers	26
Chapter 5 · Orchestration with Kubernetes	33
Chapter 6 · Configuration and secrets	40
Chapter 7 · The cloud underneath	46
Part III · Delivery	52
Chapter 8 · Infrastructure as code	53
Chapter 9 · Continuous integration	60
Chapter 10 · Deploying safely	66
Part IV · Observability	71
Chapter 11 · Metrics	72
Chapter 12 · Logs	78
Chapter 13 · Traces	84
Chapter 14 · Dashboards and alerts	90
Part V · Reliability	96
Chapter 15 · SLIs, SLOs and error budgets	97
Chapter 16 · On call and incident response	102
Chapter 17 · Postmortems	108
Part VI · Keeping it running	114
Chapter 18 · Backups and recovery	115
Chapter 19 · Cost as an engineering constraint	120

Chapter 20 · The operating discipline	125
The twenty rules	131
Reading list	133

The chapters follow the shape of the rest of the series: the idea in one sentence, worked examples, the traps, then exercises with folded solutions, a checklist and further reading. Unlike the earlier volumes, much of this one only becomes real when you run it. Read a chapter, then do its part of the stage deliverable on something you can break.

Prerequisites. Operating systems and networking (stages 05 and 06) are assumed, as is having shipped something (stage 10). Distributed systems (stage 11) helps but isn't required: where a chapter leans on it, it says so.

The running example. The invoicing product from *The Engineer's Shipyard* is now live: a web service, a worker pool, a Postgres database, a queue and an object store, serving real customers who notice when it's down. Everything here is about keeping that specific system healthy, and about what you'd do differently next time.

Tools change; the questions don't. The particular tools named here will age: this volume was written in 2026, when Kubernetes was on its third release of the year and half the industry was migrating off an ingress controller that had just been retired. What lasts is the reasoning: what a resource limit does to a process, why a percentile can't be averaged, what an error budget buys you, and why the first move in an incident is to mitigate rather than to understand.

Pair it with the Google SRE books, all of which are free online. This volume gives the working shape of the job and the practice; those give the depth and the war stories. The stage's deliverable is the real test: take the product from stage 10, put it on infrastructure defined entirely in code, with a deployment pipeline, metrics, traces, two alerts worth waking someone for, and a runbook. Then break it deliberately and write the postmortem.

THE ONE QUESTION

Every practice in this volume answers one of two questions, and you can hold the whole stage in your head by asking them of any system you run.

How will I find out? Not "is there a dashboard", but: which signal moves when this breaks, who sees it, and how long before they do. And: **what will I be able to do about it at three in the morning?** One command to roll back, one runbook that says what to check, one switch that sheds load, one restore that has actually been tested. A system you can't observe and can't change safely is one you don't really operate.

PART I

The machine

Three chapters on the ground everything else stands on: the Linux box your code runs on, the method for working out why it's sick, and the arithmetic of latency, queues and capacity.

Linux in production

3 exercises

Your code runs as a process on a Linux machine, under an init system that decides when it starts, what it may consume, and what happens when it stops. Most production surprises come from those rules rather than from your logic.

What a process actually is

A running service is a process with an identity and a set of budgets: a user and group, a working directory, an environment, open file descriptors, memory, and a share of the CPU. Nearly every mysterious production failure is one of those budgets running out.

- **File descriptors.** Every socket, file and pipe is one, and the default limit is often 1,024. A busy server with connection pools and keep-alive connections passes that easily, and then fails with "too many open files", usually while accepting new connections, which looks like a network problem.
- **Memory.** The kernel will hand out more than it has and hope; when it runs out, the OOM killer picks a process and kills it with SIGKILL. Under a cgroup limit, your container is chosen first, and the exit status is 137, which is $128 + 9$.
- **CPU.** A quota is enforced by throttling: the process is stopped for the rest of each 100 ms period once it has used its share, which shows up as latency, not as high CPU.
- **Threads and tasks.** A limit on tasks turns "can't create thread" into a crash under load.

systemd: the thing that runs your thing

On a plain Linux host, systemd is process 1. It starts your service, restarts it when it dies, applies the limits above through cgroups, collects its output into the journal, and orders it against other units. A unit file is the whole operational contract in one page, which is why it's worth writing carefully rather than copying.

EXAMPLE /etc/systemd/system/invoicing-api.service

```
[Unit]
Description=Invoicing API
After=network-online.target
Wants=network-online.target
# Flapping protection lives here, not in [Service]: give up after 5 starts in a minute.
StartLimitIntervalSec=60
StartLimitBurst=5

[Service]
Type=exec
User=invoicing
```

```

Group=invoicing
WorkingDirectory=/srv/invoicing
# Secrets in a root-owned 0600 file, not in the unit: unit files are world-readable.
EnvironmentFile=/etc/invoicing/env
ExecStart=/srv/invoicing/venv/bin/gunicorn --bind 127.0.0.1:8080 app:application
Restart=on-failure
RestartSec=2s
TimeoutStopSec=30s
LimitNOFILE=65535
MemoryHigh=1200M
MemoryMax=1500M
CPUQuota=200%
TasksMax=512
NoNewPrivileges=true
ProtectSystem=strict
ProtectHome=true
PrivateTmp=true
ReadWritePaths=/var/lib/invoicing
StandardOutput=journal
StandardError=journal

[Install]
WantedBy=multi-user.target

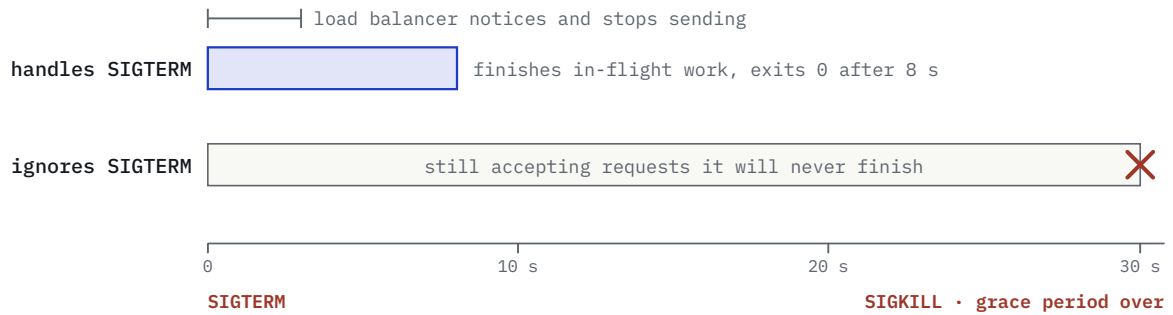
```

Three lines there deserve attention. `MemoryHigh` below `MemoryMax` means the kernel throttles and reclaims before it kills, which turns some out-of-memory kills into slowness you can alert on. `Restart=on-failure` with a start limit stops a crash loop from hammering a database for hours. And `ProtectSystem=strict` makes the filesystem read-only except the paths you name, so a compromised or confused process can't rewrite the host.

The commands you'll use daily are few: `systemctl status invoicing-api` for state and recent logs, `systemctl restart`, `journalctl -u invoicing-api -f` to follow output, `systemctl cat` to see the effective unit, and `systemd-analyze verify` to check one before it goes near a machine.

Shutdown is part of your program

Stopping a service is a negotiation. `systemd` sends `SIGTERM` and waits `TimeoutStopSec`; Kubernetes does the same with `terminationGracePeriodSeconds`, 30 seconds by default. If the process hasn't exited by then, it gets `SIGKILL`, which it cannot catch: in-flight requests die, buffers are lost, half-written files stay half-written.



The grace period is a contract. The platform promises not to kill you for thirty seconds; your process promises to stop taking new work, finish what it holds, and exit. Ignoring the signal doesn't buy time, it just moves the loss to the end.

EXAMPLE worker.py · finish the job in hand, then stop

```
import queue
import signal
import sys
import threading

jobs: queue.Queue = queue.Queue()
stopping = threading.Event()

def on_sigterm(signum, frame):
    stopping.set()          # the handler does one thing: raise a flag

signal.signal(signal.SIGTERM, on_sigterm)

def run_worker(process, poll_s=0.2):
    """Take no new work after SIGTERM, finish the job in hand, then exit cleanly."""
    while not stopping.is_set():
        try:
            job = jobs.get(timeout=poll_s)
        except queue.Empty:
            continue
        process(job)        # a job that started before the signal runs to completion
        jobs.task_done()
    sys.exit(0)             # unclaimed work stays on the queue for the next worker
```

For an HTTP server the shape is the same, with one extra step at the front: fail the readiness check first, wait a few seconds for load balancers to stop routing to you, and only then stop accepting. *The Engineer's Fleet*, chapter 15, has the full ordering. Two rules make the rest work: keep individual units of work shorter than the grace period, and make them idempotent, because sooner or later one will be cut in half.

Where things are, and who you are

Run as a dedicated unprivileged user, never root. A service that needs a port below 1024 doesn't need root for it: give the binary the `CAP_NET_BIND_SERVICE` capability, or bind to 8080 and let a proxy or a load balancer map the port. Configuration belongs in `/etc`, variable state in `/var/lib`, logs in the journal or `/var/log`, and runtime facts about processes in `/proc`, where `/proc/<pid>/limits`, `/proc/<pid>/status` and `/proc/<pid>/fd` answer most "what is it actually doing" questions.

In a container, write logs to standard output and let the platform collect them. On a plain host, write to the journal or to files with `logrotate` configured. A disk filled by logs is one of the most common self-inflicted outages there is, and it takes down everything on the machine at once, usually starting with the database.

COMMON TRAPS

Default file descriptor limits on a server holding thousands of connections.

Ignoring SIGTERM, so every deploy drops requests.

Restart=always with no start limit, turning one bad deploy into an infinite crash loop.

Secrets in the unit file, which is world-readable, or in the process's command line, which every user can see.

No memory limit, so a leak takes the whole machine rather than one service.

Logs to an unrotated file until the disk fills.

EXERCISES

EXERCISE 1.1 Review the unit file

A colleague submits this unit for a payment worker. Find at least four problems.

```
EXERCISE payments-worker.service
```

```
[Unit]
Description=Payments worker

[Service]
Environment=DATABASE_URL=postgres://app:hunter2@db/payments
ExecStart=/usr/bin/python3 /opt/payments/worker.py
Restart=always
RestartSec=0
KillSignal=SIGKILL
```

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

SOLUTION

- **A password in the unit file.** Unit files are world-readable and end up in version control. Use `EnvironmentFile=` pointing at a root-owned 0600 file, or a secrets manager.
- **It runs as root,** because no `User=` is set. Add a dedicated user, plus `NoNewPrivileges=true` and the `Protect*` options.
- **KillSignal=SIGKILL** means the worker is never asked to stop politely. Every restart and deploy kills a payment mid-flight. Remove it and handle SIGTERM.
- **Restart=always with RestartSec=0 and no start limit** is a crash loop at full speed. Use `Restart=on-failure`, a delay, and `StartLimitIntervalSec / StartLimitBurst` in `[Unit]`.
- **No resource limits** and no ordering after the network. Add `MemoryMax=`, `TasksMax=`, and `After=network-online.target`.

EXERCISE 1.2 Too many open files

At peak the API starts logging `OSError: [Errno 24] Too many open files` and stops accepting connections, while CPU and memory look fine. Walk through the diagnosis and the fix.

SOLUTION

1. **Measure.** `ls /proc/<pid>/fd | wc -l` for the current count and `cat /proc/<pid>/limits` for the ceiling actually in force, which is the one the unit set, not what your shell shows.
2. **Decide whether it's a leak or a ceiling.** If the count grows and never falls, something isn't being closed: unclosed HTTP responses, a connection pool created per request, files opened in a loop. Raising the limit hides a leak for a day and then it fails again.
3. **Set the ceiling deliberately.** `LimitNOFILE=65535` in the unit, or in the container runtime. A server holding 20,000 keep-alive connections needs more than 20,000 descriptors, plus pools and files.
4. **Alert on the ratio,** open descriptors against the limit, well before it's reached.

EXERCISE 1.3 Exit code 137

A worker restarts every few hours with exit code 137 and nothing in its own logs. What happened, how do you confirm it, and what are the three possible fixes?

SOLUTION

What happened: 137 is $128 + 9$, so the process received SIGKILL, and a process that dies silently under memory pressure was almost certainly killed by the kernel's out-of-memory killer when it exceeded its cgroup limit.

Confirm it in the kernel log (`dmesg -T | grep -i oom` or `journalctl -k`), which names the process and the limit, or in the container platform's event for the pod, which says `OOMKilled` .

The three fixes, in the order you should consider them: find the leak or the unbounded buffer, because steady growth over hours is usually a leak; make the workload fit, for example by streaming a large file instead of reading it into memory, or by capping a runtime's heap below the container limit rather than letting it discover the limit by dying; or raise the limit, if the workload genuinely needs the memory and the machine has it. Setting `MemoryHigh` below `MemoryMax` converts some of these kills into reclaim and slowness, which you can see coming.

CHECKLIST

- ☐ Does every service run as a non-root user with explicit memory, CPU, task and descriptor limits?
- ☐ Does your process handle SIGTERM, and does its work fit inside the grace period?
- ☐ Is restart behaviour bounded, so a bad deploy can't loop forever?
- ☐ Are secrets out of unit files, command lines and images?
- ☐ Can logs fill the disk?

GO DEEPER

[DOCS](#) [systemd.service](#) and [systemd.resource-control](#) the reference for unit files

[DOCS](#) [Control Group \(Cgroup\) 1](#) [1.1](#) [1.1.1](#) [1.1.1.1](#)

How Linux Works, 3rd edition Brian Ward · the clearest tour of the system underneath

Diagnosing a sick box

3 exercises

When a machine misbehaves, a method beats a favourite tool. Walk every resource asking three questions, and ask of every slow thing whether it is running or waiting. The answer picks your next command.

The first minute

Netflix's performance team published a checklist of what to run in the first sixty seconds on an unfamiliar, unhappy Linux box. It's still the fastest way to orient yourself, and it costs nothing to learn.

COMMAND	WHAT YOU'RE LOOKING FOR
<code>uptime</code>	Load averages over 1, 5 and 15 minutes: is this getting worse or better?
<code>dmesg -T tail</code>	Out-of-memory kills, disk errors, dropped packets, hardware complaints
<code>vmstat 1</code>	Run queue (r), blocked tasks (b), swap in and out, CPU split including steal
<code>mpstat -P ALL 1</code>	One hot core versus all cores busy: a single-threaded bottleneck shows here
<code>pidstat 1</code>	Which process is using the CPU, sampled over time rather than as an average
<code>iostat -xz 1</code>	Per-device throughput, queue size and average wait: storage saturation
<code>free -m</code>	Available memory, not "used": page cache is not a leak
<code>sar -n DEV 1, sar -n TCP,ETCP 1</code>	Interface throughput, retransmits, connection errors
<code>top</code>	Everything at once, as a cross-check on the above

Two of those numbers are routinely misread. **Load average** on Linux counts processes waiting for disk as well as for CPU, so a load of 40 on eight cores can mean a saturated disk and an idle processor. And **steal time**, the `st` column in `vmstat`, is time your virtual

machine wanted the CPU and the hypervisor gave it to someone else: your code didn't get slower, your neighbour got busier.

The USE method

Brendan Gregg's **USE method** turns "the box is slow" into a finite checklist. For every resource, check **utilisation** (how much of the time it's busy), **saturation** (how much work is queued for it) and **errors**. Saturation is the one people skip, and it's usually where the answer is: a disk at 100% utilisation with no queue is fine, and a disk at 60% with thirty requests waiting is the outage.

RESOURCE	UTILISATION	SATURATION	ERRORS
CPU	mpstat	vmstat r, cgroup throttling	machine check events
Memory	free -m	vmstat si/so, reclaim, OOM kills	failed allocations
Disk	iostat %util	iostat aqu-sz, await	dmesg I/O errors
Network	sar -n DEV	retransmits, backlog overflow	interface errors, drops
Descriptors, tasks	open against the limit	allocation failures	EMFILE, "can't create thread"

Running, or waiting?

Before reaching for any profiler, answer one question: is the slow thing burning CPU, or waiting for something else? The two have completely different tool sets, and you can tell them apart by comparing CPU time with wall-clock time.

EXAMPLE where.py · on-CPU or off-CPU

```
import hashlib
import resource
import time

def where_did_the_time_go(fn, *args):
    """Wall time against CPU time: is this work running, or waiting for something?"""
    r0, t0 = resource.getrusage(resource.RUSAGE_SELF), time.monotonic()
    fn(*args)
    r1, t1 = resource.getrusage(resource.RUSAGE_SELF), time.monotonic()
    wall = t1 - t0
    cpu = (r1.ru_utime - r0.ru_utime) + (r1.ru_stime - r0.ru_stime)
    return wall, cpu, cpu / wall
```

```
# hashing wall 0.42 s cpu 0.42 s on-CPU 98% -> profile the code
```

```
# waiting wall 0.51 s cpu 0.00 s on-CPU 0% -> find out what it is waiting for
```

High CPU means a **CPU profile**, best read as a **flame graph**: stacks on the x-axis sorted alphabetically, depth on the y-axis, width proportional to time. You read it by looking for wide plateaus, which are where the time goes, and ignoring the height. Near-zero CPU with high latency means the process is blocked, and the question becomes what on: a lock, a disk, a database, a DNS lookup, a downstream service. That's **off-CPU analysis**, done with off-CPU flame graphs, with `strace -c -p <pid>` for a syscall summary, or with the application's own thread dump.

Modern Linux tooling for both is eBPF: `bpftrace` for one-liners and short scripts, the `bcc` collection for ready-made tools such as `biolatency`, `execsnoop` and `offcputime`. These attach to a running kernel with no restarts and low overhead, which is why they've become the default way to see inside a production box.

The parts people forget

- **Sockets and backlogs.** `ss -ltn` shows listening sockets with `Recv-Q`, the number of connections waiting to be accepted, and `Send-Q`, the accept queue's size. A full accept queue silently drops connections: clients see timeouts while your server looks idle.
- **Connection tracking and ports.** A NAT gateway or firewall with a full connection-tracking table, or a client that has exhausted its ephemeral ports, produces failures that look like packet loss.
- **DNS.** Slow or failing name resolution is one of the most common causes of "everything is a bit slow", and it rarely appears on a dashboard.
- **The container's own limits.** Inside a cgroup, the numbers in `top` describe the host, not you. Read `memory.current` against `memory.max`, and `cpu.stat`, whose `nr_throttled` and `throttled_usec` tell you how often the kernel stopped you at the ceiling.

Working the problem

During an incident the method is narrower than this: mitigate first, diagnose second (chapter 16). But whenever you are actually diagnosing, three habits pay for themselves. Form a hypothesis before running the command, so the output means something. Change one thing at a time, or you'll never know which change worked. And write down what you ran and what it said, because in two hours you will not remember, and the postmortem will need it.

COMMON TRAPS

Reading load average as CPU. On Linux it includes tasks waiting on disk.

"Memory is nearly full." Page cache counts as used: look at available.

Averages over a minute hiding a five-second stall that timed out every request.

Host metrics inside a container, which describe the machine, not the cgroup.

Leaving strace attached to a busy production process, which can slow it dramatically.

Changing three things at once and declaring the third one the fix.

EXERCISES

EXERCISE 2.1 Symptom to first command

What do you run first, and what would confirm the hypothesis?

- a. Requests are timing out; CPU on every instance is under 20%.
- b. One instance out of twelve is slow; the others are fine.
- c. The service was fine for six days and is now slow, with no deploy since.
- d. Latency is fine from inside the cluster and terrible from the internet.

SOLUTION

- a. Low CPU with timeouts means waiting. Check saturation and waiting: connection pool usage, `ss -ltn` for a full accept queue, downstream latency, and locks in the database. Confirm with an off-CPU profile or a thread dump showing where threads are blocked.
- b. Compare that instance against its peers: noisy neighbour (steal time), a failing disk (`dmesg`), a different kernel or configuration, or an unbalanced share of traffic. Confirm by removing it from the load balancer and seeing whether its own metrics recover.
- c. Six days of growth points at a resource that accumulates: memory, file descriptors, threads, disk, a cache that never expires, or a table that has crossed a size where a query plan changed. Plot those four counters over the week.
- d. The problem is in the path, not the process: TLS termination, the load balancer, DNS, a CDN, or a connection-tracking table. Measure from outside with `curl -w` timing breakdowns, which separate DNS, connect, TLS and first byte.

EXERCISE 2.2 Read the numbers

An eight-core machine shows a load average of 40, CPU at 15% user, 5% system and 60% iowait, `vmstat` with `r` around 2 and `b` around 30, and `iostat` showing the data disk at 98% utilisation with an average wait of 400 ms. What is wrong, and what do you do?

SOLUTION

The CPU is idle. With `r` at 2 there are almost no processes waiting to run; with `b` at 30 there are thirty blocked on I/O, and that's what inflates the load average to 40. The disk is saturated: 98% utilisation with a 400 ms average wait means requests are queueing at the device.

Next: find who is doing the I/O (`pidstat -d 1` , or `biolatency` and `biosnoop` for the distribution and the callers), and what changed. Common causes are a backup or batch job running in the hours, a query that lost its index and is scanning, log volume, or a cloud volume whose burst credits have run out, which is a very common surprise: the disk performed well for weeks and then didn't. Mitigate by stopping the competing work or raising the volume's provisioned throughput; fix by making the workload fit the device.

EXERCISE 2.3 Throttled, not busy

A container has a CPU limit of 0.5 cores. Its dashboard shows average CPU at 45% of the limit, yet p99 latency is terrible. Inside the container, `cpu.stat` reports 6,000 periods and 2,400 of them throttled. Explain, then fix.

SOLUTION

The kernel enforces a CPU limit over short windows, by default 100 ms. A limit of 0.5 cores means 50 ms of CPU per 100 ms period. A request that needs 80 ms of CPU cannot have it: the container is stopped partway and resumes in the next period, adding tens of milliseconds of pure waiting. Averaged over a minute the container looks half idle, while 40% of periods hit the ceiling.

The fix is to stop thinking in averages. Raise or remove the CPU limit, keeping the **request** as the real reservation, since limits throttle even when the node has spare capacity. If you keep a limit, set it well above the request, such as one and a half to two times, so bursts fit. And alert on throttled periods, which is the signal that actually predicts latency.

CHECKLIST

- ☐ Can you get utilisation, saturation and errors for CPU, memory, disk and network on any box you run?
- ☐ Do you know whether your slowest endpoint is on-CPU or waiting?
- ☐ Are container-level metrics, including CPU throttling, collected as well as host metrics?
- ☐ Can you take a CPU profile of a production process without restarting it?
- ☐ Do you write down what you ran during an investigation?

GO DEEPER

ESSAY The USE Method Brendan Gregg

REFERENCE Linux performance tools, and where they attach Brendan Gregg

REFERENCE eBPF tracing tools bcc and bpftrace

BOOK Systems Performance, 2nd edition Brendan Gregg · the reference for this chapter, and the next

Performance and capacity

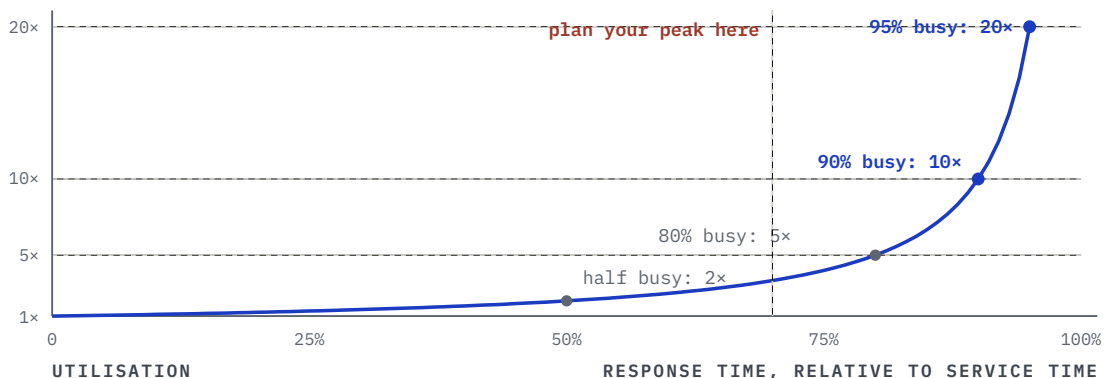
3 exercises

Latency isn't a number, it's a distribution, and it climbs steeply as a queue fills. Measure the distribution, respect the queue, and size the system for the peak you'll actually get, with room to lose part of it.

