

COMPANION VOLUME • STAGE 11 OF THE ENGINEER'S SYLLABUS

The Engineer's Fleet

One machine is a ship. Many machines are a fleet: coordinated by messages that can be lost, delayed and reordered, keeping time with clocks that disagree, and carrying on while some of them sink. This volume is about making many machines behave like one dependable system, and about knowing when not to try.

CHAPTERS

20

CODE EXAMPLES

30

EXERCISES

60

STAGE BUDGET

280 h

Contents

How to use this volume	4
Part I · Why it's hard	5
Chapter 01 · Why distribute, and what it costs	6
Chapter 02 · Failure, the network and the fallacies	11
Chapter 03 · Time, clocks and ordering	17
Part II · Data on many machines	23
Chapter 04 · Replication	24
Chapter 05 · Partitioning and sharding	30
Chapter 06 · Consistency models	36
Chapter 07 · CAP and PACELC, stated correctly	41
Part III · Agreement	46
Chapter 08 · Consensus and Raft	47
Chapter 09 · Leases, locks and fencing	52
Chapter 10 · Transactions across services	57
Part IV · Messages and streams	63
Chapter 11 · Queues, logs and delivery guarantees	64
Chapter 12 · Event-driven architecture, CQRS and event sourcing	69
Chapter 13 · Batch and stream processing	74
Part V · Surviving failure	79
Chapter 14 · Timeouts, retries and overload	80
Chapter 15 · Load balancing, discovery and isolation	85
Chapter 16 · Caching at scale	90
Part VI · Architecture and design	97
Chapter 17 · Architecture styles	98
Chapter 18 · Domain-driven design	104
Chapter 19 · System design method	110

Chapter 20 · Testing and verifying	117
The twenty rules	124
Reading list	126

The chapters follow the same shape as the rest of the series: the idea in one sentence, worked examples, the traps, then exercises with folded solutions, a checklist and further reading. This is the deepest stage in the syllabus, and the one whose ideas take longest to settle, so expect to reread.

Prerequisites matter here. Operating systems, networking and databases (stages 05, 06 and 08) are assumed throughout, and so is some experience of running a real product, which is why this stage comes after *The Engineer's Shipyard*. The ideas land much harder once you've been woken by a system misbehaving.

The running example has grown up. The invoicing app from the previous volume now serves two million teams on four continents, takes payments around the clock, and has discovered that one database in one region is no longer enough. Other examples appear where they explain a mechanism better.

Pair it with *Designing Data-Intensive Applications*, whose second edition appeared in 2026. That book gives depth; this one gives working intuition and practice. The stage's deliverables build on both: implement Raft in the MIT 6.5840 labs, build the distributed rate limiter designed in chapter 19, and write design documents for five well-known systems. The self-contained examples here were run before publication to confirm they behave as described.

THE ONE QUESTION

Every mechanism in this volume exists because of three facts. Messages can be lost, delayed, duplicated or reordered. Clocks on different machines disagree. And machines fail independently while the rest carry on, often without anyone being sure which ones have failed.

Before trusting any design, walk through each message in it and ask: **what happens if this never arrives, arrives twice, or arrives after the next one?** Then ask what happens if the machine that sent it dies straight afterwards. Most distributed bugs are an unasked version of those questions.

PART I

Why it's hard

Three chapters on the ground rules: why you'd put a system on many machines at all, what the network does to your assumptions about failure, and why there's no such thing as "now" across machines.

Why distribute, and what it costs

3 exercises

Distribute a system only for a reason you can name: more load than one machine can carry, more availability than one machine can give, or users too far from one machine. Every one of those gains is paid for in complexity, and the bill never stops arriving.

Three good reasons

- **Scale.** More requests, or more data, than one machine can handle.
- **Availability.** Any single machine will eventually fail. Keeping copies elsewhere lets the service continue.
- **Latency.** Light in optical fibre covers about 200 kilometres per millisecond. London to Sydney is about 17,000 kilometres, so no request can make that round trip in under about 170 milliseconds, and real routes take longer. If users are on the other side of the world, some of your system has to be near them.