Two laws worth knowing by heart

Little's law says that the average number of requests in flight equals the arrival rate times the average time each one takes: $L = \lambda W$. It holds for any stable system, whatever it does inside. At 500 requests a second with an average latency of 200 ms, 100 requests are in your system at any moment, so a connection pool of 40 is a bottleneck you built yourself. Run it backwards to size things: 64 workers, each taking 200 ms, can serve 320 requests a second and not one more.

The second law is what happens as you approach that ceiling. For a simple queue, the response time is the service time divided by the idle fraction: $W = S / (1 - \rho)$. The shape of that curve is the single most useful thing to know about capacity.



Response time against utilisation for a single queue. Nothing much happens until about 60%, then the curve turns: at 90% busy, work waits nine times longer than it takes to do. This is why a system sized to run at 95% of capacity feels broken while every dashboard says there's headroom.

Percentiles, and what they hide

An average latency is the least interesting number you can compute, because it describes nobody: it's dragged around by both the fast majority and the slow tail. Percentiles describe users. The p99 is the experience of the unluckiest request in a hundred, and on a page that makes twenty requests, most users meet it several times a session (*The Engineer's Fleet*, chapter 14).

Two rules follow. **Percentiles cannot be averaged.** The mean of ten instances' p99s is not the fleet's p99, and the error is not small. In a run with nine healthy instances and one sick one, the mean of the ten p99s was 152 ms while the true p99 across all the requests was 916 ms: averaging hid a sick instance almost perfectly. To get a real percentile you need the underlying distribution, which is why metrics systems store histograms (chapter 11). **And percentiles are not additive:** a request that fans out to five services, each with a 1% slow tail, is slow far more often than 1% of the time.

Your load generator is probably lying

Gil Tene named the most common measurement bug **coordinated omission**. A load generator with a fixed number of workers sends the next request only when the last one returns. When the server stalls, the generator stops sending, so the requests that would have hit the stall are never made, never measured, and never appear in the tail. The result is a beautiful latency report from a system that was unusable.

EXAMPLE loadgen.py · two ways to measure the same server

```
def closed_loop(port, seconds):
    lat, end = [], time.monotonic() + seconds
    while time.monotonic() < end:
        t0 = time.monotonic(); one_request(port); lat.append(time.monotonic() - t0)
    return lat

def open_loop(port, seconds, rate):
    lat, lock, threads = [], threading.Lock(), []
    start = time.monotonic()
    def fire(intended):
        try:
            one_request(port)
        except Exception:
            return
        with lock:
            lat.append(time.monotonic() - intended)      # measured from when it SHOULD have
                                                         been sent
    for i in range(int(seconds * rate)):
        intended = start + i / rate
        delay = intended - time.monotonic()
        if delay > 0:
            time.sleep(delay)
        t = threading.Thread(target=fire, args=(intended,), daemon=True)
        t.start(); threads.append(t)
    for t in threads:
        t.join(timeout=10)
    return lat
```

Pointed at the same server, a toy that serves in 5 ms but stalls for 200 ms once a second, the two generators describe different worlds:

GENERATOR	MEDIAN	P99	OVER 100 MS
Closed loop, one worker	7.9 ms	210 ms	1.0%

Open loop, 100 requests a second	51 ms	213 ms	36%
----------------------------------	-------	--------	-----

Same server, same stalls. One report says one request in a hundred was slow; the other says more than a third were. The open-loop generator is right, because real users arrive whether or not you're ready for them, and it measures each request from the moment it should have been sent. Use generators that hold a fixed arrival rate and correct for this, such as k6, Gatling, wrk2 or Vegeta, and be suspicious of any tool whose concurrency you set instead of its rate.

Load testing that tells you something

- **Test at rates, not at concurrency, and ramp:** 10%, 25%, 50%, 75%, 100%, 125% of expected peak. You're looking for the knee in the curve above, not a single pass or fail.
- **Find the breaking point deliberately,** so you know what happens beyond it: graceful degradation, or collapse.
- **Use realistic data and cache states.** A test against a warm cache and a thousand-row table measures nothing.
- **Watch the generator too.** If it's CPU-bound or running out of ports, you're measuring the generator.
- **Record the system's own metrics during the run,** not just the client's: utilisation, saturation and queue depths tell you which resource broke first.

Capacity planning, on one page

Capacity is arithmetic once you have a number for what one instance can do. Suppose peak traffic is 12,000 requests a second, and load testing puts the knee for one instance at 800 a second. Run instances at 60% of that, keeping the latency multiplier near two:

STEP	WORKING	RESULT
Usable capacity per instance	$800 \times 60\%$	480 rps
Instances for peak	$12,000 \div 480$	25
Survive losing one of three zones	$25 \times 3 \div 2$	38
Growth until the next review	38×1.15	44 instances

Then check the same arithmetic against the things that don't autoscale: database connections, the connection limit on the database itself, queue consumers, third-party rate limits, and licences. The bottleneck at 44 instances is rarely the instances.

COMMON TRAPS

Averaging percentiles across instances, shards or time windows.

Closed-loop load tests that omit exactly the traffic you needed to measure.

Planning to run at 90% utilisation because the hardware is expensive.

Forgetting failover headroom, so losing a zone takes the remaining two down with it.

Benchmarking one component and assuming the system inherits its numbers.

EXERCISES

EXERCISE 3.1 Size it with Little's law

An API handles 900 requests a second with a mean latency of 120 ms, of which 90 ms is waiting on the database. Each request holds a database connection for that 90 ms.

- How many requests are in flight on average?
 - How many database connections does the fleet need at minimum?
 - The database allows 200 connections. What are your options?
-

SOLUTION

- $900 \times 0.12 = 108$ requests in flight.
- $900 \times 0.09 = 81$ connections on average, so plan for perhaps twice that at peak: the mean is not the peak.
- You are close to the ceiling, and a burst or a slow query will hit it. Options, roughly in order: cut the 90 ms, since every millisecond saved frees connections; put a connection pooler such as PgBouncer in front, so hundreds of application connections share tens of database ones; cap each instance's pool so the fleet total is bounded; move reads to a replica. Raising the database's limit is last, because each connection costs memory there.

EXERCISE 3.2 The dashboard disagrees with the users

Your dashboard shows p99 latency at 120 ms, steady all week. Support is forwarding complaints about requests taking many seconds. Give three ways the dashboard could be right and useless at the same time.

SOLUTION

- **It averages percentiles**, either across instances or across five-minute windows, which hides both sick instances and short spikes. Compute percentiles from pooled histograms instead.
- **It measures the wrong point.** Server-side timers start when the request is dequeued, so queueing, TLS, DNS and the network are invisible. Measure at the load balancer, and from real browsers.
- **It's the wrong population.** One endpoint, one customer, one region or one device class can be terrible while the overall p99 is fine: 1% of users having a 100% bad time doesn't move the aggregate. Break down by route and tenant, and watch the p99.9.
- **Failed requests are excluded.** Timeouts often aren't recorded at all, so the slowest requests are missing from the statistic entirely: the same omission the load generator makes.

EXERCISE 3.3 Plan for a bad Monday

A service peaks at 4,000 requests a second on Monday mornings, and one instance handles 500 a second at its knee. You run in two zones and want to survive losing one, with instances at 60% of the knee and 20% growth expected before the next review. How many instances, and what else would you check?

SOLUTION

Usable capacity per instance is $500 \times 0.6 = 300$ rps, so peak needs $4,000 \div 300 \approx 14$ instances. With two zones, all the traffic must fit in one, so double it to 28, then add 20% for growth: **34 instances.**

Two zones is expensive for exactly this reason: each zone must be able to carry everything. With three zones the multiplier is 1.5 rather than 2, so the same protection costs 25 instances instead of 34.

Then check what doesn't scale with instance count: database connections and CPU, cache capacity, queue throughput, third-party rate limits, and how long new instances take to become useful, since a Monday spike that arrives faster than your autoscaler needs pre-warmed capacity, not a policy.

CHECKLIST

- ☐ Do you know each service's knee, measured rather than guessed?
- ☐ Are percentiles computed from pooled histograms, never averaged?
- ☐ Does your load generator hold a fixed arrival rate?
- ☐ Is there headroom for losing a zone, and for growth until the next review?
- ☐ Have you checked the limits that don't autoscale: connections, quotas, licences?

GO DEEPER

TALK **How NOT to Measure Latency** Gil Tene · coordinated omission, explained by the person who named it

ESSAY **On Coordinated Omission** ScyllaDB

DOCS **k6** an open-loop load generator worth learning

BOOK **Systems Performance, 2nd edition** Brendan Gregg · the queueing and capacity chapters

PART II

Packaging and platform

Four chapters on getting your code onto machines you didn't hand-build: what a container really is, what an orchestrator does with it, where configuration and secrets belong, and the cloud primitives underneath all of it.

Containers

3 exercises

A container is a normal process with a different view of the filesystem and its own resource budget, not a small virtual machine. Its image is a build artefact, and it should be reproducible, minimal, pinned and traceable back to a commit.

What a container actually is

Three kernel features, combined. **Namespaces** give the process its own view of process IDs, network interfaces, mounts, hostname and users. **Cgroups** give it the budgets from chapter 1: memory, CPU, tasks, I/O. And an **image** provides the root filesystem it sees. The kernel is the host's, shared with every other container on the machine, which is why a container is a weaker isolation boundary than a virtual machine, and why a kernel-level limit or bug affects everything on the box at once.

Two consequences bite immediately. Your process usually runs as **PID 1**, which on Linux has no default signal handlers: if your program doesn't handle SIGTERM itself, it simply ignores it and gets killed at the end of the grace period, exactly as in chapter 1. PID 1 is also expected to reap orphaned child processes, which most application runtimes don't do, so a container that spawns subprocesses accumulates zombies unless you add a tiny init such as `tini`, or run the container with `--init`.

Layers, and why build order matters

Each instruction in a Dockerfile produces a layer, and layers are cached: a build reuses every layer up to the first one whose inputs changed, then rebuilds the rest. That single rule dictates the order of a good Dockerfile: things that rarely change first, things that change every commit last.

form, so the process is PID 1 and receives signals rather than being wrapped in a shell that swallows them; and a `.dockerignore` keeps `.git`, local virtual environments and test fixtures out of the build context.

Build-time attestations are worth turning on: `docker buildx build --sbom=true --provenance=true` attaches a software bill of materials and a record of how the image was built, which is what lets you answer "is this image affected?" during the next dependency emergency without guessing. Scan images in the pipeline, and rebuild regularly even when your code hasn't changed, because the base image's vulnerabilities accumulate whether or not you deploy.

Make the rules a check

Habits that depend on everyone remembering them don't survive a busy week. Most of the advice above can be enforced in a few dozen lines, run in CI on every pull request.

EXAMPLE `dockerfile_rules.py` · run in CI

```
"""Fail the build on Dockerfile habits that hurt in production."""
import re
import sys

SECRETISH = re.compile(r"(SECRET|TOKEN|PASSWORD|PASSWD|API_?KEY)\s*=", re.I)

def instructions(text):
    """(line number, instruction, argument), with continuations joined."""
    out, buf, start = [], "", 0
    for n, raw in enumerate(text.splitlines(), 1):
        line = raw.split("#")[0].rstrip() if not raw.lstrip().startswith("#") else ""
        if not line.strip():
            continue
        if not buf:
            start = n
        buf += line.rstrip("\\")
        if line.endswith("\\"):
            continue
        verb, _, arg = buf.strip().partition(" ")
        out.append((start, verb.upper(), arg.strip()))
        buf = ""
    return out

def check(text):
    ins = instructions(text)
    froms = [(n, a) for n, v, a in ins if v == "FROM"]
    last_from = froms[-1][0] if froms else 0
    problems = []
    for n, image in froms:
        if "@sha256:" not in image:
            problems.append(f"{n}: pin the base image by digest, not by tag: {image.split()[0]}")
    users = [a for n, v, a in ins if v == "USER" and n > last_from]
    if not users or users[-1] in ("root", "0"):
        problems.append(f"{last_from}: the final stage runs as root; add a USER")
    for n, v, a in ins:
        if v in ("ENTRYPOINT", "CMD") and not a.startswith("["):
            problems.append(f"{n}: use the exec form of {v} so the process receives SIGTERM")
```

```

if v == "ADD" and not a.startswith(("http://", "https://")):
    problems.append(f"{n}: use COPY unless you need ADD's URL or archive handling")
if v in ("ENV", "ARG") and SECRETISH.search(a):
    problems.append(f"{n}: secrets do not belong in an image layer")
if v == "RUN" and "apt-get install" in a and "rm -rf /var/lib/apt/lists" not in a:
    problems.append(f"{n}: clean the package lists in the same RUN, or they stay in the
        layer")
copies = [n for n, v, a in ins if v == "COPY" and a.split()[0] in (".", "/")]
installs = [n for n, v, a in ins if v == "RUN" and re.search(r"\b(pip|npm|poetry|uv|go
    mod|bundle)\b", a)]
if copies and installs and min(copies) < max(installs):
    problems.append(f"{min(copies)}: copy dependency manifests before the source, or every
        "
                    "code change rebuilds the dependency layer")
return problems

if __name__ == "__main__":
    found = [p for path in sys.argv[1:] for p in check(open(path).read())]
    print("\n".join(found) or "Dockerfile looks fine")
    sys.exit(1 if found else 0)

```

Running them safely

- **Read-only root filesystem**, with a writable volume for the few paths that need it.
- **Drop all capabilities** and never run privileged. If you need to bind a low port, map it outside instead.
- **Set memory and CPU budgets** for every container, for the reasons in chapters 1 and 2.
- **Rootless where you can**. Rootless runtimes map the container's root to an unprivileged host user, so an escape lands nowhere useful.
- **Don't patch running containers**. An image you edited in production no longer matches anything you scanned, signed or can rebuild.

COMMON TRAPS

Copying the whole context first, so every commit reinstalls every dependency.

:latest in production, which makes "the same image" unknowable.

Secrets passed as build arguments, which stay in the image history.

Shell-form entryptoints, so the application never sees SIGTERM.

Running as root because the image does by default.

Fat images that pull slowly, which shows up as slow scale-ups exactly when you need speed.

EXERCISE 4.1 Fix the Dockerfile

Find the problems, then rewrite it.

EXERCISE Dockerfile

```
FROM python:latest
ENV DATABASE_PASSWORD=hunter2
ADD . /app
WORKDIR /app
RUN apt-get update && apt-get install -y build-essential
RUN pip install -r requirements.txt
CMD python -m src.server
```

SOLUTION

Running the checker above prints six complaints, one per line:

SOLUTION output

```
1: pin the base image by digest, not by tag: python:latest
1: the final stage runs as root; add a USER
2: secrets do not belong in an image layer
3: use COPY unless you need ADD's URL or archive handling
5: clean the package lists in the same RUN, or they stay in the layer
7: use the exec form of CMD so the process receives SIGTERM
```

Beyond those: there is no multi-stage split, so `build-essential` and every build artefact ship to production; the password is in the image history even if a later layer unsets it; and the build has no `.dockerignore`, so the context includes `.git` and any local environment files. The rewrite is the example earlier in this chapter.

EXERCISE 4.2 The image that got bigger

A team adds `RUN rm -rf /root/.cache /usr/share/doc` as the last line of their Dockerfile to shrink the image. The image gets slightly larger. Explain, and give two ways to actually make it smaller.

SOLUTION

Layers are stacked, not edited. Deleting a file in a later layer records a deletion on top; the bytes stay in the earlier layer, and the extra layer adds metadata. Anything ever written into an image is still in it, which is also why a secret removed in a later step is still recoverable.

To really shrink it: delete in the **same** `RUN` that created the files, so the layer is written without them; or use a **multi-stage build** and copy only the artefacts you need into a clean final stage, which is the reliable answer. Choosing a smaller base, and using `--no-install-recommends` and `--no-cache-dir`, helps too.

EXERCISE 4.3 Which image is running?

An incident is traced to a bug fixed three weeks ago. Production is running `api:v2.4`, and so is staging, where the bug doesn't reproduce. How is that possible, and what would have prevented it?

SOLUTION

A tag is a movable pointer, not an identity. Someone rebuilt and pushed `v2.4` after the fix, so staging pulled the new image while production still runs the old one it pulled weeks ago: two different images, one name. A rebuild of the same commit can also differ because a base image or an unpinned dependency moved underneath it.

Prevention: deploy by **digest**, not by tag; make tags immutable in the registry; record the image digest and the source commit in the deployment and in an annotation on the running workload, so "what is running" is answerable in one command; and keep provenance attestations so a digest can be traced back to the build that produced it.

CHECKLIST

- ☐ Are base images pinned by digest, and rebuilt on a schedule?
- ☐ Does the final image contain only what runs: no toolchain, no package manager, no secrets?
- ☐ Does the container run as a non-root user with a read-only root filesystem?
- ☐ Does the entrypoint use `exec` form, and does the process handle `SIGTERM`?
- ☐ Can you get from a running container back to the commit that built it?

GO DEEPER

DOCS Dockerfile best practices Docker

DOCS Distroless images Google Container Tools

REFERENCE SLSA levels for build provenance

Orchestration with Kubernetes

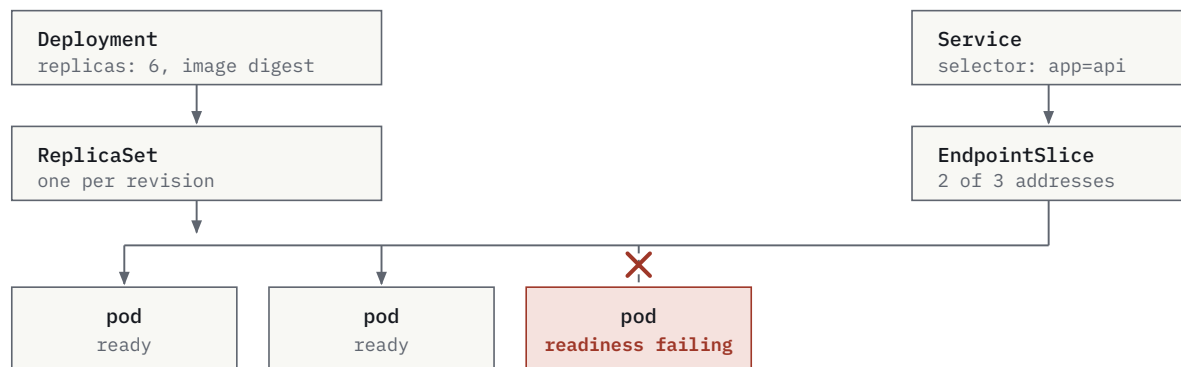
3 exercises

Kubernetes is a set of control loops: you declare what should be true, and controllers keep making it so. Most of what you need is one Deployment, one Service, honest resource requests and probes that mean what they say.

The model

Every object you create is a record in the API server, backed by etcd. Controllers watch those records and work to make reality match: the Deployment controller creates a ReplicaSet per revision, the ReplicaSet controller creates pods, the scheduler assigns each pod to a node, and the kubelet on that node starts containers and reports back. Nothing is a command; everything is a statement of intent that something else reconciles, continuously. That is why deleting a pod doesn't reduce your capacity for long, and why a bad declaration recreates itself no matter how many times you delete the symptom.

Objects find each other by **labels**, never by name. A Service sends traffic to whichever pods carry the labels in its selector and pass their readiness probe.



all three carry app= invoicing-api; only ready pods get traffic

Nothing refers to anything by name. The Deployment owns pods that match its selector; the Service routes to pods that match its own selector and are ready. A pod failing readiness is removed from the EndpointSlice and receives no traffic, without being restarted.

A Deployment worth copying

EXAMPLE deployment.yaml

```
apiVersion: apps/v1
```

[illegible]

```

    volumes:
      - name: tmp
        emptyDir: {}
  ---
  apiVersion: policy/v1
  kind: PodDisruptionBudget
  metadata:
    name: invoicing-api
  spec:
    maxUnavailable: 1                                # node drains take one pod at a time, not all six
    selector:
      matchLabels: { app: invoicing-api }

```

Requests, limits and what the kernel does with them

A **request** is a reservation: the scheduler only places a pod on a node with that much unclaimed CPU and memory, and it decides which pods get evicted first when a node runs short. A **limit** is a ceiling enforced by the kernel, and the two resources behave completely differently at that ceiling. Exceeding a memory limit gets the container killed (chapter 1); reaching a CPU limit gets it throttled inside every 100 ms period, which shows up purely as latency (chapter 2).

That asymmetry drives the usual advice: **set memory requests equal to memory limits**, so the pod is predictable and gets the highest quality-of-service class, and **be very careful with CPU limits**, which throttle even when the node is idle. Set CPU requests properly instead, and if you do set a limit, leave real headroom above the request. Since version 1.35, Kubernetes can resize a running pod's CPU and memory in place, without restarting it, which finally makes vertical autoscaling usable for long-lived workloads.

Probes: three questions, not one

- **Startup:** "has it finished booting?" Until it passes, the other probes are suspended, so a slow start isn't mistaken for a failure.
- **Readiness:** "should it get traffic now?" Failing takes the pod out of the Service and leaves it alone. It's allowed to consider dependencies.
- **Liveness:** "is it broken beyond recovery?" Failing restarts the container, so it must check only the process itself, never a dependency. A liveness probe that checks the database restarts every pod in the fleet the moment the database is slow, which is how a database hiccup becomes an outage.

Around shutdown, one race matters. When a pod is deleted, the kubelet sends SIGTERM and the endpoint controller removes it from the EndpointSlice at the same time, and those propagate independently. A short `preStop` sleep, as in the manifest, gives routing time to catch up so no request is sent to a process that has already begun shutting down.

Keeping the fleet spread and available

A **PodDisruptionBudget** constrains voluntary disruptions, such as node drains during a cluster upgrade, so the platform can't take all your pods at once. **Topology spread constraints** keep replicas across zones and nodes, so a lost zone doesn't take a whole service (*The Engineer's Fleet*, chapter 15). Neither is on by default, and both matter most on the day someone upgrades the cluster.

For scaling, three controllers do different jobs: the **HorizontalPodAutoscaler** adds pods based on a metric, **KEDA** does the same from queue depth or external events, and a node autoscaler such as **Karpenter** or the cluster autoscaler adds machines when pods won't fit. A common mistake is to scale pods on CPU when the real signal is queue depth or concurrency.

Getting traffic in

Inside the cluster, a Service gives a stable virtual address and DNS name. From outside, you need an ingress controller or a cloud load balancer. This is one area where 2026 forced everyone's hand: `ingress-nginx`, by far the most widely deployed ingress controller, was retired by the Kubernetes project and reached end of life on 24 March 2026, after repeated critical vulnerabilities and a maintainer shortage. The Ingress API itself remains, and other controllers are unaffected, but new work belongs on the **Gateway API**, which replaces a pile of annotations with typed resources for routes, and has a supported migration tool. If you still run `ingress-nginx`, migrating is not a 2027 project.

When something is wrong

The order that answers most questions: `kubectl describe pod`, whose Events section explains scheduling failures, image pull errors, probe failures and OOM kills; `kubectl logs --previous` for the container that just died; `kubectl get events --sort-by=.lastTimestamp` for the cluster's own story; and `kubectl debug` to attach an ephemeral container with a shell to a distroless pod that has none. `CrashLoopBackOff` is not an error, it's the platform backing off between restarts; the error is in the previous container's logs.

COMMON TRAPS

Liveness probes that check dependencies, turning a slow database into a fleet-wide restart storm.

CPU limits set equal to requests, which throttles bursts on an idle node.

No PodDisruptionBudget, so a cluster upgrade drains every replica at once.

Deploying by tag, so a rollout is not reproducible (chapter 4).

Secrets as plain Kubernetes Secrets with no encryption at rest and no rotation (chapter 6).

Running a database on the cluster without an operator, a backup plan and someone who knows it.

EXERCISES

EXERCISE 5.1 Why did every pod restart?

The database had 90 seconds of high latency. During that window every pod of every service restarted several times, turning a slow minute into a twenty-minute outage. The liveness probe is `GET /healthz`, which checks the database connection and returns 500 if a query takes more than a second. What happened, and what should each probe check?

SOLUTION

The liveness probe made every pod declare itself dead the moment the database slowed. Kubernetes restarted them, all at once, which dropped in-flight requests, emptied every connection pool and cache, and then hit the struggling database with a reconnection storm from a whole fleet of cold pods. The restart made recovery strictly harder.

Liveness should test only what a restart could fix: the process is running and its event loop isn't wedged. A static handler that returns 200 is usually right. **Readiness** may consider dependencies, because its consequence is "send me no traffic for now", which is what you want when a dependency is down, though even there, taking every pod out of service at once turns a degraded system into a dead one, so prefer serving degraded responses. **Startup** covers slow boots. And set `failureThreshold` so a single slow response doesn't restart anything.

EXERCISE 5.2 Pending pods

Six replicas are requested; four are Running and two are Pending.

`kubectl describe` on a pending pod shows `0/9 nodes are available: 6 Insufficient cpu, 3 node(s) didn't match pod topology spread constraints`. Explain each part and give three ways forward.

SOLUTION

Six nodes have less unreserved CPU than the pod **requests**, which is about reservations, not actual usage: a cluster can be 30% busy and still full. The other three nodes have room, but placing a pod there would break the zone spread constraint, and `DoNotSchedule` means the scheduler refuses rather than unbalances.

Ways forward: lower the CPU request if it was padded, which is the most common real fix and usually reveals that requests were set by copy-paste; add nodes, or let a node autoscaler add them in the zones that are short; or relax the constraint to `ScheduleAnyway`, accepting imbalance in exchange for capacity. Check for pods that were sized "just to be safe": surveys of real clusters keep finding utilisation against requests down around 40%.

EXERCISE 5.3 A rollout that eats capacity

A service runs 10 replicas with `maxSurge: 25%` and `maxUnavailable: 25%`. Each pod takes 40 seconds to become ready. During deploys, error rates rise. What's happening, and what settings would you change?

SOLUTION

With `maxUnavailable: 25%`, Kubernetes may take two or three pods out before their replacements are ready, so the service runs at 70–75% capacity for the length of the rollout. If you sized the fleet with little headroom (chapter 3), the remaining pods pass the knee of the latency curve and start failing.

Set `maxUnavailable: 0` and `maxSurge` to one or two pods, so capacity only ever goes up during a rollout; make sure the readiness probe reports ready only when the pod can actually serve, so surge capacity is real; and keep `progressDeadlineSeconds` long enough for the 40-second start but short enough to fail a stuck rollout. Then check `kubectl rollout status` in the pipeline and roll back on failure (chapter 10).

CHECKLIST

- ☐ Does every workload set resource requests, and memory limits equal to requests?
- ☐ Does the liveness probe avoid every dependency?
- ☐ Is there a `PodDisruptionBudget` and a spread across zones?
- ☐ Do rollouts add capacity before removing it, with a progress deadline?
- ☐ Are you off `ingress-nginx`, or is there a dated plan to be?

GO DEEPER

DOCS

Pod lifecycle, probes and termination [Kubernetes](#)

NEWS

Ingress NGINX retirement Kubernetes blog

DOCS

Gateway API the replacement for Ingress

BOOK

Kubernetes Up & Running, 3rd edition Burns, Beda, Hightower & Evenson

Configuration and secrets

3 exercises

Configuration breaks production more often than code does, because it changes faster, is reviewed less and is tested least. Treat it like code: typed, validated at start-up, versioned, and rolled out gradually.

The largest outages are configuration changes

On 18 November 2025 a permissions change in one of Cloudflare's databases caused a generated file used by its bot-management system to double in size. The file was loaded by the proxy that fronts the network, which had a fixed limit on how large it could be, and the proxy began returning 5xx for a large part of the internet. It took hours to resolve. No code was deployed; a piece of generated data changed.

That shape recurs everywhere: a value nobody reviewed, applied everywhere at once, hitting a limit nobody remembered. Configuration deserves the same care as a deploy, which means review, staged rollout, limits with sane behaviour when exceeded, and a way back.

Three kinds of configuration

- **Baked in at build time:** compiled defaults and the code itself. Changing it means a new artefact, so it's the slowest and safest.
- **Read at start-up:** environment variables and mounted files. Changing it means a restart, so it's covered by your normal rollout (chapter 10).
- **Changed at run time:** feature flags, dynamic config, rule and model files. This is the fastest, the most useful, and the most dangerous, because it bypasses every safeguard the deployment pipeline gives you unless you rebuild those safeguards around it.

Put settings in the slowest category that still meets the need, and give anything in the third category a staged rollout, a verification step and an obvious rollback.

Validate at start-up, and say everything that's wrong

The usual pattern, `int(os.environ.get("WORKERS", "4"))` scattered through the code, fails in the worst possible way: a typo becomes a default, a bad value fails at the first request rather than at boot, and nobody can list what the service actually reads. Parse the whole configuration once, at start-up, and refuse to run if it's wrong.

EXAMPLE `config.py`

```

import os
from dataclasses import MISSING, dataclass, fields

class ConfigError(Exception):
    """Raised once, listing everything that is wrong."""

def _parse(kind, raw):
    if kind is bool:
        if raw.lower() not in ("true", "false", "1", "0"):
            raise ValueError("expected true or false")
        return raw.lower() in ("true", "1")
    return kind(raw)

@dataclass(frozen=True)
class Config:
    database_url: str # no default: it must be supplied
    port: int = 8080
    workers: int = 4
    request_timeout_s: float = 5.0
    new_pdf_renderer: bool = False

    def redacted(self) -> dict:
        out = {}
        for f in fields(self):
            value = getattr(self, f.name)
            out[f.name] = "***" if any(w in f.name for w in ("url", "password", "token",
                                                             "key")) else value
        return out

CHECKS = {
    "port": (lambda v: 1 <= v <= 65535, "must be between 1 and 65535"),
    "workers": (lambda v: v >= 1, "must be at least 1"),
    "request_timeout_s": (lambda v: v > 0, "must be positive"),
}

def load(env=None) -> Config:
    """Read the whole configuration, then report every problem at once and stop."""
    env = os.environ if env is None else env
    values, problems = {}, []
    for f in fields(Config):
        raw = env.get(f.name.upper())
        if raw is None or raw == "":
            if f.default is MISSING:
                problems.append(f"{f.name.upper()} is required")
            continue
        try:
            value = _parse(f.type, raw)
        except ValueError as exc:
            problems.append(f"{f.name.upper()}={raw!r}: {exc}")
            continue
        ok, why = CHECKS.get(f.name, (lambda v: True, ""))
        if not ok(value):
            problems.append(f"{f.name.upper()}={value}: {why}")
            continue
        values[f.name] = value
    if problems:
        raise ConfigError("bad configuration:\n " + "\n ".join(problems))
    return Config(**values)

```

A bad environment now produces one clear failure before the process serves anything:

EXAMPLE output

```
ConfigError: bad configuration:
  DATABASE_URL is required
  PORT=70000: must be between 1 and 65535
  WORKERS=0: must be at least 1
  REQUEST_TIMEOUT_S=-1.0: must be positive
  NEW_PDF_RENDERER='yes': expected true or false
```

Note that it reports *every* problem, not the first: fixing configuration one error per deploy is a miserable way to spend an afternoon. Log the effective configuration at start-up with secrets redacted, as `redacted()` does, so an incident never turns into an argument about which value was in force.

One artefact, many environments

The same image runs in staging and production, with different configuration supplied to it; environments differ in values, never in build. Keep the values in version control next to the code that reads them, apply them through the pipeline rather than by hand, and keep the difference between environments small enough to print. A staging environment configured differently from production tests something other than production.

Secrets

Everything above applies to secrets, plus a set of rules of its own. Stage 13 covers the cryptography and the threat model; these are the operational parts.

- **Never in the image, the repository, or a unit file.** Images are shared, repositories are cloned, and layers keep deleted files (chapter 4).
- **Prefer short-lived credentials over stored ones.** Workload identity lets a running service exchange its platform identity for a token that expires in minutes, and lets a pipeline authenticate to a cloud without a long-lived key in a CI variable. A secret that doesn't exist can't leak.
- **Where you must store one, use a secrets manager** with access control, audit logs and rotation, and pull it at start-up or mount it as a file. Files are usually better than environment variables, because environments are inherited by child processes and dumped by crash reporters.
- **Kubernetes Secrets are base64, not encryption.** Enable encryption at rest, restrict access with RBAC, and consider an operator that syncs from a real secrets manager.
- **Rotate on a schedule, not after an incident,** and make sure rotation is a routine that works: a credential nobody can rotate is a credential you can never revoke.

COMMON TRAPS

Silent defaults. A misspelled variable name becomes "the default was fine in staging".

Configuration read at use time, so a bad value fails on a Saturday under load rather than at boot.

Secrets in environment variables that get logged by a crash reporter or a debug endpoint.

Applying config globally and instantly, with no staging and no limit on its size or effect.

Manual changes on one machine, which survive until that machine is replaced, then vanish.

Environments that differ in more than values, so staging proves nothing.

EXERCISES

EXERCISE 6.1 Four bugs in six lines

Find the problems, then rewrite it.

EXERCISE settings.py

```
import os

DEBUG = os.environ.get("DEBUG", "false") == "true"
TIMEOUT = int(os.environ.get("TIMEOUT", 5))
DATABASE_URL = os.environ.get("DATABASE_URL", "postgres://localhost/dev")
STRIPE_KEY = os.environ.get("STRIPE_KEY", "sk_test_abc123")
print(f"starting with {os.environ}")
```

SOLUTION

- **DATABASE_URL** is misspelled, so production quietly connects to a local development database, or fails in a way that looks like a network problem. A required setting with no default would have failed at boot.
- **A default secret.** The Stripe key falls back to a test key, so in production you get silent no-ops instead of an error. Secrets must be required.
- **A development default for the database,** which is the same bug in a different suit: defaults should be safe, not convenient.

- **Printing the whole environment** writes every secret to the logs, where they live for as long as your retention does, in a system with different access control.
- **No validation:** `TIMEOUT=0` or a negative value is accepted, and `DEBUG=True` (capital T) silently means false.

EXERCISE 6.2 Roll out a risky value

You maintain a rules file that every instance of the API loads at start-up and reloads every five minutes. A new release of the rules is generated by a nightly job from a database query. Design the rollout so that the failure mode from the Cloudflare outage can't happen to you.

SOLUTION

- **Validate before publishing:** the generating job checks the file against a schema and against sanity bounds, such as size, row count and the change since the last version, and refuses to publish an outlier.
- **Validate on load, too,** and on failure keep serving the previous version rather than crashing or returning errors. Loading bad data should degrade the feature, not the service.
- **Version the file and roll forward gradually:** a canary group of instances picks up the new version first, and the rest follow only if their error rates stay flat. That alone limits a bad file to a small blast radius.
- **Keep the previous version available** and make switching back a single action, not a regeneration.
- **Alert on the file's own properties,** not just on the service: size, age, version and load failures.

EXERCISE 6.3 Where does each setting live?

For each, choose build time, start-up, run time, or a secrets manager, and say why.

- a. The database password.
- b. The percentage of users who see a new checkout flow.
- c. The log level.
- d. The timeout for calls to the PDF service.

e. The list of supported currencies.

SOLUTION

- a. **Secrets manager**, fetched at start-up or mounted as a file, rotated on a schedule.
- b. **Run time**, a feature flag: that's the point of it, and it needs to change in seconds without a deploy.
- c. **Run time**, ideally changeable per service without a restart, because you want more detail exactly when something is wrong.
- d. **Start-up**, with a compiled default. It should change rarely and deliberately, and a restart to change it is fine.
- e. **Build time**, because adding a currency means code: formatting, rounding, tax rules and tests. Making it "configurable" invites a config change that no code supports.

CHECKLIST

- ☐ Does the service refuse to start on bad configuration, reporting every problem at once?
- ☐ Can you see the effective configuration of a running instance, with secrets redacted?
- ☐ Do configuration and data files roll out gradually, with validation and a fallback?
- ☐ Are there any long-lived credentials left that workload identity could replace?
- ☐ Is every environment's difference from production small enough to list?

GO DEEPER

REPORT [Cloudflare outage on November 18, 2025 the clearest public postmortem of a configuration failure](#)

REFERENCE [The Twelve-Factor App: config still the right default](#)

DOCS [Good practices for Kubernetes Secrets](#) [Kubernetes](#)

The cloud underneath

3 exercises

One cloud known properly beats three known vaguely. The primitives are the same everywhere: compute you should expect to lose, storage you can't, a network you have to design, and an identity system that decides what every piece of it may do.

Compute you can lose

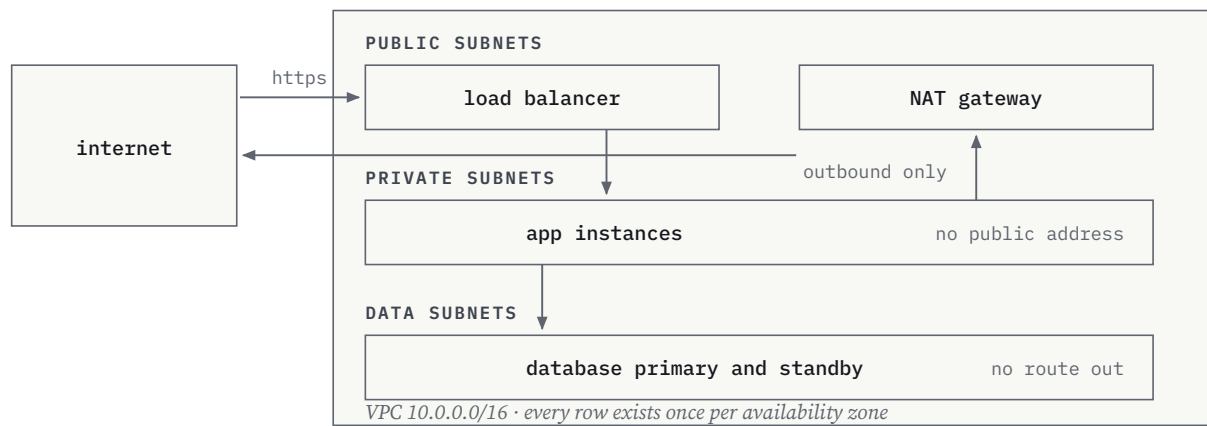
The first mental shift is that instances are disposable. They get rebooted for host maintenance, retired with two minutes' notice, or reclaimed in seconds if they're spot capacity. Anything that must survive belongs in storage, not on a disk you attached by hand, and anything that takes an hour to rebuild by hand is an outage waiting for its moment. Build machines from images and code (chapter 8), spread them across zones, and let the platform replace them.

Spot or preemptible instances are typically a fraction of the on-demand price and suit anything that can be interrupted: batch jobs, queue workers, CI runners, stateless capacity above a baseline. Mixing a committed baseline with spot for the peak is where most of the compute savings in chapter 19 come from.

Storage you can't

- **Block storage** is a virtual disk with a size, an IOPS allowance and a throughput allowance, sometimes with burst credits that run out. Very many "the database got slow after three weeks" mysteries are exhausted burst credits (chapter 2).
- **Object storage** is the default home for anything large or numerous: uploads, generated PDFs, backups, logs. It's cheap, effectively unlimited, and durable across zones, with lifecycle rules that move old objects to colder, cheaper tiers.
- **Managed databases** give you backups, patching, replicas and failover. You still own the schema, the queries, the connection limits, major-version upgrades, and the knowledge that a failover takes tens of seconds during which writes fail.

A network you have to design



Only two things belong in a public subnet: what takes traffic in, and what lets traffic out. Everything that serves users sits in private subnets with no address of its own, and the database has no route to the internet at all. That NAT gateway is also charged per gigabyte, which makes it a regular surprise on the bill.

Within that shape, the parts to get right are: security groups as the real firewall, written as "this service may talk to that service" rather than to address ranges; private endpoints so traffic to managed services doesn't leave the network; and an awareness that **cross-zone traffic is charged**, which quietly makes chatty service-to-service calls a line item as well as a latency problem.

Identity is the control plane

Everything in a cloud is an API call, and IAM decides which calls succeed. Two rules carry most of the weight. **Grant the narrowest permission that works**, scoped to specific resources and prefixes. And **stop storing long-lived credentials**: give workloads an identity the platform issues and rotates, and let pipelines exchange a short-lived OIDC token for a role rather than holding an access key in a CI variable.

EXAMPLE app-policy.json · what the service may do

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ReadWriteThisTenantsPdfs",
      "Effect": "Allow",
      "Action": ["s3:GetObject", "s3:PutObject"],
      "Resource": "arn:aws:s3:::invoicing-pdfs-prod/tenants/*"
    },
    {
      "Sid": "ListOnlyUnderThatPrefix",
      "Effect": "Allow",
      "Action": "s3:ListBucket",
      "Resource": "arn:aws:s3:::invoicing-pdfs-prod",
      "Condition": { "StringLike": { "s3:prefix": "tenants/*" } }
    }
  ]
}

```

```
]
}
```

EXAMPLE `deploy-role-trust.json` · who may become this role

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": { "Federated": "arn:aws:iam::123456789012:oidc-
        provider/token.actions.githubusercontent.com" },
      "Action": "sts:AssumeRoleWithWebIdentity",
      "Condition": {
        "StringEquals": {
          "token.actions.githubusercontent.com:aud": "sts.amazonaws.com",
          "token.actions.githubusercontent.com:sub": "repo:acme/invoicing:ref:refs/heads/main"
        }
      }
    }
  ]
}
```

The condition in the second policy is the whole security of it. Without the `sub` condition, any repository on that provider could assume the role; with a wildcard such as `repo:acme/*`, any repository in your organisation could, including one a contractor forked. Exact matches, then.

Quotas, regions and the limits you can't see

Every cloud enforces quotas you didn't choose: instances per region, IP addresses, API calls per second, open connections on a load balancer. They're invisible until a scale-up stops at a round number. Find the ones that apply to your growth, ask for increases before you need them, and alert on usage against quota the way you alert on disk.

On geography: a **zone** is a failure domain within a region, and running in three of them is the standard defence (*The Engineer's Fleet*, chapter 15). A **region** is a much bigger step, involving data residency, replication and latency, and multi-region is a project, not a checkbox. Multi-cloud is another order of magnitude again, and usually buys less resilience than spending the same effort on backups you have actually restored (chapter 18).

COMMON TRAPS

Long-lived access keys in CI variables, developer laptops and application config.

Public subnets by default, so databases and workers have addresses on the internet.

Security groups that allow 0.0.0.0/0 "temporarily" on port 22 or 5432.

One account for everything, so a mistake in staging can reach production.

Ignoring quotas until an autoscaler hits one during a traffic spike.

Treating a managed database as fully managed: failovers, upgrades and connection limits are still yours.

EXERCISES

EXERCISE 7.1 Tighten the policy

This is attached to the role the invoicing API assumes. What can go wrong, and how would you narrow it?

EXERCISE app-policy.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    { "Effect": "Allow", "Action": "s3:*", "Resource": "*" },
    { "Effect": "Allow", "Action": ["iam:PassRole", "kms:Decrypt"], "Resource": "*" }
  ]
}
```

SOLUTION

The first statement allows every S3 action on every bucket in the account: reading other services' data, deleting backups, changing bucket policies. The second is worse than it looks. `iam:PassRole` on `*` lets the service hand any role, including an administrative one, to something it creates, which is a standard path from a small compromise to a whole account. `kms:Decrypt` on `*` lets it decrypt anything the account can.

Narrow it: name the exact actions, scope resources to one bucket and one prefix, scope `kms:Decrypt` to the single key that encrypts that bucket, and remove `iam:PassRole` unless you can name the one role it may pass. Then let a machine enforce it: a policy checker in CI, or your cloud's own policy analyser, which will tell you that a statement grants more than it needs before anyone merges it.

EXERCISE 7.2 Why did the bill triple?

Traffic grew 20% last month, but the network line on the bill tripled. The architecture is the one in the figure, with services calling each other freely across zones and fetching a 4 MB reference file from an external API on every request. Where is the money going, and what would you change?

SOLUTION

- **NAT gateway processing.** Every outbound request to that external API passes through the NAT gateway, which charges per gigabyte in addition to the data transfer. A 4 MB fetch per request is an enormous amount of NAT traffic; cache it, or fetch it once per instance per hour.
- **Cross-zone traffic** between services that call each other constantly. Prefer zone-local routing where the platform supports it, and treat a chatty pair of services as a design smell (*The Engineer's Fleet*, chapter 17).
- **Traffic to managed services over the public path** rather than a private endpoint, which is both slower and billable.

The general lesson for chapter 19: the bill is a measurement of your architecture, and network charges in particular measure how chatty it is.

EXERCISE 7.3 Design the failure domains

Your service must survive the loss of one availability zone with no data loss and a few minutes of degradation. Sketch what runs where, and state what you are explicitly not protecting against.

SOLUTION

- **Compute** in three zones, each sized so two can carry the peak (chapter 3), with the capacity already provisioned rather than requested during the incident.
- **Database** with a synchronous standby in a second zone, so committed writes survive, plus automated failover. Accept and test the failover window: writes fail for tens of seconds.
- **Object storage** is already replicated across zones; backups go to a separate account or project, so a mistake in this one can't delete them (chapter 18).

- **Not protected:** the loss of the whole region, a bad deploy, a bad configuration change, deletion of data by your own code, or the failure of a third party you depend on. Those need different mechanisms, and saying so out loud is the point of the exercise.

CHECKLIST

- ☐ Could you lose any single instance right now without anyone noticing?
- ☐ Is anything but the load balancer and NAT gateway in a public subnet?
- ☐ Are there any long-lived cloud credentials left, anywhere?
- ☐ Do you know which quotas you'll hit at three times today's traffic?
- ☐ Are production and non-production in separate accounts or projects?

GO DEEPER

REFERENCE [AWS Well-Architected Framework](#) the pillars generalise to any cloud

ESSAY [Static stability using Availability Zones](#) [Amazon Builders' Library](#)

DOCS [OpenID Connect for cloud credentials in CI](#) [GitHub](#)

PART III

Delivery

Three chapters on the path from a commit to production: infrastructure declared in code, a pipeline that answers whether a change is safe, and a deployment that can be undone.

Infrastructure as code

3 exercises

Infrastructure that exists because somebody clicked is infrastructure nobody can rebuild, review or explain. Declare it in code, review the plan as carefully as a diff, and make the pipeline the only path to production.

What you get for the trouble

Writing infrastructure as code costs time up front and repays it in four ways. You can **rebuild**: a region, an environment or a deleted resource comes back from a repository rather than from memory. You can **review**: a change to a security group is a pull request with a diff, an author and a reason. You can **audit**: what production looks like is knowable without logging in. And your environments can actually **match**, which is what makes testing in staging mean anything.

The tools are declarative. You describe the state you want; the tool compares it with a record of what it built, computes a plan, and applies it. That record, the **state file**, is the only thing that ties your code to real resources, which makes it the most important file in the repository and the one you never edit by hand. Keep it in a remote backend with locking, so two people applying at once can't interleave, and encrypt it, because it contains resource details and sometimes secrets.

Terraform, OpenTofu, and the fork in the road

In 2023 HashiCorp relicensed Terraform under the Business Source License, and the community forked the last open version as **OpenTofu**, now under the Linux Foundation. In practice the two remain compatible: the same HCL, the same providers, the same state format, and most configurations move without edits. OpenTofu has shipped features Terraform hasn't, notably state encryption; Terraform has HashiCorp's platform around it, and IBM behind it since the 2025 acquisition. Either is a reasonable choice. What matters is that you pick one, keep your configuration portable, and don't spend a quarter on the question.

EXAMPLE storage/main.tf