There are organisational reasons too, such as letting separate teams own and deploy separate parts, and legal ones, such as keeping European customers' data in Europe. Those are real, but notice that none of the reasons is "it's more interesting" or "we might be big one day".

Scale up before you scale out

A single modern server can have hundreds of cores, terabytes of memory and storage that serves millions of operations a second. A well-tuned relational database on one large machine, with a couple of read replicas and a cache in front, carries far more traffic than most products will ever see. Scaling up is boring and expensive per machine. Scaling out is exciting and expensive in engineering time, forever.

A useful map of the options is the **scale cube** from Abbott and Fisher's *The Art of Scalability*. Scale along the X axis by running identical copies behind a load balancer, which is easy for stateless services. Scale along the Y axis by splitting the system by function into services (chapter 17). Scale along the Z axis by splitting the data itself, so each machine holds part of it (chapter 5). The X axis is cheap. The other two are where the difficulty in this volume lives, because they involve *state*.

Orders of magnitude

Distributed design is mostly arithmetic about delays. Memorise these rough figures for current hardware, and then measure your own, because they vary.

OPERATION	ROUGHLY	RELATIVE TO A MEMORY READ
Read from main memory	100 ns	1×
Random 4 KB read from an NVMe SSD	20 µs	200×
Round trip within one data centre	0.5 ms	5,000×
Round trip between nearby regions	10–70 ms	100,000× or more
Round trip across an ocean	150–250 ms	2,000,000×

The lesson is in the right-hand column. A function call that becomes a network call inside a data centre gets thousands of times slower; one that crosses an ocean, millions. That's why "just split it into services" so often makes things slower, and why chatty conversations between regions are the first thing to design out.

Availability is arithmetic too

Availability is usually quoted in "nines". Each extra nine cuts the downtime you're allowed by a factor of ten.

AVAILABILITY	DOWNTIME PER YEAR	PER MONTH
99%	3.65 days	7.3 hours
99.9%	8.8 hours	44 minutes
99.99%	53 minutes	4.4 minutes
99.999%	5.3 minutes	26 seconds

Dependencies on the critical path *multiply*: if a request needs ten services that are each up 99.9% of the time, the request succeeds only about 99% of the time. Redundancy works the other way: two independent copies that are each up 99% of the time give 99.99% together. But only if they fail independently, and they rarely do. Copies share a deploy pipeline, a configuration system, a region, a bug. Correlated failure is why "we have three replicas" so often turns out to mean "we had one failure, three times".

EXAMPLE `availability.py`

```
def series(*availabilities: float) -> float:
    """Every component is needed: availabilities multiply."""
    total = 1.0
```

```

for a in availabilities:
    total *= a
return total

def parallel(availability: float, copies: int) -> float:
    """Any one copy is enough, assuming failures are independent."""
    return 1 - (1 - availability) ** copies

series(*[0.999] * 10)    # 0.99000: ten "three nines" dependencies make two nines
parallel(0.99, 2)        # 0.9999: two "two nines" copies make four, if independent

```

What distribution costs

The rest of this volume is the invoice. Some parts will fail while others carry on (chapter 2). There's no shared clock (chapter 3). Copies of data disagree (chapters 4 to 7). Agreeing on anything needs careful protocols (chapters 8 to 10). And the operational burden grows with every moving part: deploying, observing, debugging and paying for many machines instead of one. Distribute when the alternative is worse. It often isn't, yet.

COMMON TRAPS

Distributing for the résumé. Complexity chosen for interest rather than need.

Counting replicas, not failure modes. Three copies behind one bad config push is one copy.

Forgetting the multiplication. Each synchronous dependency lowers the ceiling on your availability.

Chatty calls across regions. Three sequential calls across an ocean cost most of a second.

Scaling out a problem that indexing would solve. Check the query plan before you buy more machines.

EXERCISES

EXERCISE 1.1 Multiply the dependencies

"Pay invoice" needs the authentication service (99.95%), the database (99.99%) and the payment provider (99.9%). A confirmation email is sent afterwards, through a provider that's up 99.5% of the time. What's the availability of paying an invoice, how much downtime is that per year, and what does the email teach you?

SOLUTION

$0.9995 \times 0.9999 \times 0.999 \approx 0.9984$, or 99.84%: about 14 hours of failed payments a year, dominated by the least available dependency.

The email doesn't appear in the calculation because it's sent by a background job (*The Engineer's Shipyard*, chapter 12). If it were sent inside the request, the availability would drop to about 99.34%, and a flaky email provider would stop people paying. Moving non-essential work off the critical path is the cheapest availability improvement there is.

EXERCISE 1.2 Budget the latency

Users in Sydney must see the invoice list within 300 milliseconds. Every server is in northern Virginia, about 15,600 kilometres away, and the page makes three sequential API calls. What's wrong, and what are your options?

SOLUTION

The physical minimum round trip is about $2 \times 15,600 \div 200 \approx 156$ milliseconds, and real routes take 200 or more. Three sequential calls cost 600 milliseconds before any server does any work, so the budget is impossible.

- Collapse the three calls into one, which alone gets close to budget.
- Serve static assets from a CDN edge in Australia.
- Render the page on a server in the region, backed by a read replica there, so only writes cross the ocean.
- If Australian teams are a large market, give each team a home region and keep its data there (chapter 5).

EXERCISE 1.3 Should we distribute?

The invoicing app's database runs at 30% CPU at peak, holds 400 GB, and takes about 2,000 writes a second at its busiest. The team wants to shard it now "so we're ready". What do you recommend?

SOLUTION

Not yet. There's plenty of headroom, and sharding would cost months of work and add permanent complexity to every query and migration. Instead:

- Add a synchronous standby in another availability zone. The real risk today is losing the only machine, not outgrowing it.

- Move reporting to a read replica and cache the hottest reads.
- Archive or partition old invoices within PostgreSQL.
- Agree on triggers that start sharding work early enough: sustained CPU above about 60%, storage growth heading past what one machine can hold within a year, or write rates approaching what you've load-tested.
- Meanwhile, make every query include the team id, so sharding by team later is a migration rather than a rewrite.

CHECKLIST

- ☐ Can you name the reason for distributing: scale, availability, latency or law?
- ☐ Have you used up the simpler options: a bigger machine, replicas, caching, indexes?
- ☐ Do you know the latency of every hop, especially across regions?
- ☐ Have you multiplied the availability of everything on the critical path?
- ☐ Are your redundant copies really independent of each other?

GO DEEPER

- | | |
|-----------|---|
| BOOK | Designing Data-Intensive Applications, 2nd edition Martin Kleppmann & Chris Riccomini, 2026 |
| <hr/> | |
| COURSE | Distributed Systems lecture notes Martin Kleppmann, Cambridge · with the videos |
| <hr/> | |
| REFERENCE | Latency numbers everyone should know Google SRE |
| <hr/> | |
| BOOK | Understanding Distributed Systems Roberto Vitillo · the most approachable introduction |

Failure, the network and the fallacies

3 exercises

In a distributed system, any part can fail while the rest keeps running, and you often can't tell a slow machine from a dead one. Design for that from the first line, because the network will otherwise teach it to you at the worst possible moment.

Partial failure

On one machine, failure is mostly all-or-nothing: the process crashes and everything stops. Across machines, some parts fail and others don't, and no participant has a complete picture. Send a request to another service and wait. Any of these may have happened:

- The request was lost before it arrived.
- It arrived, and the server crashed before doing anything.
- It arrived, was fully processed, and the response was lost.
- It's still being processed, slowly, perhaps behind a garbage-collection pause.