```
terraform {
  required_version = ">= 1.11"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.60"
    }
  }
  backend "s3" {
```

```

    bucket      = "acme-tfstate-production"
    key         = "invoicing/storage.tfstate"
    region      = "eu-west-1"
    encrypt     = true
    use_lockfile = true # state locking in S3 itself, no separate table
  }
}

variable "environment" {
  type = string
  validation {
    condition     = contains(["staging", "production"], var.environment)
    error_message = "environment must be staging or production."
  }
}

resource "aws_s3_bucket" "pdfs" {
  bucket = "invoicing-pdfs-${var.environment}"

  # Renaming this resource in code must never delete the customers' invoices.
  lifecycle {
    prevent_destroy = true
  }
}

resource "aws_s3_bucket_versioning" "pdfs" {
  bucket = aws_s3_bucket.pdfs.id
  versioning_configuration {
    status = "Enabled" # a deletion by our own code is recoverable
  }
}

resource "aws_s3_bucket_public_access_block" "pdfs" {
  bucket                  = aws_s3_bucket.pdfs.id
  block_public_acls      = true
  block_public_policy    = true
  ignore_public_acls     = true
  restrict_public_buckets = true
}

resource "aws_s3_bucket_lifecycle_configuration" "pdfs" {
  bucket = aws_s3_bucket.pdfs.id

  rule {
    id      = "cold-storage-after-90-days"
    status = "Enabled"
    filter {
      prefix = "tenants/"
    }
    transition {
      days          = 90
      storage_class = "STANDARD_IA"
    }
  }
}

output "pdf_bucket" {
  value = aws_s3_bucket.pdfs.bucket
}

```

Four habits are visible there. The provider version is pinned, so a fresh `init` next month doesn't bring a new major version. The backend is encrypted and locked. The variable validates itself, so a typo fails at plan time rather than creating `invoicing-pdfs-prodction`.

And the bucket that holds customers' invoices carries `prevent_destroy`, because the most expensive infrastructure mistakes are deletions that the plan described accurately and nobody read.

The plan is the review

A plan is a diff against reality, and it deserves the attention you give code. Run it on every pull request, post it in the review, and apply only from the main branch after merge. Three things to look for every time:

- **Anything that says `forces replacement`** on a resource that holds state. Replacing a database or a volume means creating a new one and deleting the old one, with your data inside it.
- **Destroys you didn't intend.** Renaming a resource in code, or moving it between modules, reads as "destroy that, create this" unless you tell the tool otherwise with a `moved` block or a state move.
- **A plan that is much larger than your change**, which usually means drift: someone changed something by hand, and your apply is about to revert it, or the provider has changed its defaults.

Then automate the parts a tired reviewer will miss: policy-as-code checks in the pipeline that reject public buckets, unencrypted volumes, security groups open to the world, or any plan containing a destroy of a protected resource.

Drift, and where it comes from

Drift is the difference between what your code says and what exists. It appears because someone fixed something during an incident at 3am, which is legitimate, or because a console click was easier, which is not. Detect it with a scheduled plan that alerts when the diff is non-empty, then close the loop: either codify the change, or revert it. Drift that nobody reconciles slowly turns infrastructure as code back into documentation.

Structure, blast radius and the rest of the toolchain

- **Split state by environment and by component.** One state file for everything means one lock, one blast radius and a twenty-minute plan. Separate production from staging entirely, ideally in different accounts.
- **Modules for repeated shapes**, kept small and versioned. A module that takes thirty variables is a program with a bad interface.
- **Immutable where possible.** Replace machines rather than mutating them: bake an image, roll it out, delete the old one. Configuration management such as Ansible is for what you can't replace.

- **GitOps for clusters.** Argo CD or Flux continuously reconcile a Kubernetes cluster against a repository, which is the same idea applied to a system that already reconciles: the repository becomes the desired state, and drift is corrected rather than reported.
- **Secrets stay out.** Reference a secrets manager rather than committing values, and remember that anything the tool reads can land in state, which is why the state is encrypted and access to it is restricted (chapter 6).

COMMON TRAPS

Applying from a laptop, so the state, the credentials and the audit trail are personal.

Unpinned providers, so an unrelated change brings a new major version with it.

One giant state file whose plan nobody reads because it's always long.

Approving a plan you skimmed, which is how stateful resources get replaced.

Console changes during incidents that are never codified.

Secrets in `terraform.tfvars`, committed and forgotten.

EXERCISES

EXERCISE 8.1 Read the plan

This plan is waiting for your approval on a Friday afternoon. What do you do?

```
EXERCISE terraform plan
```

```
# aws_db_instance.primary must be replaced
-/+ resource "aws_db_instance" "primary" {
  ~ engine_version      = "15.5" -> "16.3" # forces replacement
  ~ id                  = "invoicing-prod" -> (known after apply)
  ~ endpoint            = "invoicing-prod.xxxx.rds.amazonaws.com" -> (known after
    apply)
  # (23 unchanged attributes hidden)
}

# aws_s3_bucket.uploads will be destroyed
- resource "aws_s3_bucket" "uploads" {
  - bucket = "invoicing-uploads-production" -> null
}

# module.network.aws_security_group_rule.db_ingress will be updated in-place
```

```
~ resource "aws_security_group_rule" "db_ingress" {
  ~ cidr_blocks = ["10.0.0.0/16"] -> ["0.0.0.0/0"]
}
```

Plan: 2 to add, 1 to change, 2 to destroy.

SOLUTION

Do not apply. Three separate problems, any one of which is an incident.

- **The database will be replaced, not upgraded:** a new instance is created and the current one destroyed, along with its data, and every connection string changes. Major version upgrades are done with the provider's in-place upgrade path and a maintenance window, after a verified snapshot (chapter 18), never by a plan that says `forces replacement`.
- **A bucket will be destroyed.** Almost certainly it was renamed or moved in code, and the fix is a `moved` block or a state move, not a delete and recreate. It should also have carried `prevent_destroy`.
- **The database's ingress rule is being opened to the whole internet.** That is either a mistake or an attack on your review process; either way a policy check should have failed the pipeline before a human ever saw it.

And, separately: this is the kind of change that goes out on a Tuesday morning, not on a Friday afternoon.

EXERCISE 8.2 The 3am fix

During an incident, an engineer doubles a database's IOPS in the cloud console and the incident ends. What should happen next, and what should the team change so this is normal rather than shameful?

SOLUTION

The console change was right: mitigation beats process at three in the morning (chapter 16). What must follow is reconciliation, the same week, while anyone still remembers: put the new value in code, run a plan, confirm the only diff is that change, and apply it. Otherwise the next apply silently reverts it, usually during another busy moment.

To make this routine: run a scheduled drift check that alerts on any non-empty plan, so the reversion is caught by a machine rather than by an outage; add "codify any emergency change" to the incident

checklist; and make the code path fast enough that codifying isn't a day's work.

EXERCISE 8.3 Split the state

One repository holds one state file describing the network, the cluster, three databases and eleven services, across staging and production. Plans take twenty minutes and everyone has learned to skim them. Propose a split, and say what each boundary buys.

SOLUTION

- **By environment first**, and ideally by account: a mistake in staging then cannot reach production, and the credentials for one don't work on the other.
- **By rate of change**: foundations that change monthly (network, cluster, databases) separate from application infrastructure that changes daily. Small, frequent plans stay readable.
- **By blast radius**: anything holding data gets its own state, its own permissions and `prevent_destroy`, so an application change can't produce a plan that touches it.

Pass values between them through published outputs or data sources rather than one shared state, and expect the split itself to be the fiddly part: moving resources between states needs care, but it's a one-off, and it's much cheaper than the plan nobody reads.

CHECKLIST

- ☐ Could you rebuild an environment from the repository, and have you tried?
- ☐ Is state remote, locked, encrypted and never edited by hand?
- ☐ Does every change go through a plan in a pull request and an apply from the main branch?
- ☐ Do policy checks reject public buckets, open security groups and destroys of protected resources?
- ☐ Would you find out this week if someone changed something by hand?

GO DEEPER

DOCS

Terraform: state what it is and why it matters

DOCS

OpenTofu documentation including state encryption

BOOK

Terraform: Up & Running, 3rd edition Yevgeniy Brikman · modules, state and team workflows

Continuous integration

3 exercises

A pipeline answers one question on every change: is this safe to ship? Its value comes from how fast and how honestly it answers, which is why speed and flakiness are engineering problems, not annoyances.

The shape of a pipeline

Cheap checks first, so feedback is fast, and one artefact built once and promoted everywhere afterwards (*The Engineer's Shipyard*, chapter 17). A workable order: format and lint, type check, unit tests, build the artefact, integration tests against real dependencies started as containers, security and dependency scans, publish the artefact with its provenance, then hand over to deployment (chapter 10). Everything before the artefact runs on every pull request; everything after runs on the main branch.

EXAMPLE `.github/workflows/ci.yml`

```
name: ci

on:
  pull_request:
  push:
    branches: [main]

permissions:
  contents: read          # the default token gets nothing else unless a job asks

concurrency:
  group: ci-${{ github.ref }}
  cancel-in-progress: true # a new push cancels the old run

jobs:
  test:
    runs-on: ubuntu-24.04
    timeout-minutes: 15    # a hung job fails instead of burning an hour
    services:
      postgres:             # a real database, not a mock
        image: postgres:17@sha256:aaaa...
        env: { POSTGRES_PASSWORD: test }
        options: --health-cmd pg_isready --health-interval 5s --health-retries 10
        ports: ["5432:5432"]
    steps:
      - uses: actions/checkout@08c6903 # v5.0.1 - third-party actions pinned by commit
      - uses: actions/setup-python@a26af6.. # v6.0.0
        with:
          python-version: "3.13"
          cache: pip
      - run: pip install -r requirements-dev.txt
      - run: ruff check .
      - run: mypy src
      - run: pytest -q --maxfail=1 --durations=10
    env:
```

```

    DATABASE_URL: postgres://postgres:test@localhost:5432/postgres

image:
  needs: test
  if: github.ref == 'refs/heads/main'
  runs-on: ubuntu-24.04
  timeout-minutes: 20
  permissions:
    contents: read
    id-token: write          # OIDC: a short-lived cloud token, no stored keys
  steps:
    - uses: actions/checkout@08c6903 # v5.0.1
    - uses: docker/setup-buildx-action@e468171 # v3.11.1
    - uses: aws-actions/configure-aws-credentials@f503a18 # v5.0.0
    with:
      role-to-assume: arn:aws:iam::123456789012:role/ci-push-images
      aws-region: eu-west-1
    - name: Build, attest and push one artefact
      run: |
        docker buildx build \
          --sbom=true --provenance=true \
          --cache-from type=registry,ref=$REGISTRY/invoicing/api:cache \
          --cache-to type=registry,ref=$REGISTRY/invoicing/api:cache,mode=max \
          --tag $REGISTRY/invoicing/api:${{ github.sha }} \
          --push .
  env:
    REGISTRY: 123456789012.dkr.ecr.eu-west-1.amazonaws.com

```

The security-relevant lines are easy to miss. `permissions: contents: read` at the top means the automatic token can do nothing else unless a job asks; the image job asks for `id-token: write` so it can exchange an OIDC token for a cloud role, which removes the long-lived access key that would otherwise sit in a CI variable forever (chapter 7). Third-party actions are pinned to a commit, not a tag, because a tag can be moved by whoever owns the repository, and an action runs with access to your build. Every job has a timeout, because a hung job is a wasted hour and sometimes a wasted budget.

Speed is a feature

A pipeline you wait for is a pipeline people work around: they batch changes, skip runs, and merge on red. Aim for pull-request feedback in about ten minutes, and treat anything longer as a bug with an owner. The usual wins, in order of payoff:

- **Cache dependencies and build layers.** Downloading the world on every run is the most common single cost.
- **Run independent things in parallel,** and shard the test suite across several machines.
- **Fail fast** on the cheapest signal, and cancel superseded runs, as the `concurrency` block above does.
- **Only build once.** If the image is built again for the deploy, you are testing one artefact and shipping another.
- **Keep integration tests honest but few.** They're the slowest and the most valuable; spend the time deliberately.

Flaky tests are a reliability problem

A test that fails one run in a thousand sounds harmless until you multiply. Five hundred such tests turn a green suite into a coin flip: the probability that all of them pass is $0.999^{500} \approx 61\%$, so two runs in five are red for no reason. Two thousand of them, and 14% of runs pass.

The damage isn't the lost time, it's the lost signal: once "just rerun it" becomes normal, a real failure is indistinguishable from noise, and the pipeline has stopped answering the only question it exists to answer. Treat flakiness as a defect: detect it by rerunning failures automatically and recording which tests pass on retry; quarantine the worst offenders out of the blocking suite with an owner and a deadline; and fix the causes, which are nearly always shared state, real clocks, real networks, or a dependency on test ordering.

The pipeline is a production system

It has credentials, it runs code, and it publishes artefacts that go straight to your users, which makes it a target. The defaults worth having: least-privilege tokens; no secrets available to workflows triggered by pull requests from forks; third-party actions pinned by commit; lockfiles committed and dependency updates reviewed; artefacts signed with an SBOM and provenance attached (chapter 4); and a protected main branch where checks must pass, review is required, and nobody pushes directly.

Measuring delivery

DORA's four keys remain the useful summary: **deployment frequency** and **lead time for changes** for throughput, **change failure rate** and **failed deployment recovery time** for stability. The finding that made them famous is that these move together: teams that deploy more often also fail less, because small changes are easier to review, test and reverse. Use them to watch your own trend, not to compare teams, and remember that any metric a team is judged on becomes a metric a team can produce.

COMMON TRAPS

Rebuilding the artefact for each environment, so what you tested isn't what you ship.

Reruns as a culture. The suite stops meaning anything.

Actions pinned to tags, which can move under you.

Secrets exposed to fork pull requests, where anyone can print them.

YAML's helpfulness. In YAML 1.1 `on:` parses as the boolean true, and `NO` as false, which is why the country code for Norway is a famous bug. Quote anything that might be read as a value.

A pipeline nobody owns, slowly growing to forty minutes.

EXERCISES

EXERCISE 9.1 Cut the pipeline in half

A pull-request pipeline runs in sequence: checkout and install dependencies 6 min, lint 2, type check 3, unit tests 14, integration tests 19, image build 8. Total 52 minutes. Get it under 15 without deleting tests.

SOLUTION

- **Cache dependencies** keyed on the lockfile: 6 minutes becomes about 1 on a hit.
- **Shard the tests** across four runners: unit 14 $\rightarrow$ ~4, integration 19 $\rightarrow$ ~6, assuming they can run in parallel, which is worth the work to make true.
- **Run lint and type checking in parallel** with the tests rather than before them.
- **Cache image layers** in the registry: 8 $\rightarrow$ ~3.

The critical path becomes $1 + \max(5, 10) + 3 = 14$ **minutes**, and the next thing to attack is the integration tests, which now dominate. Two further levers when you need them: run only the tests affected by the change on pull requests while running everything on the main branch, and move slow end-to-end tests to a post-merge stage with a fast rollback (chapter 10) as the safety net.

EXERCISE 9.2 Write the flaky test policy

Your suite has about 30 tests that fail intermittently. Engineers rerun until green. Write a policy the team can actually follow.

SOLUTION

1. **Measure first:** retry failures automatically once, record every test that passed on retry, and publish a weekly list ranked by frequency. Without data, every discussion is anecdote.
2. **Quarantine the top offenders:** move them out of the blocking suite, keep running them, and assign each an owner and a date. A quarantined test still fails the build after its deadline.

3. **Cap the quarantine** at a number the team agrees on, say ten. Beyond that, fixing flakes takes priority over features, because the pipeline has stopped being a signal.
4. **Fix causes, not symptoms:** inject clocks, seed randomness, isolate shared state, avoid sleeps and real networks in tests (*The Engineer's Fleet*, chapter 20).
5. **Never allow merge on red.** The moment that's acceptable, the suite is decoration.

EXERCISE 9.3 Find the supply-chain holes

What's wrong with this job, and what would an attacker do with it?

EXERCISE `deploy.yml`

```
on: pull_request_target

permissions: write-all

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          ref: ${ github.event.pull_request.head.sha }
      - uses: some-org/deploy-action@main
        with:
          aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
          aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
```

SOLUTION

- **pull_request_target** with a checkout of the pull request's own code is the classic mistake: the workflow runs with the repository's secrets, but the code it runs comes from a stranger's branch. Anyone who opens a pull request can print your credentials. Build untrusted code with `pull_request`, without secrets.
- **permissions: write-all** hands that same stranger the ability to push commits, publish packages and edit releases.
- A third-party action pinned to `@main` runs whatever that repository contains today, inside your job, with your secrets. Pin by commit.

- Long-lived cloud keys in secrets, which are copyable and rarely rotated. Use OIDC and a role limited to one repository and branch (chapter 7).
- `ubuntu-latest` changes underneath you; pin the runner image for reproducibility.

CHECKLIST

- ☐ Does a pull request get an answer in about ten minutes?
- ☐ Is the artefact built once and promoted, never rebuilt per environment?
- ☐ Do you know your flaky tests by name, with owners?
- ☐ Are pipeline permissions least-privilege, with no long-lived cloud keys?
- ☐ Are third-party actions and runner images pinned?

GO DEEPER

DOCS Security hardening for GitHub Actions [GitHub](#)

REFERENCE The four keys DORA

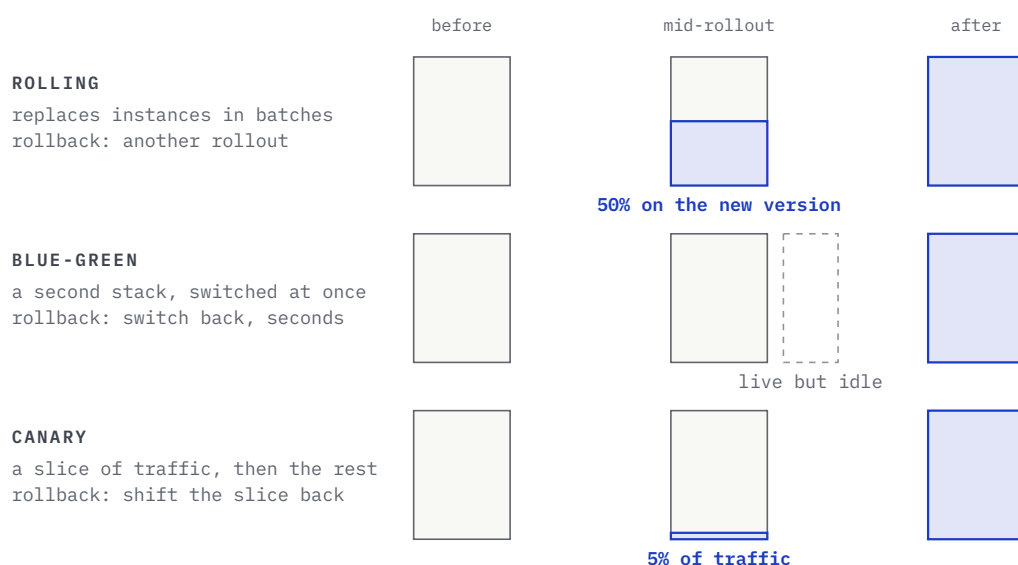
BOOK Continuous Delivery Jez Humble & David Farley · the book the practice comes from

Deploying safely

3 exercises

A deploy is a change to a system that is currently working. Safety comes from four properties: small, gradual, observable, reversible. A deploy missing any one of them is a gamble with your users' time.

Three ways to replace a running version



The three differ in how many users meet a bad version, and in how quickly you can stop. Rolling exposes a growing share and comes back at the speed of another rollout. Blue-green exposes everyone at once and returns in seconds, at the cost of running two full stacks. A canary exposes a sliver, which is also why it takes longer to notice a small regression.

In practice most teams run rolling updates by default (chapter 5), blue-green where a switch must be instant and capacity is cheap, and canaries for the changes that scare them. Feature flags are the finest grain of all, letting the deploy and the release be separate events, which is the subject of *The Engineer's Shipyard*, chapter 17.

What a canary can actually see

Canary analysis compares the new version against a baseline on four things: error rate, latency percentiles, saturation, and at least one business metric such as invoices created a minute. But detection is a statistics problem, and the arithmetic is sobering. To notice an

error rate rising from 0.1% to 1% with reasonable confidence you need on the order of a thousand requests through the canary; to notice a rise to 0.2%, over twenty thousand.

REGRESSION TO DETECT	REQUESTS NEEDED	AT 5% OF 2,000 RPS	AT 1% OF 200 RPS
0.1% → 1.0% errors	~1,100	11 seconds	9 minutes
0.1% → 0.5% errors	~2,900	29 seconds	24 minutes
0.1% → 0.2% errors	~23,000	4 minutes	3 hours

So a ten-minute canary on a small service can only catch a disaster, not a regression, and that's fine as long as you know it: the canary catches the catastrophic, the error budget and alerting catch the subtle (chapter 15). Two details make the comparison honest: compare against a **baseline of the same age**, ideally a freshly started set of old-version instances rather than the long-warm fleet, since cold caches and cold runtimes make any new deployment look slower; and give the canary **representative traffic**, because 5% routed by user ID may be 5% of a very unrepresentative kind.

Rollback is the feature

The single most valuable property of a deployment system is that you can undo it quickly, without thinking. In an incident, roll back first and diagnose afterwards (chapter 16): you cannot debug faster than your users are suffering. That means rollback must be one command or one button, must be practised, and must not depend on the thing that's broken.

The usual thing that makes rollback impossible is the database. A migration that drops a column, renames one, or writes data in a new format can't be reversed by redeploying the old binary, which no longer understands the data. The discipline is **expand and contract**: deploy the schema change that both versions tolerate, deploy the code, then remove the old shape in a later release, once you're sure. Never ship a destructive migration in the same release as the code that depends on it.

Make the platform do the safety

- **Add capacity before removing it**, with `maxUnavailable: 0` and a real readiness probe (chapter 5).
- **Fail the rollout automatically** when it stops making progress, and have the pipeline watch rollout status rather than declaring success on `apply`.
- **Gate on signals, not on time**. An automated canary analysis that checks error rate and latency against the baseline for a fixed window is worth more than a human deciding it "looks fine".

- **Deploy one thing at a time**, so a bad change is identifiable. Batching five changes into one release means bisecting in production.
- **Keep the previous artefact ready to deploy**, by digest.

As for deploying on a Friday: the day isn't the risk. The risk is deploying when you can't observe the result and can't reverse it, which is equally true on Tuesday morning. Teams that deploy rarely have riskier deploys, because each one carries more change, which is the mechanism behind DORA's finding that frequency and stability rise together.

COMMON TRAPS

A rollback path nobody has used, discovered to be broken during the incident.

Destructive migrations shipped with the code that needs them.

Canary windows too short to see anything but a catastrophe, treated as proof.

Comparing a cold canary against a warm fleet and blaming the release.

Deploying ten changes at once because the pipeline is slow (chapter 9).

Declaring success when the apply returns, rather than when the rollout is healthy.

EXERCISES

EXERCISE 10.1 Make this change reversible

The next release renames the `customers.name` column to `legal_name`, and the new code reads and writes only the new name. Plan the releases so that you can roll back at every point.

SOLUTION

1. **Release 1, expand:** add `legal_name`, nullable. No code change. Reversible trivially.
2. **Release 2, dual write:** code writes both columns and reads `name`. Rolling back to release 1 is safe, because `name` is still authoritative.
3. **Backfill** `legal_name` for existing rows, in batches, with the application still running.
4. **Release 3, switch reads** to `legal_name`, still writing both. Rollback returns to reading `name`, which is still current.

5. **Release 4, stop writing** `name` , once you are confident and the backfill is verified.
6. **Release 5, contract: drop** `name` , days or weeks later, when no rollback would reach for it.

Five releases for one rename looks slow until the first time you need step 3 to be reversible at nine in the evening.

EXERCISE 10.2 Design the canary

Your API takes 2,000 requests a second with a 0.1% error rate. You want to catch a release that would double errors, before more than a small number of users are affected. Specify the canary: share of traffic, duration, what you compare, and what stops it.

SOLUTION

Doubling 0.1% to 0.2% needs roughly 23,000 requests through the canary for a confident verdict: at 5% of traffic that's about four minutes, so **5% for five minutes, then 25% for five minutes, then all** is a defensible schedule, and it exposes about 0.6 million requests in total to a version that might be bad, of which about 600 would fail rather than 300.

Compare error rate, p99 latency, saturation, and one business metric, against freshly started baseline instances, broken down by endpoint so a single broken route isn't averaged away. **Stop automatically** if error rate or latency exceeds the baseline by a set margin, or if any alert fires, and make the stop shift traffic back rather than requiring a new rollout. Everything below the canary's resolution is the job of your alerting and error budget (chapter 15), and that's the honest division of labour.

EXERCISE 10.3 The rollback that didn't

An incident: the new release is bad, the team rolls back, and the errors continue. Give four plausible reasons.

SOLUTION

- **A migration ran** that the old code can't read, so the rollback restored code that no longer matches the data.

- **The damage is in state, not code:** the bad version wrote wrong rows, poisoned a cache, or published bad messages that consumers are still processing. Rolling back stops the source and leaves the mess.
- **The rollback didn't take:** the old artefact was rebuilt from a moving tag, the deploy targeted one region, or config changed independently of code and wasn't reverted (chapter 6).
- **The trigger wasn't the release:** a coincident traffic spike, a dependency failing, or a certificate expiring. Correlation with a deploy is strong evidence, not proof.

All four argue for the same habits: reversible migrations, idempotent consumers, deploys by digest, config rolled back with code, and a timeline in the incident channel that lets you check what actually changed when (chapter 17).

CHECKLIST

- ☐ Can you roll back in one command, and when did you last do it?
- ☐ Are migrations backward compatible with the previous release?
- ☐ Does the pipeline verify rollout health rather than the apply succeeding?
- ☐ Does your canary see enough traffic to detect what you claim it detects?
- ☐ Is one deploy one change?

GO DEEPER

ESSAY Blue-green deployment Martin Fowler

ESSAY Parallel change, or expand and contract Martin Fowler

DOCS Automated analysis for progressive delivery Argo Rollouts

PART IV

Observability

Four chapters on knowing what your system is doing: numbers over time, the detail behind them, the path of a single request, and the small number of signals worth waking someone for.

Metrics

3 exercises

Metrics are cheap numbers aggregated over time. They answer "is this healthy, and is it getting worse", and they cannot answer "why was that request slow". The craft is in choosing few series, with labels you can afford, and reading distributions rather than averages.

What to measure

Two checklists cover most of it. For every service, **RED**: request rate, error rate, and duration as a distribution. For every resource, **USE** from chapter 2: utilisation, saturation, errors. Google's four golden signals say the same thing in different words: latency, traffic, errors, saturation. Add one or two numbers that describe the product working, such as invoices created a minute, because those are the only metrics that notice when the system is up and useless.

EXAMPLE metrics.py

```
from prometheus_client import Counter, Histogram

# Labels are dimensions you will group by. Never put anything unbounded in one:
# no user ids, no request ids, no raw URLs. Route templates, not paths.
REQUESTS = Counter(
    "http_requests_total", "HTTP requests",
    ["method", "route", "status"],
)
LATENCY = Histogram(
    "http_request_duration_seconds", "Request duration",
    ["method", "route"],
    buckets=(0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5, 5.0),
)

def record(method, route, status, seconds):
    REQUESTS.labels(method=method, route=route, status=str(status)).inc()
    LATENCY.labels(method=method, route=route).observe(seconds)
```

Scraped, that produces a handful of series per endpoint: two counters and one histogram's worth of buckets.

EXAMPLE GET /metrics

```
http_requests_total{method="GET",route="/invoices/{id}",status="200"} 993.0
http_requests_total{method="GET",route="/invoices/{id}",status="500"} 7.0
http_request_duration_seconds_bucket{le="0.005",method="GET",route="/invoices/{id}"} 0.0
http_request_duration_seconds_bucket{le="0.01",method="GET",route="/invoices/{id}"} 23.0
http_request_duration_seconds_bucket{le="0.025",method="GET",route="/invoices/{id}"} 420.0
```

```
http_request_duration_seconds_bucket{le="0.05",method="GET",route="/invoices/{id}"} 882.0
...
http_request_duration_seconds_bucket{le="+Inf",method="GET",route="/invoices/{id}"} 1000.0
http_request_duration_seconds_count{method="GET",route="/invoices/{id}"} 1000.0
http_request_duration_seconds_sum{method="GET",route="/invoices/{id}"} 55.9
```

Counters only ever increase, which is what makes them reliable across restarts: the query computes a `rate` over them, and a reset to zero is detected rather than counted as a huge negative. Gauges are for things that go up and down, such as queue depth or connections in use. Histograms are for distributions, which is nearly always what you want for latency and size.

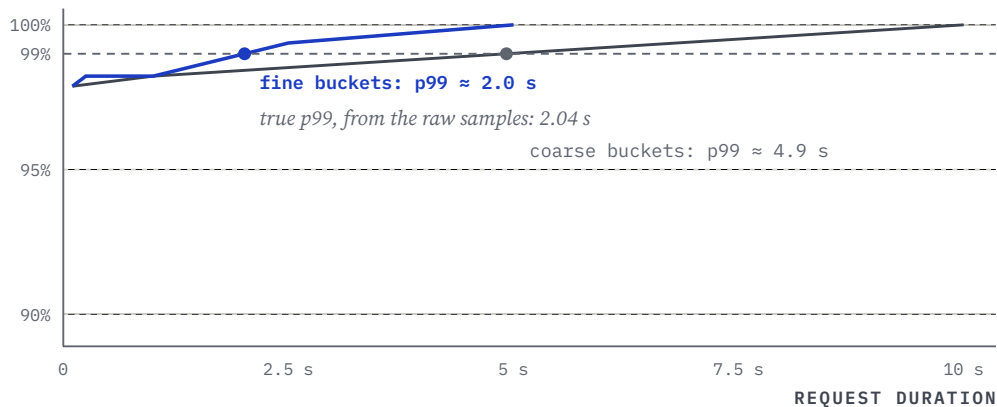
Cardinality is the budget

A metric's cost is its number of **series**, and that's the product of its label values. One request counter with 10 routes, 4 methods and 12 status codes is 480 series per instance, and 24,000 across 50 instances: entirely normal. Add a `customer_id` label with 5,000 customers and it becomes 120 million, which will take down your monitoring before it tells you anything about your service.

So: labels must have **small, bounded** sets of values. Route templates, not URLs with identifiers in them. Status classes or codes, not messages. Never user, request, trace, session or order identifiers, email addresses, or anything a user can influence. That detail belongs in logs and traces, which are built for high cardinality (chapters 12 and 13). When you need per-customer numbers, aggregate them in a separate, deliberately small set of series for your biggest few tenants, or compute them from logs.

Histograms, and the truth about your p99

A histogram counts how many observations fell into each bucket. Percentiles are then *estimated* by finding the bucket the quantile lands in and interpolating across it, which means your p99 is only as good as your bucket boundaries. Ten thousand requests, mostly around 30 ms with 2% of them between 1.5 and 3 seconds, give a true p99 of 2.04 s. With well-chosen buckets the estimate is 2.01 s. With four round buckets at 10 ms, 100 ms, 1 s and 10 s, the same data gives 4.9 s.



The same ten thousand requests, summarised by two sets of buckets. The estimator walks the cumulative counts and interpolates inside whichever bucket the quantile falls in, so a wide, nearly empty bucket at the top of the range produces a confident and badly wrong number. Put bucket boundaries where your service level objective is (chapter 15), not at round numbers.

Prometheus 3.0, released at the end of 2024, adds **native histograms**, which choose exponential buckets automatically at a resolution you pick, and largely remove this problem along with the guesswork. The same release ingests OpenTelemetry metrics directly and accepts UTF-8 metric names. If you are starting now, start there.

Querying without lying to yourself

Four patterns cover most dashboards. `rate(http_requests_total[5m])` for throughput, always summed after the rate, never before. `sum by (route) (rate(...))` to group. `histogram_quantile(0.99, sum by (le, route) (rate(http_request_duration_seconds_bucket[5m])))` for a latency percentile, which works precisely because histogram buckets can be added across instances even though percentiles cannot (chapter 3). And an error ratio as a division of two rates, which is what your objective will be built from.

Expensive queries that dashboards and alerts both need belong in **recording rules**, evaluated once and stored, so an incident doesn't begin with slow dashboards.

COMMON TRAPS

An unbounded label, which is the single most common way to break a metrics system.

Averages, which describe nobody (chapter 3).

Round-number buckets that put your objective in the middle of a wide one.

Summing before taking the rate, which turns a counter reset into a spike.

Instrumenting only the framework, so you know about HTTP and nothing about invoices.

Metrics for debugging. When you find yourself wanting a label with an identifier in it, you want a trace.

EXERCISES

EXERCISE 11.1 Count the series

An engineer adds this to a service running on 40 pods, serving 25 routes: `Counter("api_calls_total", "calls", ["route", "customer_id", "status_code", "pod"])`, with 3,000 customers and 15 status codes. How many series can it create, and what should they have done?

SOLUTION

$25 \times 3,000 \times 15 \times 40 = 45$ **million series**, and that's the ceiling for one metric in one service. Even at a fraction of that, it will exhaust the memory of the metrics system, and its dashboards will time out during the incident it was added to help with.

Two of the labels are wrong for different reasons. `customer_id` is unbounded and belongs in traces or logs. `pod` is usually redundant, because the scrape already identifies the instance, and it multiplies everything by the fleet size while making rollouts churn series. Keep `route` and `status_code`, drop the other two, and if per-customer numbers really are needed, record them for a short list of large tenants or derive them from logs.

EXERCISE 11.2 Pick the instrument

Counter, gauge or histogram, and which labels?

- Invoices successfully sent.
 - Messages waiting in the payments queue.
 - Time taken to render a PDF.
 - Size of uploaded files.
 - Whether the nightly reconciliation job succeeded.
-

SOLUTION

- a. **Counter**, labelled by channel (email, post, portal) at most. Rates and totals come from queries, not from separate metrics.
- b. **Gauge**, labelled by queue. This is a saturation signal and often your best early warning.
- c. **Histogram**, with buckets around your objective, labelled by template or document type if that's bounded.
- d. **Histogram**, because you care about the distribution, particularly the tail that causes timeouts.
- e. **A gauge holding the timestamp of the last success**, which lets you alert on "hasn't succeeded in 26 hours". A boolean success gauge goes stale silently when the job doesn't run at all, which is the failure you most want to catch.

EXERCISE 11.3 Fix the dashboard

A latency panel shows

`avg(rate(http_request_duration_seconds_sum[5m])) / avg(rate(http_request_duration_seconds_count[5m]))` and an error panel shows `sum(rate(http_requests_total{status="500"}[5m]))`. Both look calm during a partial outage. Why, and what would you show instead?

SOLUTION

The first panel is the **mean** latency, in which a few very slow requests disappear among thousands of fast ones, and it's averaged across instances, so one sick pod is invisible (chapter 3). Show `histogram_quantile` at 0.95 and 0.99 over summed buckets, broken down by route, and a maximum across instances so an outlier pod stands out.

The second panel counts 500s only, in absolute numbers. It misses 502s and 503s from the proxy in front, misses timeouts where no response was recorded at all, and tells you nothing about proportion: 50 errors a second is fine at 50,000 requests and an emergency at 100. Show the **ratio** of failed to total requests, count every failing class, and make sure something records requests that never got a response.

CHECKLIST

- ☐ Does every service expose rate, errors and a duration histogram, by route?

- ☐ Is there at least one metric that describes the product working?
- ☐ Could any label take an unbounded set of values?
- ☐ Are histogram buckets placed around your objectives?
- ☐ Do you know the series count of your metrics system, and its trend?

GO DEEPER

DOCS Metric and label naming Prometheus

DOCS Native histograms Prometheus

BOOK Monitoring distributed systems Google SRE book, the four golden signals

Logs

3 exercises

Logs are the detail that metrics throw away, and the most expensive telemetry you keep. Make them structured, correlated with everything else, redacted at the source, and sampled deliberately rather than by accident.

Events, not sentences

A log line is a structured event that happens to be readable, not a sentence with values glued into it. "Processed invoice 8812 for acme in 142ms" is unsearchable at scale; the same event as JSON fields can be filtered, grouped, counted and correlated. Every line should carry the same core context, and adding that context by hand at every call site guarantees it will be missing exactly where you need it.

EXAMPLE logging_setup.py

```
import contextvars
import json
import logging
import time
import zlib

# Set once per request; every line logged while handling it carries these fields.
context = contextvars.ContextVar("log_context", default={})

SENSITIVE = ("password", "token", "secret", "authorization", "card", "api_key")

def _redact(key, value):
    return "***" if any(word in key.lower() for word in SENSITIVE) else value

class JsonFormatter(logging.Formatter):
    def format(self, record):
        event = {
            "ts": time.strftime("%Y-%m-%dT%H:%M:%S", time.gmtime(record.created)) + ".%03dZ" %
                (record.msecs),
            "level": record.levelname.lower(),
            "logger": record.name,
            "message": record.getMessage(),
        }
        event.update(context.get())
        event.update(getattr(record, "fields", {}))
        if record.exc_info:
            event["error_type"] = record.exc_info[0].__name__
            event["stack"] = self.formatException(record.exc_info)
        return json.dumps({k: _redact(k, v) for k, v in event.items()}, default=str)

def log(logger, level, message, **fields):
    """log(logger, logging.INFO, "invoice sent", invoice_id=..., channel="email")"""
    logger.log(level, message, extra={"fields": fields})
```

```
def keep(trace_id: str, rate: float) -> bool:
    """Deterministic sampling: every service keeps the same traces, so they stay complete."""
    return zlib.crc32(trace_id.encode()) % 10_000 < rate * 10_000
```

Set the context once when the request arrives, and every line logged anywhere under it, including from libraries and background tasks in the same task context, carries the trace ID, tenant and version:

EXAMPLE output

```
{
  "ts": "2026-09-25T17:35:45.483Z",
  "level": "info",
  "logger": "invoicing",
  "message": "invoice sent",
  "trace_id": "4bf92f3577b34da6a3ce929d0e0e4736",
  "tenant": "acme",
  "version": "2.7.1",
  "invoice_id": "inv_8812",
  "channel": "email",
  "duration_ms": 142
}
{
  "ts": "2026-09-25T17:35:45.483Z",
  "level": "warning",
  "logger": "invoicing",
  "message": "provider slow",
  "trace_id": "4bf92f3577b34da6a3ce929d0e0e4736",
  "tenant": "acme",
  "version": "2.7.1",
  "provider": "postmark",
  "api_key": "***",
  "duration_ms": 2400
}
```

Note the redaction: a field whose name looks sensitive never reaches the log store, whoever wrote the call. That's the only kind of rule that holds, because the alternative depends on every engineer remembering, forever.

Correlation is the whole point

One field matters more than the rest: the **trace ID**, shared with the tracing system (chapter 13) and propagated to every service the request touches. With it, a customer complaint becomes one search that returns every line from every service for that request, in order. Without it, an investigation is a series of guesses about timestamps. Add the tenant, the release version, and the identifiers a human would ask about, such as the invoice ID, and stop: fields are cheap, but a schema nobody agreed on is not.

Levels, and saying something worth reading

- **Error** means a human should look at this, eventually. If nobody would act, it is not an error.
- **Warning** means something recovered but is worth knowing: a retry, a fallback, a slow dependency.
- **Info** records the decisions a system made: sent, rejected, refunded, retried, throttled.
- **Debug** is for development and for temporary investigation, off in production by default.

Log an event once, where the decision was made, rather than at every layer it passes through. Log the reason, not just the outcome: `"rejected"` is useless, `{"decision": "rejected", "reason": "tax_id_invalid"}` is a dashboard. And never log secrets, tokens,

card numbers or personal data; retention and access control for logs are almost always weaker than for your database, which makes logs a favourite place for data to leak out of (stage 13).

What it costs, and what to do about it

Logging is priced by volume, and volume grows with traffic. Four lines per request at 500 bytes, at 2,000 requests a second, is 4 MB a second: **346 GB a day**, 10 TB a month, which at hosted prices is a serious line item and at self-hosted scale is a cluster someone has to run.

Three levers, in order of effect:

- **Sample successes, keep failures.** Deterministic sampling on the trace ID, as in `keep()` above, means every service makes the same decision, so a kept trace is complete rather than full of holes.
- **Drop debug in production**, and make turning it on for one service, or one tenant, a runtime switch (chapter 6).
- **Tier retention:** days in a searchable store, months in compressed object storage you can restore from when an auditor asks.

A policy of "keep one summary line per successful request, sample the detail at 5%, keep everything for errors and slow requests" takes that 346 GB a day down to about 100 GB, a 70% cut, while making incident investigation *better*, because what remains is signal.

Getting them out of the process

Write to standard output and let the platform collect it (chapter 1). A collector such as the OpenTelemetry Collector, Fluent Bit or Vector adds metadata, batches, and forwards to a store. Don't write log files inside containers, don't make the application talk to the log store directly, and remember that this path is best-effort: logs get dropped under load, so nothing that must be exact, such as billing or audit, may depend on them.

Audit logs are a different system with different rules: who did what to which record, written durably as part of the transaction, append-only, retained for years, and readable by people who are not engineers.

COMMON TRAPS

String interpolation, which makes fields unsearchable.

No trace ID, so correlation is manual.

Logging in a loop, a thousand lines per request when something goes wrong.

Secrets and personal data in logs, where retention is long and access is broad.

Errors logged at three layers, so one failure looks like three.

Billing or audit built on best-effort logs.

EXERCISES

EXERCISE 12.1 Rewrite these lines

What is wrong with each, and what would you write instead?

EXERCISE log output

```
INFO Processing...
ERROR Exception occurred
INFO User admin@acme.com paid with card 4111111111111111, token=eyJhbGci...
WARN Retrying (attempt 3)
DEBUG {"body": "<the entire 40KB request>"}
```

SOLUTION

- **"Processing..."** says nothing: no subject, no identifiers, no outcome. Either log the completed event with its fields and duration, or don't log at all.
- **"Exception occurred"** without a type, a stack, or the identifiers involved forces you to guess. Log the exception type, the stack, and the context.
- **An email address, a card number and a token** in one line: three separate incidents. Redact at the formatter; never log a full card number anywhere, under any circumstances.
- **"Retrying (attempt 3)"** omits what is being retried, why, and how long you'll keep trying. Include the operation, the error, the attempt and the backoff.
- **A 40 KB body** at debug level is a cost and a data-protection problem. Log a hash or a few identifying fields; keep bodies in a system built for them, with a short retention.

EXERCISE 12.2 Cut the bill without going blind

You log 4 lines per request at about 500 bytes, at 2,000 requests a second. Finance wants the logging bill cut by two thirds. Design the policy, and show the arithmetic.

SOLUTION

Today: $2,000 \times 4 \times 500$ bytes = 4 MB/s = **346 GB a day**.

Policy: keep one summary line for every request; keep all four lines for the 1.3% that error or exceed the latency objective; sample the other three lines at 5%, deterministically by trace ID so sampled traces stay whole.

Per request that's $0.987 \times (500 + 0.05 \times 1,500) + 0.013 \times 2,000 \approx 594$ bytes, against 2,000 before: **103 GB a day, a 70% cut**. Then tier it: seven days searchable (about 0.7 TB) and ninety days compressed in object storage for audits and slow investigations.

What you must not cut: anything that errors, anything slow, and audit events. And check the assumption that a summary line is enough by using one during the next incident.

EXERCISE 12.3 Find the request

A customer says their invoice failed to send "around 2pm today". What do you search, in what order?

SOLUTION

1. **Find one identifier.** The tenant, the invoice number, or the email address they were sent from. Search by field, not by free text.
2. **Get the trace ID** from whatever that finds, then search on the trace ID alone. Now you have every service's view of that request in order, and probably the answer.
3. **Widen to the tenant** over that hour: is this one request or every request of theirs?
4. **Compare against the fleet:** error rate and latency for that route during the window (chapter 11), and check whether a deploy or a config change landed then (chapter 10).

If step 2 is impossible because there's no trace ID in the logs, that's the finding: fix it before the next incident, because you will pay this cost every single time.

CHECKLIST

- ☐ Is every log line structured, with a stable set of core fields?
- ☐ Does every line carry a trace ID that also exists in your tracing system?
- ☐ Is redaction enforced centrally rather than by convention?
- ☐ Do you know your daily log volume and its cost, per service?
- ☐ Are audit events written somewhere durable, separate from operational logs?

GO DEEPER

DOCS [Logs in OpenTelemetry and how they join traces](#)

BOOK [Google SRE Workbook on telemetry that supports releases](#)

BOOK [Observability Engineering](#) Majors, Fong-Jones & Miranda · events, and why cardinality matters

Traces

3 exercises

A trace is the story of one request across every service it touched: what happened, in what order, and where the time went. It is the only telemetry that answers "why was that request slow".

Spans, traces and the header that ties them together

A **span** is one operation with a start, a duration, attributes and a parent. A **trace** is the tree of spans that share a trace ID. When a service calls another, it sends the current trace and span IDs in a header, and the receiver makes its spans children of that one. The W3C standard header looks like this:

EXAMPLE `traceparent`

```

traceparent: 00-b6a6820054606aad82c4b3f5c2e55b1a-1d3858ed71703947-03
              ^  ^                               ^           ^
              | trace id (16 bytes)             parent span id   flags: sampled
              version

```

That single header is what makes a distributed system explicable. The same trace ID goes into your logs (chapter 12), so one identifier connects every signal you have.

Instrumenting

OpenTelemetry is the standard: one API and SDK per language, plus a collector that receives, processes and forwards. Traces, metrics and logs are stable across the major languages; continuous profiling entered public alpha in 2026, the year the project graduated from the CNCF. Use automatic instrumentation for your framework, HTTP client and database driver, then add manual spans for the steps your business cares about.

EXAMPLE `tracing.py`

```

import time
from opentelemetry import trace
from opentelemetry.propagate import inject
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor, ConsoleSpanExporter

trace.set_tracer_provider(TracerProvider(
    resource=Resource.create({"service.name": "invoicing-api", "service.version": "2.7.1"})))
trace.get_tracer_provider().add_span_processor(BatchSpanProcessor(ConsoleSpanExporter()))
tracer = trace.get_tracer(__name__)

```