From the sender's side, all four look identical: silence. That single fact is the root of most of this volume. It's why operations need idempotency keys, why "exactly once" is so hard, and why you can't simply ask whether a machine is dead.

The eight fallacies

In the 1990s, Peter Deutsch and colleagues at Sun Microsystems listed the false assumptions that programmers new to distributed systems invariably make; James Gosling later added the eighth. Every one of them is still being made today.

FALLACY	REALITY	WHAT IT MEANS FOR YOUR CODE
The network is reliable	Packets drop, connections reset, links partition	Every call needs a timeout, a retry policy and idempotency
Latency is zero	Each hop costs from half a millisecond to hundreds	Chatty interfaces and N+1 calls between services are slow
Bandwidth is infinite	Links saturate; cross-region traffic costs money	Payload sizes and replication traffic matter
The network is secure	Inside the perimeter is not safe	Authenticate and encrypt service-to-service calls

Topology doesn't change	Instances come and go with every deploy and autoscale	Discover services; never hard-code addresses
There is one administrator	Many teams, vendors and configurations	Version everything; coordinate changes
Transport cost is zero	Serialisation burns CPU; egress is billed	Cost is part of the design
The network is homogeneous	Mixed versions, protocols and hardware	Stay compatible across a rolling deploy

Failure models

To reason about a protocol, you have to say what kinds of failure it tolerates. The common models, from mildest to worst:

- **Crash-stop.** A node halts and never returns.
- **Crash-recovery.** A node halts and may come back later, with whatever it wrote to disk but nothing that was in memory. This is what real servers do, and most protocols in this volume assume it.
- **Omission.** Messages are lost.
- **Byzantine.** A node behaves arbitrarily: it lies, sends conflicting messages, or is controlled by an attacker. Systems inside one organisation usually assume this doesn't happen; blockchains and some aerospace systems must not.

Timing needs a model too. In a **synchronous** system, messages and pauses have known upper bounds. In an **asynchronous** one, there are no bounds at all. Real systems are **partially synchronous**: usually well-behaved, with bounded delays, but occasionally not, for unbounded stretches. Good protocols stay correct even when timing goes wrong, and only need the good periods to make progress.

Timeouts are guesses

The only practical way to decide that a node has failed is to stop waiting for it. But a timeout can't distinguish "dead" from "slow". Make it too short and you declare healthy-but-busy nodes dead, trigger failovers and retries that add load exactly when the system is struggling, and cause the outage you feared. Make it too long and users wait, and recovery is slow.

Derive timeouts from measured latency, such as a little above the 99.9th percentile, not from round numbers someone typed years ago. Adaptive failure detectors, such as the phi accrual detector used by Cassandra and Akka, go further: they learn the normal distribution of heartbeat intervals and report a *suspicion level* rather than a yes or no.

The underlying impossibility has a famous form, the **two generals problem**. Two armies must attack together, and can only communicate by messengers who may be captured. No protocol can ever make both generals certain they've agreed, because the last messenger might always be lost. The practical response isn't to find certainty. It's to design for its absence: idempotent operations, retries with the same key, and reconciliation afterwards.

Make "unknown" a real outcome

The most useful habit this chapter can give you is to stop treating a timeout as a failure. It's a third outcome, and code should say so.

EXAMPLE outcomes.py

```
from enum import Enum

class Outcome(Enum):
    SUCCEEDED = "succeeded"
    FAILED = "failed"          # the server definitely refused: report it, or fix and retry
    UNKNOWN = "unknown"       # it may or may not have happened: find out, don't guess

def charge(client, request, timeout_s: float):
    try:
        response = client.post("/charges", json=request, timeout=timeout_s,
                                headers={"Idempotency-Key": request["idempotency_key"]})
    except (Timeout, ConnectionLost):
        return Outcome.UNKNOWN, None
    if response.status_code >= 500:
        return Outcome.UNKNOWN, None    # the server may have failed after committing
    if response.status_code >= 400:
        return Outcome.FAILED, response.json()
    return Outcome.SUCCEEDED, response.json()
```

An `UNKNOWN` result must be resolved, never assumed: retry with the same idempotency key, or ask the server about the operation by its id. The user sees "processing", not "failed", because telling someone their payment failed when it went through is how double payments happen.

Partitions happen, and short ones cause long outages

A **network partition** splits machines into groups that can't talk to each other, while each group is healthy on its own. Peter Bailis and Kyle Kingsbury's 2014 survey, *The Network Is Reliable*, collected many of them from real companies: failed switches, misconfigured firewalls, saturated links, cloud network faults. They're rare per link and routine across a large fleet.

GitHub's incident of 21 October 2018 shows how a short partition becomes a long outage. Maintenance cut connectivity between its East Coast data centre and the rest of its network for 43 seconds. In that window, its database orchestration tool, which used Raft to decide leadership among its own nodes, promoted replicas on the West Coast to primaries. When the network returned, writes had landed in both places, and the primaries were now across the continent from the applications using them. Untangling it took 24 hours and 11 minutes of degraded service. The lesson isn't that automatic failover is bad. It's that failover is itself a risky operation, and deciding to fail over because of a brief silence can cost far more than waiting would have.

Gray failure

Not every failure is a clean crash. A disk becomes slow. One network card drops one packet in a hundred. A node answers health checks perfectly while failing half of real requests. Microsoft researchers named this **gray failure**: the system's own failure detectors think everything is fine while users suffer. The defence is to measure what users experience, compare nodes with their peers, and treat an outlier as a suspect even when it passes its checks.

COMMON TRAPS

Timeout means failure. It means you don't know. Resolve it.

Round-number timeouts. Thirty seconds everywhere, unrelated to any measurement.

Health checks that check the process is running, rather than that it can do its job.

Assuming partitions are too rare to matter. Across enough links, they're weekly events.

Failing over on every blip. Automated failover must wait long enough to be sure it's needed.

EXERCISES

EXERCISE 2.1 Name the fallacy

Which fallacy is each team assuming?

- The mobile app makes 40 API calls to draw its home screen.
- The reporting service's configuration contains the database's IP address.
- An internal admin API has no authentication "because it's behind the firewall".

- d. Nightly replication to another region copies every table in full and now takes nine hours.
- e. A new message format is deployed, and during the rollout, old consumers crash on it.

SOLUTION

- a. Latency is zero.
- b. Topology doesn't change.
- c. The network is secure.
- d. Bandwidth is infinite, and transport cost is zero.
- e. The network is homogeneous: during a rolling deploy, several versions always run at once.

EXERCISE 2.2 The request that timed out

A request to move credit from one team's balance to another times out after five seconds. List everything that might have happened on the server, and say what the client should do next.

SOLUTION

- The request never arrived.
- It arrived and failed before committing anything.
- It committed, and the response was lost or delayed.
- It's still running and will commit in a moment.

The client must not assume failure, and must not retry as a new transfer. It should retry with the same idempotency key, which is safe in every case above, or query the transfer's status by a client-generated id. Meanwhile, the interface says "processing". If the question can't be resolved quickly, a reconciliation job settles it later.

EXERCISE 2.3 Choose a timeout

A service answers in 12 ms at the median, 80 ms at the 99th percentile and 400 ms at the 99.9th, with occasional garbage-collection pauses of 1.5 seconds. Callers may retry once, against a different replica. What timeout would you set, and why?

SOLUTION

Around 500 ms, just above the 99.9th percentile. Almost no healthy request is cut off, and a request stuck behind a 1.5-second pause is abandoned early and retried on another replica, which will usually answer in milliseconds.

A 100 ms timeout would retry about 1% of all requests, adding load for no benefit. A 5-second timeout would tie up the caller's threads during every pause. For the pause specifically, hedged requests (chapter 14) are better still. And pass the caller's overall deadline along, so no retry outlives the user's patience.