```

def get_invoice(invoice_id, tenant):
    with tracer.start_as_current_span("GET /invoices/{id}") as span:
        span.set_attribute("http.route", "/invoices/{id}") # bounded, like a metric label
        span.set_attribute("tenant.id", tenant) # high cardinality is fine here
        span.set_attribute("invoice.id", invoice_id)
        check_permissions(tenant)
        invoice = load_invoice(invoice_id)
        render_pdf(invoice)
        return invoice

def check_permissions(tenant):
    with tracer.start_as_current_span("authz.check"):
        time.sleep(0.004)

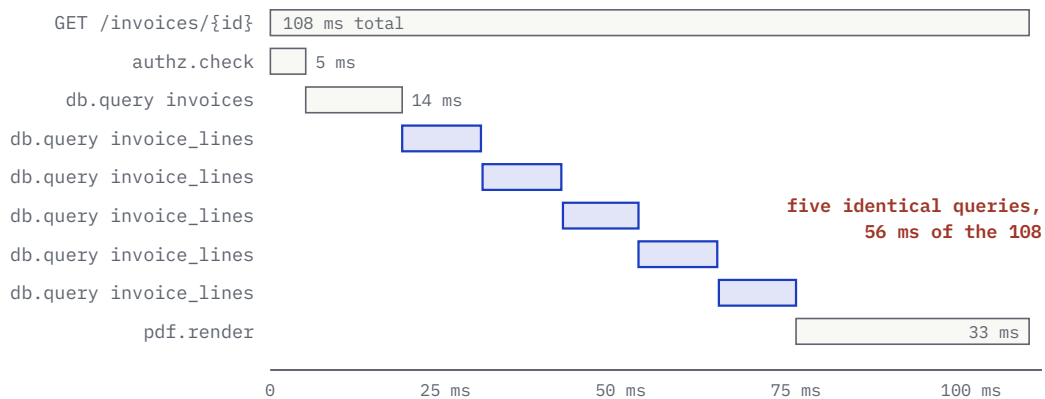
def load_invoice(invoice_id):
    with tracer.start_as_current_span("db.query invoices") as span:
        span.set_attribute("db.statement", "SELECT * FROM invoices WHERE id = $1")
        time.sleep(0.011)
        lines = [f"line_{i}" for i in range(5)]
    for line in lines: # one query per line: the N+1
        with tracer.start_as_current_span("db.query invoice_lines") as span:
            span.set_attribute("db.statement", "SELECT * FROM invoice_lines WHERE id = $1")
            time.sleep(0.009)
    return {"id": invoice_id, "lines": lines}

def render_pdf(invoice):
    with tracer.start_as_current_span("pdf.render") as span:
        headers = {}
        inject(headers) # W3C traceparent, so the next service joins this trace
        span.set_attribute("http.request.header.traceparent", headers["traceparent"])
        time.sleep(0.030)

```

Attributes here can be as specific as you like: tenant, invoice, user, query. That's the complement to chapter 11: metrics must keep cardinality low, and traces exist precisely to carry the high-cardinality detail. Follow the semantic conventions for the standard names, such as `http.route` and `db.statement`, so tools understand your spans without configuration.

Reading a waterfall



One request, nine spans, real timings from the code above. The shape is the finding: five identical queries in sequence, more than half the request, which no metric would ever reveal. This is the classic N+1, and the fix is one query instead of six.

Four shapes are worth recognising at a glance. A **staircase** of identical spans is an N+1 query or a loop of remote calls. One **very long child** is a slow dependency, and the question becomes whether it's slow for everyone. A **gap** between spans is time nobody instrumented: queueing, garbage collection, a lock, or a service that isn't traced. And a **wide parent with narrow children** means the time is in your own code, which is where a profiler comes in (chapter 2).

Sampling

Tracing every request is affordable at small scale and ruinous at large scale, so you sample. **Head sampling** decides at the first service and propagates the decision in that `03` flag, which keeps traces complete but is blind: you throw away 99% of requests before knowing which ones were interesting. **Tail sampling**, done in the collector, buffers the spans of a trace and decides once it's finished, so you can keep every trace that errored, every trace slower than your objective, and a small percentage of the rest. That's the policy most teams end up with, and it's worth the collector.

Whatever you choose, make the logging sampler agree with it, and keep **exemplars**: metric buckets that carry a few example trace IDs, so a spike on a dashboard is one click from a trace that shows why.

The gap that ruins traces

Traces break at asynchronous boundaries. A request publishes a message; the consumer runs five seconds later in another process and starts a brand-new trace, and the story ends exactly where the interesting part began. The fix is to carry the trace context in the

message itself, in a header field, and have the consumer start its span as a child of, or a link to, the producer's. Scheduled jobs and retries deserve the same treatment: a background task with no trace context is a black box the moment it matters.

COMMON TRAPS

No propagation across queues or jobs, so traces stop at the interesting boundary.

Head sampling at 1% and then wondering why the failing request wasn't captured.

Spans without attributes, which tell you an operation was slow but not for whom or on what.

Custom attribute names where a semantic convention exists.

Clock skew between hosts making children appear to start before their parents; trust durations more than absolute times (*The Engineer's Fleet*, chapter 3).

Tracing everything in enormous detail, which costs more than it returns; span count is a budget too.

EXERCISES

EXERCISE 13.1 Diagnose from the shape

What does each waterfall suggest, and what would you do next?

- A 2-second root span with one child of 1.98 s named `http.client GET billing`.
- A 900 ms root span whose children total 120 ms, with a 700 ms gap before the first child.
- A root span with 60 children, each 8 ms, all the same name.
- A root span of 400 ms with two children of 200 ms each, running one after the other, that don't depend on each other.

SOLUTION

- A slow dependency. Check whether billing is slow for everyone or just for this call: its own p99, the endpoint, and whether your timeout is longer than your own objective (*The Engineer's Fleet*, chapter 14).

- b. Uninstrumented time before any work started: queueing at the server, waiting for a connection from a saturated pool, a lock, or a long garbage-collection pause. Add a span around acquisition, and check saturation (chapter 3).
- c. An N+1, or a loop of remote calls. Batch it: one query with an `IN` clause, or one bulk call.
- d. Two independent calls made in sequence. Run them concurrently and halve the latency; this is the most common free win in a trace.

EXERCISE 13.2 Write the sampling policy

Ten thousand requests a second, and your tracing vendor charges by span. Design a policy that keeps the bill sane and still lets you investigate incidents.

SOLUTION

- **Tail sampling in the collector:** keep 100% of traces containing an error, 100% slower than your latency objective, and 1–5% of the rest, chosen deterministically so logs and traces agree.
- **Keep rare routes at a higher rate** than busy ones, or the long tail of endpoints is never sampled at all: sample per route, not globally.
- **Always keep traces for a named set of important tenants or flows**, such as payments, where one broken request matters more than a thousand page views.
- **Cap span volume per service** so one badly instrumented loop can't produce a million spans a minute, and alert when a service's span rate jumps.
- **Keep exemplars in metrics** regardless, so even unsampled periods have a few example traces attached to the dashboard.

EXERCISE 13.3 Trace the background job

An API request enqueues a "send invoice" job. The worker's traces are unrelated to the request's. Describe the changes on both sides, and what to do when one message fans out into ten jobs.

SOLUTION

Producer: inject the current context into the message (a `traceparent` field in the message headers or envelope) inside the span that publishes it.

Consumer: extract that context when the message is received, and start the job's span with it as the parent. Because the job may run much later, most teams start a new trace with a **link** to the original rather than a child span, so a five-second request doesn't appear to last an hour; either is defensible, as long as the link exists.

Fan-out: each of the ten jobs links back to the same producing span, so you can navigate from the request to all the work it caused, and from any job back to its cause. Carry the tenant and the business identifiers as attributes too, since those are what you'll actually search by.

CHECKLIST

- ☐ Does one trace ID follow a request through every service, queue and job?
- ☐ Is the same trace ID in your logs?
- ☐ Are errors and slow requests always sampled?
- ☐ Do spans carry the identifiers you would search by?
- ☐ Could you find the slowest 1% of requests from yesterday and look at one?

GO DEEPER

DOCS [Traces OpenTelemetry](#)

SPEC [W3C Trace Context the traceparent header](#)

PAPER [Dapper Google, 2010 · where all of this comes from](#)

Dashboards and alerts

3 exercises

A dashboard should answer a question. An alert should demand an action from the person it wakes. Everything else is noise, and noise is not neutral: it trains people to ignore the one page that mattered.

Dashboards that answer questions

Start with one page per service that answers "is it healthy?" in ten seconds: request rate, error ratio, latency percentiles, saturation, and the one business metric that says the product is working. Then a second page for "what changed?": deploys and configuration changes as annotations on the same time axis, plus the health of the dependencies. Everything else belongs in drill-downs that you open when the first two pages point somewhere.

Details that make the difference: the same layout on every service's page, so people can read one they've never seen; consistent time ranges across panels; percentiles rather than averages (chapter 3); ratios rather than counts; and a maximum across instances next to the aggregate, so a single sick pod is visible. A forty-panel wall built by accretion is not a dashboard, it's a museum.

Alert on symptoms, not causes

The rule from Google's SRE practice is that pages should be about things users can feel: requests failing, requests too slow, work not being done. Causes such as high CPU, a full disk or a restarting pod belong on dashboards and in tickets, because they may or may not matter, and because a page for each cause means several pages for one incident. Three questions for every alert that wakes someone: is it **real**, is it **urgent** (does it need action now rather than tomorrow), and is it **actionable** (does the person know what to do)? If any answer is no, it's a ticket or a dashboard panel.

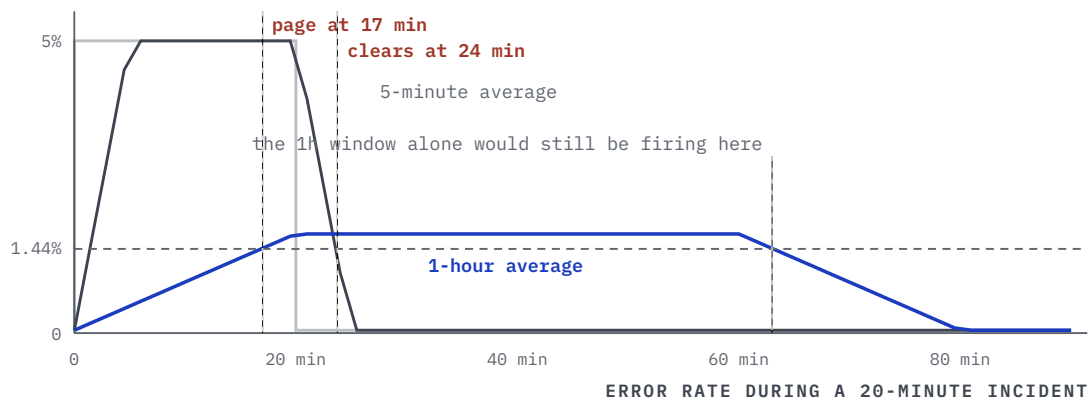
Burn-rate alerting

The best-behaved symptom alert is built on the error budget from chapter 15. If your objective is 99.9% over 30 days, the budget is 43.2 minutes of failure, and what matters is how fast you're spending it. A burn rate of 1 spends the budget exactly over the window; a burn rate of 14.4 spends 2% of a 30-day budget in an hour. Google's recommended starting point pairs a long window with a short one, so alerts fire fast and also *stop* fast:

SEVERITY	LONG WINDOW	SHORT WINDOW	BURN RATE	BUDGET SPENT
Page	1 hour	5 minutes	14.4	2%

Page	6 hours	30 minutes	6	5%
Ticket	3 days	6 hours	1	10%

The second window is the part people leave out, and it's what makes the alert usable.



A twenty-minute burst of 5% errors, seen through two windows. The long window gives precision, so a brief blip doesn't page anyone; the short window gives a fast reset, so the alert closes four minutes after the incident ends rather than forty. Requiring both to be over the threshold gets you both properties.

EXAMPLE slo-rules.yaml

```
groups:
- name: invoicing-api-slis
  interval: 30s
  rules:
    # One definition of "bad", recorded once and used by every alert and dashboard.
    - record: job:slo_errors:ratio_rate5m
      expr: |
        sum by (job) (rate(http_requests_total{job="invoicing-api",status=~"5.."}[5m]))
        / sum by (job) (rate(http_requests_total{job="invoicing-api"}[5m]))
    - record: job:slo_errors:ratio_rate1h
      expr: |
        sum by (job) (rate(http_requests_total{job="invoicing-api",status=~"5.."}[1h]))
        / sum by (job) (rate(http_requests_total{job="invoicing-api"}[1h]))
    - record: job:slo_errors:ratio_rate6h
      expr: |
        sum by (job) (rate(http_requests_total{job="invoicing-api",status=~"5.."}[6h]))
        / sum by (job) (rate(http_requests_total{job="invoicing-api"}[6h]))

- name: invoicing-api-burn-rate
  rules:
    # 99.9% availability over 30 days. 14.4x burn for an hour spends 2% of the budget.
    - alert: InvoicingApiBudgetBurningFast
      expr: |
        job:slo_errors:ratio_rate1h > (14.4 * 0.001)
        and job:slo_errors:ratio_rate5m > (14.4 * 0.001)
      for: 2m
```

```

labels:
  severity: page
  service: invoicing-api
annotations:
  summary: Invoicing API is spending its error budget 14x too fast
  description: >-
    At this rate 2% of the 30-day budget goes in an hour. Mitigate first:
    check the last deploy and config change, then roll back.
  runbook: https://runbooks.internal/invoicing-api/budget-burn