CHECKLIST

- ☐ Does every remote call have a timeout derived from measured latency?
- ☐ Does code treat "unknown" as its own outcome, resolved by idempotent retries or status queries?
- ☐ Do health checks test the ability to serve, not merely the process?
- ☐ Does automated failover wait long enough to be sure?
- ☐ Do you compare nodes with their peers to catch gray failures?

GO DEEPER

ESSAY The Network Is Reliable Peter Bailis & Kyle Kingsbury, ACM Queue, 2014

REPORT October 21 post-incident analysis GitHub, 2018

PAPER Gray Failure: The Achilles' Heel of Cloud-Scale Systems Huang et al., 2017

REFERENCE Fallacies of distributed computing

Time, clocks and ordering

3 exercises

There is no single "now" across machines. Physical clocks drift and jump, so to decide what happened before what, use logical clocks that capture cause and effect, and treat any comparison between timestamps from different machines with suspicion.

Physical clocks lie

Every server keeps time with a quartz oscillator that runs slightly fast or slow. A drift of 50 parts per million, which is unremarkable, adds up to more than four seconds a day. The Network Time Protocol corrects this by periodically adjusting the clock towards a reference, typically to within a few milliseconds inside a data centre, and much worse across the internet or on an overloaded host. Corrections can make a clock *jump*, including backwards. Virtual machines can be paused and resumed with a stale clock. Leap seconds have caused real outages.

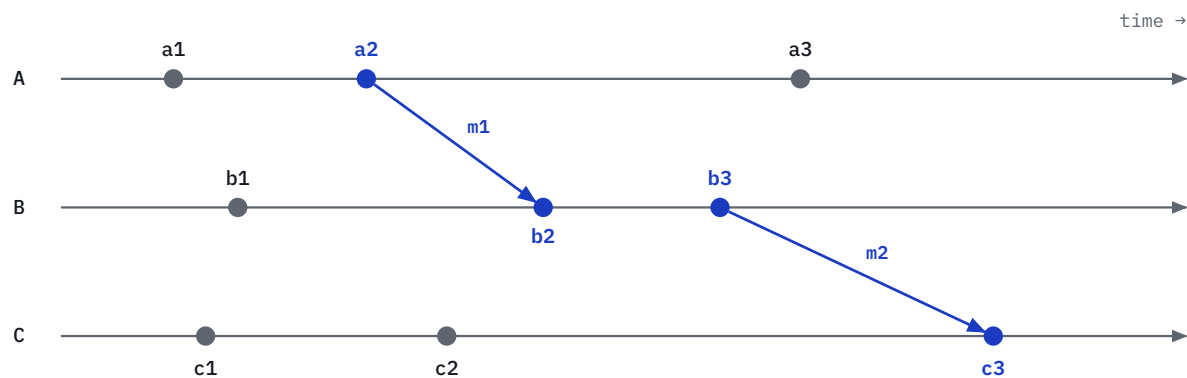
Every platform offers two kinds of clock, and you must use the right one.

- **Wall-clock time**, such as Python's `time.time()`, is the time of day. It's right for showing people when something happened, and it can jump.
- **Monotonic time**, such as `time.monotonic()`, only ever moves forward, from an arbitrary starting point. It's right for measuring durations and enforcing timeouts, and meaningless across machines.

The most expensive misuse of wall clocks is **last write wins**: two replicas each accept a write, and the one with the later timestamp is kept. If one machine's clock runs ahead, its writes keep winning even when they happened first, and the other writes vanish silently. Many databases do exactly this by default.

Happened-before

In 1978, Leslie Lamport showed how to reason about order without any clock at all. Event *a* **happened before** event *b* if they're on the same process and *a* came first; or if *a* is the sending of a message and *b* is its receipt; or if there's a chain of such steps from *a* to *b*. If neither happened before the other, the events are **concurrent**: neither could have influenced the other, whatever the wall clocks say.