# 6x burn over six hours spends 5% of the budget: slower, still worth waking someone.
- alert: InvoicingApiBudgetBurningSteadily
  expr: |
    job:slo_errors:ratio_rate6h > (6 * 0.001)
    and job:slo_errors:ratio_rate1h > (6 * 0.001)
  for: 15m
  labels:
    severity: page
    service: invoicing-api
  annotations:
    summary: Invoicing API has been failing steadily for hours
    runbook: https://runbooks.internal/invoicing-api/budget-burn

```

Note what each part buys. The recording rules give one definition of "bad" that dashboards and alerts share, so nobody argues during an incident about whether 499s count. The `for` clause absorbs single scrapes. The severity label routes the alert. And every alert carries a **runbook link**, because the person it wakes may have joined last month.

Keeping the pager honest

- **Every page needs an owner and a runbook.** An alert nobody owns is deleted, not inherited.
- **Budget the pages.** A widely used rule of thumb is no more than about two pages per on-call shift; beyond that, people stop reading them properly, and burnout follows (chapter 16).
- **Review alerts monthly.** For each page: was it real, was it actionable, did it need to be at that hour? Delete or downgrade the failures.
- **Group and inhibit.** One cause should produce one page, not twelve: group by service, and suppress dependent alerts when the upstream one is firing.
- **Silence deliberately,** with an expiry, during known maintenance. A permanent silence is a deleted alert with extra steps.
- **Alert on absence too:** no requests, no jobs run, no data received. Silence is the failure mode monitoring misses most often.

COMMON TRAPS

Paging on causes, so one incident wakes five people with five different symptoms.

Static thresholds on CPU and memory, which fire during normal peaks and stay quiet during real outages.

Long windows without short ones, so alerts linger for an hour after recovery.

Alerts with no runbook, which turn every page into an investigation from scratch.

Dashboards that only make sense to their author.

No alert for "nothing is happening", which is what a broken queue consumer looks like.

EXERCISES

EXERCISE 14.1 Triage the alert list

For each, decide: page, ticket, dashboard only, or delete.

- a. CPU above 80% for five minutes on any instance.
 - b. Checkout error ratio above 5% for two minutes.
 - c. A pod restarted.
 - d. The nightly billing job has not succeeded in 26 hours.
 - e. Certificate expires in 21 days.
 - f. Disk will be full in four hours at the current rate.
-

SOLUTION

- a. **Dashboard.** High CPU is normal at peak and says nothing about users. If it matters, it will show up as latency.
- b. **Page.** Users are being turned away from the thing that makes money.
- c. **Dashboard**, or a ticket if it becomes frequent. One restart is the platform doing its job; a restart loop is a symptom with a different alert.
- d. **Page**, if invoices depend on it; otherwise a ticket first thing in the morning. Alert on the absence of success, not on failure, so a job that never ran is caught.
- e. **Ticket.** Three weeks of warning needs a work item, not a wake-up. Page only in the last day or two, and automate renewal so it never gets there.
- f. **Page.** It is predictive, urgent and actionable, which is the ideal shape for an alert.

EXERCISE 14.2 Why did it page for 40 minutes after recovery?

An alert uses a one-hour error-rate window and fires 17 minutes into a 20-minute incident. The incident is fixed at 20 minutes, and the alert keeps firing until minute 63. Explain, and fix it without making the alert noisier.

SOLUTION

A one-hour average still contains the incident for a full hour afterwards. It decays slowly: with 5% errors for 20 minutes, the hourly ratio stays above the 1.44% threshold until the bad minutes fall out of the window, at minute 63. Meanwhile the on-call engineer sees a firing alert for a system that's healthy, which is exactly how people learn to distrust alerts.

The fix is the short window: require both the one-hour *and* the five-minute ratio to be above the threshold. The long window keeps precision, so a two-minute blip never pages; the short one collapses within minutes of recovery, so the alert clears at minute 24.

Lengthening `for` would have done the opposite, delaying the page and doing nothing about the tail.

EXERCISE 14.3 Design the service page

Design the one dashboard your team should open first when the invoicing API is paged. Name the panels and say what decision each supports.

SOLUTION

- **Error ratio and request rate, by route** — is this everything or one endpoint?
- **Latency p50, p95, p99, plus the worst instance** — is it slow, and is it one pod?
- **Error budget remaining and current burn rate** — how urgent is this, and should we stop deploying (chapter 15)?
- **Deploy and config-change annotations** on every panel — did we do this to ourselves?
- **Dependency health:** database, queue, payment provider, in one row — is it ours?

- **Saturation:** queue depth, connection pool usage, CPU throttling — is it capacity (chapter 3)?
- **One business metric:** invoices sent a minute, against last week — is the product actually working?

Seven panels, one screen, no scrolling. Everything else is a link.

CHECKLIST

- ☐ Does every page correspond to something a user would notice?
- ☐ Does every alert have an owner, a runbook and a severity that routes it?
- ☐ Do your alerts clear promptly when the problem does?
- ☐ Is there an alert for work silently not happening?
- ☐ How many pages did your team get last month, and how many were actionable?

GO DEEPER

- BOOK** Alerting on SLOs Google SRE Workbook · where the burn-rate table comes from
- BOOK** Practical alerting Google SRE book
- DOCS** Alerting best practices Prometheus

PART V

Reliability

Three chapters on the practice rather than the plumbing: agreeing how reliable the system should be, responding when it isn't, and learning enough from each failure that the next one is different.

SLIs, SLOs and error budgets

3 exercises

An objective turns "reliable" into a number you can trade against. Its purpose is not accuracy; it is to make one decision automatic: when to stop shipping features and go fix things.

Indicator, objective, budget

A **service level indicator** is a ratio: good events divided by valid events, measured as close to the user as you can get. A **service level objective** is a target for that ratio over a window, such as 99.9% over 30 days. The **error budget** is what's left: the amount of failure you have explicitly decided to tolerate, and therefore to spend.

The budget is the part that changes behaviour. It converts an argument about whether to ship into arithmetic, and it gives reliability work a moment when it automatically outranks features.

OBJECTIVE	FAILURE ALLOWED PER MONTH	PER YEAR
99%	7 hours 12 min	3.6 days
99.5%	3 hours 36 min	1.8 days
99.9%	43 minutes	8.8 hours
99.95%	21.6 minutes	4.4 hours
99.99%	4.3 minutes	53 minutes

Read that table before promising anything. At 99.99%, a single four-minute deploy that goes wrong spends the entire month, which means no manual response is fast enough: everything must be automatic. Each additional nine costs roughly an order of magnitude more engineering, so the question is never "how reliable can we be" but "how reliable do our users need us to be, and what will we give up for it".

Choosing indicators that mean something

Pick two to four per user journey, not per service:

- **Availability:** the proportion of valid requests that got a successful response. Decide up front what counts: a 400 caused by the client is usually not your failure; a 429 you issued under load probably is.
- **Latency:** the proportion of requests faster than a threshold. Note the shape: not "p99 under 300 ms" but "99% of requests under 300 ms", which is the same number stated so that it can be summed over a month.
- **Freshness** for pipelines and derived data: the proportion of time the data is newer than some age.
- **Correctness** where it matters more than speed: invoices with correct totals, payments reconciled.

Measure where the user is. A server-side timer excludes queueing, the load balancer, DNS and the network, which is where slow requests are born (chapter 3). Use the load balancer's numbers, or real user measurements, and use synthetic checks for services too quiet for statistics to work.

Spending the budget

With a 99.9% objective over 30 days, you have 43.2 minutes. One 22-minute outage spends 51% of it. A partial failure counts proportionally: 40% of requests failing for 50 minutes spends 46%. Counting requests rather than minutes gives the same answer and handles partial failure naturally: at 2,000 requests a second, a month is about 5.2 billion requests, the budget is 5.2 million failures, and that incident used 2.4 million of them.

The policy is what makes it real, and it must be agreed **before** you need it, with whoever decides priorities:

BUDGET REMAINING WHAT HAPPENS

> 50%	Normal: ship, experiment, take reasonable risks.
20–50%	Caution: no risky migrations, canaries get longer, reliability bugs jump the queue.
< 20%	Freeze feature deploys except fixes; the team works on reliability until the budget recovers.
Exhausted	Everything stops until the objective is met again, and the freeze is reviewed by the same people who set the objective.

A budget nobody spends is also information: if you finish every month at 99.99% against a 99.9% objective, you are buying reliability nobody asked for, and you could be shipping faster or spending less (chapter 19).

What you can promise

Your objective is bounded by what you depend on. An API that calls three services in sequence, each with a 99.95% objective, cannot promise better than about 99.85% before its own code fails at all (*The Engineer's Fleet*, chapter 1). Either reduce the dependencies on the critical path, make them optional with fallbacks, or promise less. Setting an objective higher than the arithmetic allows is a way of agreeing to fail.

COMMON TRAPS

An objective of 100%, which forbids every deploy and is never met.

Measuring on the server and missing everything before it.

An SLO per microservice, so nobody knows whether users are affected.

A budget with no policy, which is a chart nobody acts on.

Counting only total outages, so a week of 3% errors looks like a perfect week.

Objectives set by engineers alone, which makes the freeze a negotiation at the worst moment.

EXERCISES

EXERCISE 15.1 Write the indicators

Define an SLI and a starting objective for each, and say where you would measure it.

- Creating and sending an invoice from the web app.
 - The public API used by customers' integrations.
 - The nightly job that generates VAT reports.
 - The PDF download link in a customer email.
-

SOLUTION

- Availability:** successful invoice-send requests over all valid attempts, measured at the load balancer, with a latency SLI of "95% under 2 s". Start at 99.9% and adjust after a month of data.

- b. **Availability and latency per endpoint class**, measured at the edge, excluding 4xx caused by the client but including your own 429s. Publish it; an API's reliability is part of its interface.
- c. **Freshness**: the report is available by 07:00 on 99% of days. A batch job's users care about the deadline, not the duration.
- d. **Availability of the object store path**, plus correctness: the link resolves to the right document. Measure with a synthetic check that downloads a known invoice every few minutes, because real traffic here is too sparse to detect a problem quickly.

EXERCISE 15.2 Spend the budget

Your objective is 99.9% over 30 days. On day 4 a bad deploy causes 100% errors for 9 minutes. On day 11 a database failover causes 30% errors for 25 minutes. On day 19 a slow dependency pushes 8% of requests over the latency threshold for 3 hours. How much budget is left, and what should the team do?

SOLUTION

The budget is 43.2 minutes of failure-equivalent.

- Day 4: 9 minutes $\times$ 100% = **9.0**
- Day 11: 25 minutes $\times$ 30% = **7.5**
- Day 19: 180 minutes $\times$ 8% = **14.4** (against the latency objective, if it shares the budget)

Total spent: 30.9 minutes, leaving 12.3 minutes, about **28% remaining** with eleven days to go. Under the policy above that means caution rather than a freeze: postpone risky changes, lengthen canaries, and take the reliability work that's been waiting.

The interesting result is that the quiet three-hour degradation cost more budget than either outage. That is exactly what error budgets are for: they make slow, partial pain visible, which dashboards and incident counts do not.

EXERCISE 15.3 Negotiate the number

Sales has promised a customer 99.99% availability in a contract. You currently run at about 99.9%, with a single-region deployment and a managed database that takes 40 seconds to fail over. What do you say?

SOLUTION

Start with the arithmetic, not with "no": 99.99% is 4.3 minutes of failure a month. A single database failover spends 40 seconds of that, and you cannot control when it happens. Two failovers and an unlucky deploy exhaust the month. So the promise implies multi-region capacity with automatic failover, automated deploys with automatic rollback, no manual steps in the recovery path, and enough people to run it.

Then offer the options with their costs: keep 99.9% and set the contract there with credits attached; invest in the specific things that move the number, in order of value (eliminate the manual steps, then failover automation, then multi-region); or price the difference and let the customer decide whether they want it. And check what the customer actually needs, which is often a recovery time promise or scheduled-maintenance guarantees rather than a nines figure.

CHECKLIST

- ☐ Does each important user journey have an SLI measured near the user?
- ☐ Is there a written budget policy, agreed with whoever sets priorities?
- ☐ Can you say how much budget is left right now?
- ☐ Is your objective achievable given your dependencies?
- ☐ Has an objective ever caused you to change plans? If not, it isn't doing its job.

GO DEEPER

BOOK [Implementing SLOs Google SRE Workbook](#)

BOOK [Embracing risk Google SRE book](#)

BOOK [Implementing Service Level Objectives Alex Hidalgo · a whole book on getting these right](#)

On call and incident response

3 exercises

An incident is a coordination problem with a technical component. Mitigate before you diagnose, make one person responsible for deciding, and communicate more often than feels necessary.

Mitigate first

The instinct of every engineer is to understand the problem. During an incident that instinct is wrong. Your first job is to stop the damage, and the moves that do it rarely require knowing why: roll back the last deploy, revert the last configuration change, fail over to another zone, shed load, disable the expensive feature, turn off the batch job that's crushing the database. Understanding can wait for the postmortem, when nobody is losing money.

This has a corollary worth stating plainly: **build the mitigations before you need them**. A rollback button, a feature flag, a load-shedding switch, a read-only mode, a way to pause consumers. In the middle of an incident you will use the tools that already exist, not the ones you wish existed.

Roles, even in a team of four

The thing that makes incidents go badly is not a lack of technical skill, it's six people investigating the same thing while nobody talks to customers. Name roles out loud, even if one person holds two of them:

- **Incident commander:** decides, tracks what's being tried, and does not touch a keyboard. Their job is the shape of the response, not the fix.
- **Operations:** the people making changes, one at a time, saying what they're doing before they do it.
- **Communications:** updates the status page and the internal channel on a cadence, and is the single voice to the rest of the company.
- **Scribe:** writes the timeline as it happens. Timestamps, actions, observations. This is worth a whole person, because nobody can reconstruct it afterwards.

The commander's most useful sentences are "what do we know", "what are we trying next", "who is doing it" and "when do we check back".

Severity, declared early

SEVERITY	MEANS	RESPONSE
Sev 1	Users can't do the core thing, or data is at risk	Page immediately, commander named, status page updated, updates every 30 min
Sev 2	Significant degradation or a feature down	Page during hours, handled by the on-call, updates hourly
Sev 3	Minor or contained; a workaround exists	Ticket, fixed in normal work

Declare early and downgrade later. The cost of declaring a sev 1 that turns out to be a sev 2 is a few people's attention for twenty minutes; the cost of the reverse is discovering at hour three that nobody told the customers.

The first ten minutes

- 01 **Declare.** Open a channel, name it after the incident, state the symptom and the time it started.
- 02 **Take command,** or say who has it. Post it in the channel so it's unambiguous.
- 03 **Establish impact:** who is affected, how many, since when, and which user journey. This determines everything else.
- 04 **Check what changed:** deploys, configuration, feature flags, infrastructure changes, certificate expiries, and the dependencies' status pages. Most incidents are something someone did.
- 05 **Mitigate** with the biggest reversible lever you have.
- 06 **Tell people:** internal update, and the status page if customers can see it. Say what you know, what you don't, and when you'll next update.

Communicating while it burns

Update on a fixed cadence even when there is nothing new, because silence is read as chaos. Write for customers in terms of what they can and can't do, not components: "invoices created in the last hour may not have been emailed; we are resending them" rather than "the queue consumer is stuck". Use UTC and absolute timestamps everywhere. Keep speculation out of customer-facing messages and put it in the internal channel, where it belongs.

Sustainable on call

An on-call rotation is a load-bearing part of your system, and it fails the same way as any other overloaded component.

- **Enough people.** A 24/7 rotation needs six to eight, or two teams in different time zones. Four people means one week in four, which people leave over.
- **A page budget.** Aim for at most a couple of pages per shift, and treat anything above that as a bug in the alerts or in the system (chapter 14).
- **Recovery time.** Someone up at 3am does not work the next day. This has to be a rule, not a kindness.
- **Pay for it, or at least compensate in time.** Unpaid on call is a quiet tax on the people least able to refuse it.
- **A handover ritual:** what fired, what's still broken, what to watch. Five minutes that saves the next person an hour.
- **Only page for what a human can fix now.** If the answer is "restart it", automate the restart.

New engineers should shadow a rotation before joining it, and every team should run the occasional **game day**: break something in a controlled way and practise the response, including the parts that are not technical, such as finding the status page password (*The Engineer's Fleet*, chapter 20).

COMMON TRAPS

Debugging before mitigating, while users keep failing.

.....

Nobody in command, so three people change three things at once.

.....

Changing several things simultaneously, so nobody knows what worked.

.....

Silence towards customers because there's no news yet.

.....

No written timeline, which makes the postmortem fiction.

.....

Heroes. If one person is always the one who can fix it, that's a single point of failure with a pulse.

.....

EXERCISES

EXERCISE 16.1 Run the first ten minutes

03:14. You are paged: error ratio on the invoicing API above 20%, burning the budget fourteen times too fast. You are alone. Write what you do, in order, for ten minutes.

SOLUTION

1. **Acknowledge the page** so it stops escalating, and open an incident channel with the symptom and the time.
2. **Look at one dashboard** (chapter 14): is it every route or one, every instance or one, and when did it start?
3. **Check what changed** in the last hour: deploys, config, flags, infrastructure, dependency status pages.
4. **If a change lines up, revert it.** Don't verify the causal chain first; verify by watching the error rate after the revert.
5. **If nothing lines up,** reach for the generic mitigations: fail over, restart the unhealthy instances, shed load, disable the expensive feature.
6. **Escalate** if you don't have a mitigation within about ten minutes. Waking a colleague is cheaper than an hour of downtime, and being alone at 3am is when judgement is worst.
7. **Post an update** internally and on the status page: impact, what you're doing, next update in 30 minutes.
8. **Write the timeline as you go,** even in rough notes with timestamps.

EXERCISE 16.2 Classify and respond

Assign a severity and a response to each.

- a. PDF generation is failing for 3% of invoices; the rest of the product works.
- b. Every login fails.
- c. A bug has been overwriting the wrong customer's address for two days, and is still running.
- d. The internal admin dashboard is down; customers are unaffected.

SOLUTION

- a. **Sev 2.** A real but contained failure with a workaround (resend later). Page during hours, fix today, and check whether the 3% is random or one tenant, which changes everything.
- b. **Sev 1.** Nobody can use the product at all. Commander, status page, all hands.

- c. **Sev 1**, despite nothing being "down". Data corruption is worse than downtime because it is quiet, it compounds, and it may be unrecoverable. Stop the writer first, then assess the damage and the restore path (chapter 18).
- d. **Sev 3**, unless it blocks the response to another incident, in which case it becomes part of that one. Internal tooling gets a lower severity and a higher priority than people expect, because it is how you fix everything else.

EXERCISE 16.3 Fix the rotation

Four engineers share a 24/7 rotation, one week each. They get about 15 pages a week, most of which are "restart the worker" or noisy thresholds. Two people have resigned in six months. What do you change, in what order?

SOLUTION

1. **Stop the bleeding this week:** review the top five alert sources, delete or downgrade everything that isn't urgent and actionable, and silence the known-noisy ones with an expiry. Fifteen pages a week is an alerting failure, not a reliability one.
2. **Automate the restarts.** If the answer to a page is always the same command, the machine should run it, and the alert should become a trend you look at on Tuesday.
3. **Widen the rotation** to six or more, including engineers from adjacent teams with a shadowing period, or split day and night coverage.
4. **Make the load visible:** pages per shift, per person, per alert, reviewed monthly with the same seriousness as an SLO.
5. **Then fix the underlying causes** with budgeted time: the worker that keeps dying, the dependency that keeps timing out. Use the error budget to buy that time (chapter 15).

The order matters. Nobody will fix the causes while they're being woken up fifteen times a week.

CHECKLIST

- ☐ Can you mitigate the three most likely failures without knowing their cause?

- ☐ Does everyone know how to declare an incident and take command?
- ☐ Is there a status page, and does someone own updating it?
- ☐ Are pages per shift within a number people can sustain?
- ☐ When did you last practise, rather than wait for a real one?

GO DEEPER

BOOK Managing incidents Google SRE book

GUIDE Incident response documentation PagerDuty · roles, severities and comms,
free to copy

BOOK Being on call Google SRE book

Postmortems

3 exercises

The postmortem is where an outage becomes an improvement. Write it blamelessly, from facts with timestamps, and finish with a small number of actions that have owners and dates, because that is the only part that changes anything.

Blameless means "the system allowed it"

Blameless does not mean nobody is accountable. It means that when someone ran the command, approved the plan or missed the alert, the useful question is not "why were they careless" but "why was it easy to do, and why did nothing catch it". People act reasonably given what they know at the time; if the outcome was bad, the information, the tooling or the defaults were bad.

There is a practical reason as well as a humane one. In a culture where mistakes are punished, the details you need stop arriving: people hesitate to say what they did, near misses are never reported, and your picture of the system becomes a fiction. The test of a blameless culture is simple: does the person who typed the command write the postmortem, and do they still work here?

What it contains

- **Impact, in user terms and numbers:** who was affected, how many, for how long, what it cost, how much error budget it spent.
- **Detection:** how you found out, and how long that took. If a customer told you, say so.
- **Timeline:** facts with timestamps, including the actions you took that didn't help.
- **Contributing factors,** plural. Complex systems fail because several things line up.
- **What went well,** which is how good practice gets reinforced, and **where you were lucky,** which is the most valuable section and the one most often missing.
- **Actions** with owners, dates and priorities.

Resist "the root cause". Asking why five times produces a single chain that flatters whoever wrote it. The Cloudflare outage of November 2025 was a database permissions change, *and* a generated file with no size validation, *and* a hard limit in a proxy that failed closed instead of degrading, *and* a deployment path for that file that reached the whole network at once (chapter 6). Remove any one and the day is ordinary.

A worked example

Invoicing API failing for 23 minutes, 14 March 2026

****Impact.**** 09:12-09:35 UTC, 62% of invoice-send requests failed (~310,000 requests, 2,400 customers). 4,100 invoices were delayed; all were sent by 10:20. No data lost. Spent 51% of March's error budget.

****Detection.**** The fast burn-rate alert paged at 09:14, two minutes after the first errors. The first customer email arrived at 09:21.

Timeline (UTC)

```
| 09:08 | Release 2.7.0 finishes rolling out (canary: 3 minutes at 5%).  
| 09:12 | Error ratio rises from 0.05% to 62%.  
| 09:14 | Page. On-call acknowledges, opens #inc-2026-03-14.  
| 09:17 | Database and PDF provider both healthy; dashboards show no saturation.  
| 09:21 | On-call correlates the start with the 09:08 deploy.  
| 09:22 | Rollback to 2.6.4 started.  
| 09:26 | Rollback complete; error ratio begins falling.  
| 09:35 | Baseline restored. Incident downgraded to sev 2.  
| 10:20 | Retry worker has resent all delayed invoices.
```

What happened

2.7.0 introduced a connection pool for the PDF service, sized from a new setting, `PDF_POOL_SIZE`. The setting existed in staging but not in production. The config loader fell back to a default of 2, so every instance could make two concurrent PDF calls; requests queued behind them and timed out.

Contributing factors

1. A new required setting had a silent default instead of failing at start-up.
2. Staging had the value because it was set by hand in January; it was never in code, so "it works in staging" proved nothing.
3. The canary ran 3 minutes at 5% of traffic: ~18,000 requests, below the number needed to detect this reliably, and its analysis only checked latency, not errors.
4. Pool saturation was not instrumented, so no dashboard showed the queue building.

What went well

- The burn-rate alert fired in two minutes, before any customer wrote in.
- Rollback took four minutes and was not debated.
- The retry worker resent every delayed invoice with no manual work.

Where we were lucky

- It happened at 09:12 on a Tuesday. The same release at 02:00 would have run for hours: our slow burn-rate alert would have paged at ~01:30 into the budget, and nobody would have correlated it with a deploy from the previous afternoon.

Actions

```
| Action | Owner | By | Priority |  
|---|---|---|---|  
| Required settings fail at start-up, with all problems listed | R. Okafor | 21 Mar | P1 |  
| Canary analysis: check error ratio, require a minimum request count | S. Lind | 28 Mar | P1 |  
| Instrument pool saturation and alert on queueing | R. Okafor | 28 Mar | P2 |  
| Move staging config into the repository; add drift detection | T. Almeida | 11 Apr | P2 |
```

Actions that actually happen

Three or four actions, each small enough to finish in a fortnight, beat fifteen aspirations. Prefer changes that make the failure impossible or harmless over changes that ask people to be more careful: a validation, a default, a guard rail, a test. "Add more monitoring" is not an action; "alert when the PDF pool has been saturated for five minutes, owned by Rita, by the 28th" is.

Then track them like any other work, and review completion monthly. An organisation that writes excellent postmortems and completes none of the actions has built an expensive storytelling habit. If the same contributing factor appears in three postmortems, that is the finding, and it outranks everything else on the roadmap.

The meeting, and the audience

Hold the review within a few days, while memory is fresh: thirty minutes, with the people involved and anyone who wants to learn. The author walks the timeline, the group challenges the contributing factors and the actions, and nobody relitigates decisions with information that wasn't available at the time. Publish it where the whole engineering organisation can read it, because the second most valuable reader is someone on another team who has the same latent problem.

Customer-facing write-ups are a different document with the same facts: what happened, what it meant for them, what you're changing, no jargon and no blame shifting. Done well, they build more trust than the outage cost; the good public postmortems are worth studying for that reason alone.

COMMON TRAPS

"Human error" as a cause, which ends the investigation exactly where it should begin.

A single root cause, chosen because the chain stopped somewhere convenient.

Counterfactuals: "they should have noticed" describes a world that didn't exist.

Fifteen action items, none of which will be done.

No timeline, so the document is a memory rather than a record.

Postmortems only for sev 1, which throws away the cheap lessons from near misses.

EXERCISES

EXERCISE 17.1 Rewrite the blame

Rewrite each sentence so that it describes the system rather than the person.

- a. "Dan deployed without running the migration checklist."
- b. "The on-call engineer didn't notice the alert for 40 minutes."
- c. "QA missed this bug."
- d. "Someone should have known this would use more memory."

SOLUTION

- a. "Deploying does not require the migration checks; they live in a wiki page and are not enforced by the pipeline. Any engineer following the documented steps would have done the same."
- b. "The alert was routed to a channel rather than the pager, and the on-call had received nine non-actionable pages that shift. Detection took 40 minutes because nothing woke anyone."
- c. "There was no test covering this path, and the behaviour only appears with production-sized data, which no environment has."
- d. "Memory use of this code path was not measured before release, and there was no limit or alert that would have caught the increase before it caused evictions."

Each rewrite points at something you can change. "Be more careful" does not.

EXERCISE 17.2 Find the factors

An engineer runs a data-fix script in production that deletes 40,000 rows instead of 400 because of a missing `WHERE` clause. It is noticed 90 minutes later. The restore takes 3 hours. List the contributing factors and the actions you would propose.

SOLUTION

Factors, none of which is "the engineer was careless":

- Ad-hoc scripts run against production with write access to every table, from a personal session, with no review.
- No dry-run convention: the script had no mode that printed what it would affect.
- No guard rail: a delete affecting thousands of rows proceeded without confirmation or a transaction to inspect first.

- Nothing alerted on a large deletion, so detection took 90 minutes and came from a human.
- The restore path was slow and untested, which turned a mistake into a three-hour outage (chapter 18).

Actions: require review for production data changes and run them through a job rather than a shell; make dry-run the default and print affected counts; add a database guard or a wrapper that refuses deletes above a threshold outside a transaction; alert on unusual row-count changes; and practise the restore so it takes minutes, not hours.

EXERCISE 17.3 Judge the actions

Which of these are real action items, and how would you fix the rest?

- "Improve monitoring."
- "Add an alert when the replica lag exceeds 30 seconds — Priya, 10 April."
- "Be more careful with production changes."
- "Investigate whether we should move to a multi-region setup."
- "Add a test that the importer rejects files over 50 MB — Sam, next sprint."

SOLUTION

- **(b) and (e) are real:** specific, owned, dated, and each removes a failure mode. (e)'s "next sprint" is weaker than a date, but it's trackable.
- (a) is a wish. Turn it into the specific signal that would have shortened this incident, with a threshold and an owner.
- (c) is blame wearing a hat. Replace it with the guard rail that makes carelessness harmless.
- (d) is a project, not an action. If it's worth doing, the action is "write a one-page proposal comparing multi-region against improving failover time, owner, date" so a decision gets made instead of being reopened after every incident.

CHECKLIST

- ☐ Does every significant incident get a written postmortem within days?

- ☐ Does it list several contributing factors, not one root cause?
- ☐ Are the actions few, owned, dated, and completed?
- ☐ Is there a "where we were lucky" section?
- ☐ Would the person who typed the command be comfortable writing it?

GO DEEPER

BOOK Postmortem culture Google SRE book, with a full example

REPORT Cloudflare, 18 November 2025 a public postmortem worth imitating

ESSAY How Complex Systems Fail Richard Cook · eighteen short paragraphs, read them twice

PART VI

Keeping it running

Three chapters on what makes the rest sustainable: recovery you have actually tested, a bill you understand, and the ordinary discipline that keeps a system healthy when nobody is watching it.

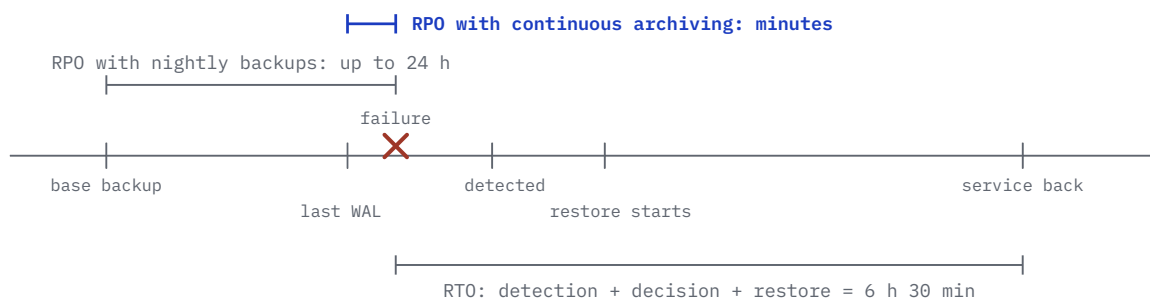
Backups and recovery

3 exercises

A backup you have never restored is a hypothesis. What you actually own is a recovery time and a recovery point, and you only know them by measuring a real restore, at real size, on a schedule.

Two numbers, agreed with the business

Recovery point objective is how much data you can afford to lose: the gap between the last recoverable moment and the failure. **Recovery time objective** is how long you can afford to be down. Both are business decisions with engineering price tags, and both are properties of the *restore*, not of the backup job.



The recovery point is set by how often you capture changes; the recovery time is set by how fast you can put them back, including the minutes spent noticing and deciding. An untested backup gives you neither number, only the belief that you have them.

What restores actually cost

Do the arithmetic before you promise anything. Copying a 2 TB database back at 250 MB/s takes 2.2 hours before anything else happens; add index rebuilds, replaying the write-ahead log since the base backup, verification, and reconnecting the application, and five to six hours is a realistic figure. If your stated RTO is four hours, you do not have a four-hour RTO, you have a wish. Either shorten the restore, by keeping a warm standby or more frequent base backups, or change the promise.

The same arithmetic tells you which technique to use. Nightly logical dumps give an RPO of up to a day, which is fine for a reporting database and catastrophic for payments. Continuous archiving with point-in-time recovery gives an RPO of seconds to minutes and

lets you restore to just before the bad migration ran, which is the failure you'll actually have.

Replicas are not backups

A replica copies your mistakes faithfully and immediately: the `DELETE` without a `WHERE` clause arrives at the standby in milliseconds. Replication protects against a machine dying. Backups protect against everything else, including your own code, your own hands, and an attacker who now has your credentials.

- **Keep copies somewhere a compromise of production cannot reach:** a different account or project, with different credentials, and object-lock immutability so nothing, including an administrator, can delete them before their retention expires.
- **Back up more than the database:** object storage (enable versioning), the encryption keys without which the backup is noise, the infrastructure code and state (chapter 8), secrets, DNS, and any data held only by a third party.
- **Turn on soft deletes and versioning** wherever they exist; most "restore" requests are one table or one file, and these turn a three-hour operation into a two-minute one.
- **Check that backups happen, and that they contain something:** alert on backup age and size. A backup job that has silently failed for three weeks is a standard ingredient of a bad month.

Drill it

Put a restore drill in the calendar, quarterly, and treat the stopwatch as the deliverable. Restore the production database into an isolated environment, run your verification queries, record how long each phase took, and write down what was missing. Teams that do this routinely discover, in order: that the runbook is out of date, that the restore needs a permission nobody has, that the encryption key is in the environment being restored, and that the restore takes three times as long as anyone thought.

A useful side effect: if you build staging or performance-test environments *from* the restore path, it gets exercised every week without a ceremony. The best-tested recovery is one that something else depends on.

How much disaster recovery to buy

STRATEGY	TYPICAL RTO	COST
Backup and restore	hours to a day	Storage only
Pilot light: data replicated, compute off	under an hour	Storage plus a little compute

Warm standby: small live copy, scaled up on failover	minutes	Significant
Active-active across regions	near zero	Double, plus the hardest engineering

Most products should be honest about sitting in the first two rows, and spend the difference on the failures that are far more likely: a bad deploy, a bad migration, a deleted bucket, a corrupted table. Multi-region active-active protects against the rarest scenario while adding daily complexity (*The Engineer's Fleet*, chapters 4 and 7).

COMMON TRAPS

Backups never restored, so the first test is during the incident.

Backups in the same account as production, deletable by the same credentials.

Encryption keys backed up inside the thing you're restoring.

Replicas counted as backups.

Object storage with no versioning, where a delete is final.

An RTO nobody has timed, agreed with customers in a contract.

EXERCISES

EXERCISE 18.1 Set the objectives

Give an RPO and RTO for each, with a one-line justification.

- The invoices database.
- Generated PDF files in object storage.
- The analytics warehouse, rebuilt nightly from the invoices database.
- The internal wiki.

SOLUTION

- RPO: seconds. RTO: an hour or less.** Losing committed invoices is losing money and trust, so continuous archiving plus a standby, and a drill that proves the hour.

- b. **RPO: near zero, RTO: hours**, and mostly by a different mechanism: versioning plus cross-region replication in the object store, because these files are regenerable in principle but expensive to reproduce at volume and sometimes legally required as issued.
- c. **RPO: 24 hours, RTO: a day**. It is derived data; the recovery plan is "rebuild it", and what needs backing up is the code that builds it.
- d. **RPO: a day, RTO: a day**, but check it's actually backed up, because "it's a hosted service" is not a backup strategy, and a wiki is where your runbooks live.

EXERCISE 18.2 Audit the plan

"We take a nightly snapshot of the database, and object storage is replicated by the provider." What is missing, and what would you do first?

SOLUTION

- **No restore has been timed**, so there is no RTO. Do this first: one drill will change the rest of the list.
- **Up to 24 hours of data loss**, which nobody has agreed to. Add continuous archiving if the data matters.
- **Snapshots live in the same account**, so a compromised credential or a mistaken automation can delete them. Copy to another account with immutability.
- **Replication is not versioning**: a deletion or an overwrite replicates too. Turn on versioning and a lifecycle policy for old versions.
- **Nothing covers secrets**, encryption keys, infrastructure state, DNS, or configuration, all of which you need to rebuild.
- **No alerting on backup age or size**, so a silent failure is invisible.

EXERCISE 18.3 Recover from a bad migration

At 14:00 a migration drops a column that still held data. It is noticed at 14:40. The database is 2 TB, with nightly base backups at 02:00 and continuous archiving. Write the recovery plan.

SOLUTION

1. **Stop the bleeding:** pause writes if the missing column is still causing errors, or put the service in read-only mode. Do not run another migration to "fix" it.
2. **Do not restore over production.** Restore a copy to a new instance, to a point in time at 13:59, from the 02:00 base backup plus the archived log.
3. **Extract, don't swap:** copy the dropped column's values out of the restored copy and back into the live database keyed by primary key, in batches, while the service keeps running. Restoring the whole database would throw away four hours of legitimate writes.
4. **Verify** row counts and a sample against the restored copy before declaring success, and keep the restored instance until you have.
5. **Afterwards:** the postmortem asks why a destructive migration shipped in the same release as the code, why nothing blocked a column drop, and how long the restore took against your stated RTO (chapters 10 and 17).

CHECKLIST

- ☐ When did you last restore from a backup, and how long did it take?
- ☐ Can production credentials delete your backups?
- ☐ Are keys, secrets, infrastructure state and object storage covered as well as the database?
- ☐ Do you alert on backup age and size?
- ☐ Do your stated RPO and RTO match measured numbers?

GO DEEPER

DOCS Continuous archiving and point-in-time recovery PostgreSQL

BOOK Data integrity: what you read is what you wrote Google SRE book

REFERENCE Disaster recovery options in the cloud AWS · the four strategies, with costs

Cost as an engineering constraint

3 exercises

The bill is a design document: it describes your architecture in money. The number to manage is not the total but the cost per unit of business value, and the biggest savings come from waste and shape, not from haggling.

Cost per invoice, not cost per month

A total that grows is not automatically a problem; a total that grows faster than the business is. Pick the unit your product is measured in — invoices sent, orders, active tenants, gigabytes processed — and track cost per unit every month. A cost that scales with usage is an input to pricing. A cost that scales with *time*, or with the number of environments, or with how long ago someone left the company, is waste.

Industry surveys put wasted cloud spend at roughly a quarter to a third of the bill, and Kubernetes clusters routinely run at around 40% utilisation against what workloads have reserved. That is not a moral failing; it is what happens when requests are set by copy-paste and nobody is measured on it.

Read the bill like a stack trace

Here is a month for the invoicing product, sending two million invoices:

LINE	COST	SHARE	THE FIX
Compute, 44 on-demand instances	\$9,800	36%	Commit the baseline, run workers on spot
Log ingestion, 346 GB a day	\$5,200	19%	The sampling policy from chapter 12
Managed Postgres, primary and standby	\$3,100	11%	Right-size; it's buying real availability
Non-production environments, running 24/7	\$2,600	10%	Stop them outside working hours
Object storage, all in the standard tier	\$2,400	9%	Lifecycle old PDFs to cold storage
NAT gateway and cross-zone transfer	\$1,900	7%	Cache the external fetch; keep traffic zone-local

Metrics and traces	\$1,400	5%	Cardinality and sampling (chapters 11 and 13)
Orphaned volumes and old snapshots	\$700	3%	Delete them; then stop creating them

Applying the obvious fixes — 35% off compute from commitments and spot, 70% off logs, half the storage tiered, 60% off network, non-production stopped at night, orphans deleted — takes \$27,100 a month to about \$15,300: a **44% cut**, and the cost per invoice falls from 1.35 cents to 0.76. None of it required a negotiation, and none of it made the product less reliable.

Two of those lines deserve emphasis, because they surprise people. **Observability often costs more than the database**, and grows with traffic and with every new label. And **non-production environments** quietly cost as much as a small production, because nobody turns them off.

Where the money actually hides

- **Reserved but unused capacity.** A pod requesting 4 GB and using 600 MB reserves the 4 GB (chapter 5). Right-size from measured usage, and use the autoscaler's recommendations rather than intuition.
- **Idle time.** Development clusters at 3am, CI runners waiting, instances kept for a spike that happens twice a year.
- **Data transfer.** NAT gateways, cross-zone chatter, egress to the internet, and traffic to managed services over public paths (chapter 7).
- **Storage nobody owns:** old snapshots, unattached volumes, logs with infinite retention, and the copy of the production database someone made in 2024.
- **Per-request pricing** on managed services, which is cheap until a retry loop turns it into the largest line on the bill.
- **Engineer time**, which is more expensive than nearly all of the above, and is why "we could save \$200 a month by self-hosting this" is usually the wrong trade.

Buying efficiency, not fragility

Every saving has a reliability price, and the point of the earlier chapters is to know which ones are worth paying. Running at 90% utilisation saves compute and costs you the latency curve from chapter 3. Removing the standby saves a database and costs you the failover in chapter 18. Deleting the third zone saves cross-zone traffic and costs you the failure domain in chapter 7. Spot instances are free money for queue workers and a bad idea for your only stateful node.

The discipline is to make the trade explicitly and write it down, ideally against the error budget: if a saving would predictably spend more budget than you have (chapter 15), it isn't a saving.

Making it somebody's job

Costs drift upwards unless someone watches, and the mechanisms are boring: tag everything by team and service so the bill can be split; put a per-service cost panel next to the reliability dashboards; alert on anomalies, because a 40% jump overnight is usually a bug, a runaway retry loop or a leaked credential, not growth; review the top five lines monthly; and give teams a budget they own rather than a rule they resent. Engineers respond to a number they can see.

COMMON TRAPS

Optimising the total while the per-unit cost gets worse.

Cutting capacity headroom to save money, then paying in incidents.

Ignoring observability spend until it exceeds the infrastructure it observes.

No tags, so nobody can tell whose spend it is.

A one-off cleanup with nothing to stop the waste returning in six months.

Spending a month of engineering to save a hundred pounds a month.

EXERCISES

EXERCISE 19.1 Find the waste

The bill jumped 30% this month with no traffic growth. Name five things you'd check, in order, and say how you'd tell them apart.

SOLUTION

1. **What changed in the bill itself:** group by service and by day to find when it started and which line moved. The shape of the jump usually names the cause.
2. **Data transfer**, which moves when someone adds a call in a hot path, moves a service across zones, or starts fetching a large file per request.

3. **Log and metric volume:** a new debug log line, or a label with an identifier in it, can double a bill in a day (chapters 11 and 12).
4. **A retry storm or a loop** against a per-request-priced service; correlate with that service's own request count, not with your traffic.
5. **Left-behind resources:** a load test that never stopped, a cluster from a spike, snapshots from a migration, an environment created for a demo.

And check for a price or commitment change: sometimes the architecture is innocent and a discount expired.

EXERCISE 19.2 Choose the purchase model

For each, choose on-demand, spot, or a one-year commitment, and say why.

- a. The baseline 20 instances that run every hour of every day.
- b. Queue workers that process PDF rendering and can be interrupted.
- c. The primary database.
- d. Capacity for the January invoicing peak, which is four times normal for three days.

SOLUTION

- a. **Commitment.** It's the definition of predictable baseline, and commitments are the largest single discount available. Commit to somewhat less than the baseline so you keep flexibility.
- b. **Spot.** They are interruptible by design; make sure the work is idempotent and the shutdown handler drains properly (chapter 1).
- c. **On-demand or committed, never spot.** An interruption here is a failover, and you don't want one for a few percent.
- d. **On-demand,** or spot for the stateless part, added by an autoscaler with the capacity pre-warmed before the peak, because an autoscaler that reacts in ten minutes doesn't help with a spike that arrives in two (chapter 3).

EXERCISE 19.3 Say no to a saving

Someone proposes saving 30% of compute by removing the third availability zone and running the remaining two at 85% utilisation. Write the counter-argument with numbers.

SOLUTION

Two effects, both quantifiable. With three zones, each carries a third of the load and the survivors need 50% more capacity to absorb one failing; with two, each must be able to carry *everything*, so the protection you're paying for is more expensive per zone, not less (chapter 3). And at 85% utilisation the queueing curve puts response time at nearly seven times the service time, against twice at 50%, so latency degrades before any failure happens.

Put it in budget terms: if losing a zone now means an outage rather than a degradation, one such event spends the month's error budget several times over. Then offer the alternative that gets most of the saving without the risk: right-size the instances, put interruptible work on spot, commit the baseline, and cut the log bill, which in the worked example above was a bigger line than the third zone would ever be.

CHECKLIST

- ☐ Do you know your cost per invoice, order or tenant, and its trend?
- ☐ Is every resource tagged with a team and a service?
- ☐ Would you find out this week if the bill jumped 40% overnight?
- ☐ Are non-production environments switched off when nobody is using them?
- ☐ Has any saving ever been rejected for costing reliability? If not, look again.

GO DEEPER

REFERENCE [What is FinOps](#) [FinOps Foundation](#)

DOCS [Resizing running pods](#) [Kubernetes · right-sizing without restarts](#)

BOOK [Cloud FinOps, 2nd edition](#) J.R. Storment & Mike Fuller

The operating discipline

3 exercises

Everything in this volume only works if it is somebody's job. The unglamorous practices — ownership, runbooks, readiness reviews, drills, and steadily deleting toil — are what decide whether the clever parts survive contact with a busy quarter.

Toil, and why it grows

Google's SRE practice defines **toil** precisely: work that is manual, repetitive, automatable, tactical, without enduring value, and that grows at least linearly with the size of the service. Restarting a stuck worker is toil. Copying a report by hand every Monday is toil. Approving the same routine change is toil. It is not the same as overhead, and it is not the same as hard work; the distinguishing feature is that it scales with the system, so a team whose toil is unchecked eventually spends all its time keeping yesterday running.

The rule of thumb is to keep it under half of the team's time, and the practice is dull: **count it**, in a shared list, for a month. Then automate the top item, and only the top item, each month. An alert whose answer is always the same command is the easiest place to start, because it's the one costing you sleep (chapter 16).

Ownership, written down

Every service needs a team whose name is on it, and a catalogue entry that anyone can find in the middle of the night: what it does, who owns it, where its dashboards and runbooks are, what it depends on, what depends on it, and where its code and deploy pipeline live. Unowned services accumulate in every organisation, and they are where incidents go to become long ones.

The same applies to on-call: if the team that builds a service also carries its pager, the feedback loop that makes systems reliable exists. If it doesn't, someone else pays for decisions they can't change, which is both unfair and ineffective.

Runbooks that survive contact

A runbook is not documentation, it's an instruction sheet for someone tired and worried. Write it during or immediately after an incident, while you still remember the commands. Keep each one short: what this alert means, how to confirm it, how to mitigate, how to escalate, and what *not* to do. Link it from the alert (chapter 14), keep it next to the code so it changes with the code, and test it by having someone who didn't write it follow it during a drill. A runbook nobody has followed is a wiki page.

The production readiness review

Before a new service takes real traffic, walk one checklist. It takes an hour and saves quarters.

- **Owned:** a named team, an on-call rotation, an entry in the catalogue.
- **Observable:** RED metrics, structured logs with trace IDs, traces, a dashboard that answers "is it healthy" (chapters 11–14).
- **Objectives:** at least one SLI and SLO, with an alert on burn rate and a runbook behind it (chapter 15).
- **Deployable:** one artefact, automated rollout, health-gated, rollback tested (chapters 9 and 10).
- **Bounded:** resource requests and limits, timeouts on every dependency, retries with budgets, graceful shutdown (chapters 1 and 5).
- **Recoverable:** backups and a timed restore, or an explicit statement that it holds no state worth keeping (chapter 18).
- **Secured:** least-privilege identity, no long-lived credentials, secrets from a manager (chapters 6 and 7).
- **Affordable:** an estimate of its monthly cost, and tags so it appears on the bill (chapter 19).

Change, without ceremony

Heavyweight change-approval boards mostly add delay: they batch changes, which makes each one riskier, and they ask people with less context to approve work they can't evaluate. The alternative that actually reduces risk is the one this volume has been building: small changes, automated checks, staged rollout, fast rollback, and a record of what changed and when, annotated on the same dashboards you watch during an incident. Where you do want a brake, tie it to the error budget rather than the calendar: freezes triggered by a number are negotiated once, not every time (chapter 15).

Practise before you need to

Schedule the drills you know you need: a restore (chapter 18), a failover, a rollback, an on-call handover, and a game day where you break something in a controlled way and run the incident process for real. Rotate who leads them, and use them to onboard: shadowing a drill teaches more in an afternoon than a month of documents. Write down what didn't work, because the drill's findings are the point, not the successful recovery.

If you have just inherited a system

A closing order of operations, for the common case where all of this needs to exist and doesn't yet:

- 01 **Find out what breaks and who is woken by it.** Read the last three months of incidents and pages.
- 02 **Make one dashboard** that answers whether the product is working, and one alert that fires when it isn't.
- 03 **Make deploys reversible.** Everything else is easier once you can undo.
- 04 **Time a restore.** Whatever the result, you will know something important.
- 05 **Write down the three things you did last month by hand,** and automate the most frequent one.
- 06 **Set one SLO** on the most important journey, and start showing the budget to whoever prioritises the work.
- 07 **Then** containers, clusters, pipelines and everything else in this volume, in whatever order the pain dictates.

COMMON TRAPS

Automating the interesting toil rather than the frequent toil.

Runbooks nobody has followed, written by the person who didn't need them.

A readiness checklist as a gate, rather than a conversation with an outcome.

Ceremony instead of safety: approvals that delay without reducing risk.

Drills cancelled because the quarter is busy, which is precisely when the system is least understood.

Reliability as a separate team's problem, so the people who can fix it never feel it.

EXERCISES

EXERCISE 20.1 Rank the toil

A team logs a month of manual work: restarting a stuck importer (11 times, 10 min each), manually approving refunds over £500 (60 times, 3 min), rotating an API key (once, 2 hours), rebuilding a broken staging environment (3 times, 90 min), and answering "why did this invoice fail" questions from support (25 times, 15 min). What do you automate first?

SOLUTION

Time first: importer 110 min, refunds 180 min, key rotation 120 min, staging 270 min, support questions 375 min. But raw time isn't the whole answer.

- **Support questions (375 min)** are the biggest, and the cheapest to fix: give support a view that shows an invoice's status and last error, using the trace ID (chapter 12). This also removes an interruption, which costs more than its minutes.
- **Staging rebuilds (270 min)** point at infrastructure that isn't reproducible (chapter 8); fixing it pays again in disaster recovery.
- **Refund approvals (180 min)** may be a control someone requires; automate the mechanics, not the decision.
- **The importer (110 min)** is the one waking people up. Even though it isn't the largest, fix the crash or make the restart automatic, because pages cost far more than their minutes (chapter 16).
- **Key rotation (120 min)** happened once, but it's the one that becomes an emergency at the worst moment; move to short-lived credentials (chapter 7).

EXERCISE 20.2 Review a new service

A team wants to launch a "customer portal" service next week. Run the readiness review: what are the five questions you'd refuse to launch without answers to?

SOLUTION

1. **Who is on call for it, and what do they do when it pages?** No owner, no launch.
2. **What does it depend on, and what happens when each dependency fails?** Timeouts, fallbacks, and the resulting user experience.
3. **How do we know it's working?** One dashboard, one objective, one alert that would have caught last month's outage in the neighbouring service.
4. **How do we undo a release?** Demonstrated, not described, including any schema change.
5. **What data does it hold, and how do we get it back?** Backups, a timed restore, and who can delete them.

Everything else — cost estimates, capacity, security review, runbook quality — is a finding to fix in the first weeks, not a reason to hold the launch, as long as these five have answers.

EXERCISE 20.3 Plan the first ninety days

You've inherited a service nobody owns: no dashboards, no alerts, deploys by hand from a laptop, one shared root credential, and a database whose backups nobody has restored. You have ninety days and your normal workload. What do you do, in order, and what do you deliberately not do?

SOLUTION

Weeks 1–2: inventory and visibility. What it does, what it talks to, where it runs. Add basic RED metrics and one dashboard. Write down everything that broke recently.

Weeks 3–4: stop the bleeding. One alert on the user-visible symptom, with a runbook. Replace the shared root credential with per-person access and a workload identity. Take a backup and *time a restore*.

Weeks 5–8: make change safe. Move the build and deploy into a pipeline, deploy the same artefact everywhere, get rollback working and use it once on purpose. Put the existing infrastructure into code, even imperfectly.

Weeks 9–12: make it sustainable. One SLO and a burn-rate alert, an on-call rotation with a handover, a catalogue entry, and the top piece of toil automated.

Deliberately not: migrating to Kubernetes, rewriting anything, adding tracing before there are metrics, or a multi-region plan. All of those are reasonable later and none of them help with the failure modes you have now: you can't see it, you can't safely change it, and you can't recover it.

CHECKLIST

- ☐ Does every service have a named owner and an on-call rotation?
- ☐ Do you measure toil, and did you delete some this month?
- ☐ Is there a runbook behind every alert, followed by someone who didn't write it?
- ☐ Does a new service pass a readiness review before it takes traffic?

☐ When is the next drill, and who is running it?

GO DEEPER

BOOK Eliminating toil Google SRE book

BOOK Production readiness reviews Google SRE Workbook

BOOK The Phoenix Project Kim, Behr & Spafford · why this is an organisational problem

One per chapter, each linked back to it. Read them before a launch, before an on-call handover, and whenever you inherit something that has been running unattended.

-
- 01 Give every process a user, explicit limits and a SIGTERM handler, and know what happens when each limit is reached.
-
- 02 Diagnose by method: utilisation, saturation and errors for every resource, then ask whether the slow thing is running or waiting.
-
- 03 Treat latency as a distribution, never average percentiles, and keep peak utilisation below the knee of the curve.
-
- 04 Ship minimal images pinned by digest, running as a non-root user, and never patch a container in place.
-
- 05 Set requests honestly, keep liveness probes free of dependencies, and make rollouts add capacity before removing it.
-
- 06 Validate configuration at start-up, roll it out as carefully as code, and prefer credentials that expire.
-
- 07 Put nothing but the load balancer and the NAT gateway in a public subnet, and give every workload a short-lived identity.
-
- 08 Declare infrastructure in code, read the plan like a diff, and never let a destroy of a stateful resource through unread.
-
- 09 Answer "is this safe to ship" in ten minutes, build the artefact once, and treat a flaky test as a defect.
-
- 10 Deploy small, gradually, observably and reversibly, and keep migrations compatible with the previous release.
-
- 11 Instrument rate, errors and duration with bounded labels, and put histogram buckets where your objective is.
-
- 12 Log structured events carrying a trace ID, redact centrally, and sample deliberately rather than by accident.
-

-
- 13 Propagate trace context everywhere, including queues and jobs, and always keep the errors and the slow requests.
-
- 14 Page only for symptoms a user would notice, always with a runbook, and pair a long alerting window with a short one.
-
- 15 Set objectives from user journeys, agree the budget policy before you need it, and let the number decide when to stop shipping.
-
- 16 Mitigate before diagnosing, name a commander, and communicate on a cadence even when there is nothing new.
-
- 17 Write postmortems blamelessly, list contributing factors rather than a root cause, and finish four actions instead of listing fifteen.
-
- 18 Treat the restore as the product: time it, keep copies out of reach of production, and drill it quarterly.
-
- 19 Track cost per unit of business value, and never buy a saving with your error budget.
-
- 20 Give every service an owner, a runbook and a readiness review, and delete one piece of toil a month.
-

READING LIST

The books behind this volume, and the chapters each one deepens. Google's three SRE books are free to read online and are the closest thing this field has to a canon; start with the first, and treat the workbook as the practical companion.

BOOK	AUTHOR	DEEPENS
Site Reliability Engineering	Betsy Beyer et al., Google	Ch. 14–17, 20
The Site Reliability Workbook	Betsy Beyer et al., Google	Ch. 14–15, 20
Systems Performance, 2nd edition	Brendan Gregg	Ch. 2–3
BPF Performance Tools	Brendan Gregg	Ch. 2
How Linux Works, 3rd edition	Brian Ward	Ch. 1
Kubernetes Up & Running, 3rd edition	Burns, Beda, Hightower & Evenson	Ch. 4–5
Terraform: Up & Running, 3rd edition	Yevgeniy Brikman	Ch. 8
Continuous Delivery	Jez Humble & David Farley	Ch. 9–10
Observability Engineering	Majors, Fong-Jones & Miranda	Ch. 11–13
Implementing Service Level Objectives	Alex Hidalgo	Ch. 15
Database Reliability Engineering	Laine Campbell & Charity Majors	Ch. 18
Cloud FinOps, 2nd edition	J.R. Storment & Mike Fuller	Ch. 19
The Phoenix Project	Kim, Behr & Spafford	Ch. 20

Compiled September 2026 as the companion to stage 12 of The Engineer's Syllabus, following *The Engineer's Craft*, *The Engineer's Shipyard* and *The Engineer's Fleet*. Research drew on the Google SRE books, the workbook's burn-rate alerting chapter, Brendan Gregg's USE method and Linux performance tooling, Gil Tene's coordinated omission, the systemd manual pages, Docker's build guidance, in-place pod resize in Kubernetes 1.35, the retirement of ingress-nginx, Cloudflare's November 2025 postmortem, Terraform's S3 backend and OpenTofu's documentation, GitHub's Actions hardening guide, Prometheus native histograms, OpenTelemetry's specification status, DORA's four keys and the FinOps Foundation.

The examples here were run before publication. The systemd unit's directives were checked against upstream documentation, section by section; the Kubernetes manifests validate against the official strict schemas; the workflow passes actionlint; the Terraform parses; the graceful-shutdown worker was sent a real SIGTERM mid-job; the Prometheus instrumentation, the histogram-quantile comparison, the structured logger and the OpenTelemetry trace produced the output shown; and the load-generator comparison, the queueing, capacity, canary, error-budget, restore and cost arithmetic were all computed rather than estimated. Figures attributed to outside sources are cited above. Commands and outputs illustrating Linux diagnosis are illustrative. Re-checked in October 2026 with Python 3.14.7: every self-contained example ran, and the claims about the graceful worker, the two load generators, configuration and structured logging were tested again and held.