Causality travels only along a process line or a message. a2 happened before c3, through m1 and m2. a3 and b2 are concurrent: no chain connects them, so neither could have influenced the other. Exercise 3.1 asks you to timestamp every event.

Lamport clocks

A Lamport clock is a counter on each node. Increment it for every event, attach it to every message, and on receipt set it to one more than the larger of your value and the message's. If *a* happened before *b*, then *a*'s timestamp is smaller. Breaking ties by node id gives a total order that every node agrees on and that respects causality.

EXAMPLE lamport.py

```
class LamportClock:
    def __init__(self):
        self.time = 0

    def tick(self) -> int:
        """A local event, including sending a message."""
        self.time += 1
        return self.time

    def receive(self, sent_time: int) -> int:
        """Receiving a message stamped with the sender's clock."""
        self.time = max(self.time, sent_time) + 1
        return self.time
```

The limitation runs one way only. A smaller timestamp doesn't mean the event happened first; the two events might be concurrent. Lamport clocks can order things consistently, but they can't tell you whether two events conflict.

Vector clocks

A vector clock keeps one counter *per node*. Each node increments its own entry for every event and, on receiving a message, takes the element-wise maximum of the two vectors before incrementing. Comparing two vectors then tells you everything: if every entry of

one is less than or equal to the other's, it happened before; if each has an entry larger than the other's, they're concurrent. That's how systems in the tradition of Amazon's Dynamo detect conflicting writes instead of silently discarding one.

EXAMPLE `vector_clock.py`

```
def increment(clock: dict[str, int], node: str) -> dict[str, int]:
    updated = dict(clock)
    updated[node] = updated.get(node, 0) + 1
    return updated

def merge(a: dict[str, int], b: dict[str, int]) -> dict[str, int]:
    return {n: max(a.get(n, 0), b.get(n, 0)) for n in a.keys() | b.keys()}

def compare(a: dict[str, int], b: dict[str, int]) -> str:
    nodes = a.keys() | b.keys()
    a_le_b = all(a.get(n, 0) <= b.get(n, 0) for n in nodes)
    b_le_a = all(b.get(n, 0) <= a.get(n, 0) for n in nodes)
    if a_le_b and b_le_a:
        return "equal"
    if a_le_b:
        return "before"
    if b_le_a:
        return "after"
    return "concurrent"  # a genuine conflict: keep both, and merge
```

The price is size. A vector grows with the number of participants, which is fine for a handful of replicas and a problem if every client is a participant. Refinements such as dotted version vectors keep it manageable.

Hybrid logical clocks and TrueTime

People want timestamps that look like real time, for reading a snapshot "as of 10:00" or for debugging. **Hybrid logical clocks** combine the physical clock with a logical counter: timestamps stay close to wall-clock time, but when a message arrives from a machine whose clock is ahead, the counter preserves causality anyway. CockroachDB, YugabyteDB and MongoDB all use them.

Google's Spanner takes the opposite approach: make the physical clock trustworthy. Its TrueTime service uses GPS receivers and atomic clocks to give every machine not a time but an *interval*, guaranteed to contain the true time, typically a few milliseconds wide. When a transaction commits, Spanner waits until the interval has passed before revealing the commit. That **commit wait** guarantees that timestamp order matches real-time order across the world. The lesson generalises: bounded clock error lets you use time for ordering, and you pay for the bound by waiting it out.

Practical rules

- Measure durations and timeouts with a monotonic clock.

- Don't decide which write wins by comparing wall-clock timestamps from different machines, unless you've accepted losing data.
- For order within a system, use versions and sequence numbers issued by one authority, such as a database, or logical clocks.
- Store timestamps in UTC, and remember which clock produced them.
- Time-ordered identifiers, such as UUIDv7, sort roughly by creation time. That's convenient for indexes, and not a causal order.
- To follow one request across services, use a trace id (chapter 20), not the timestamps in each service's logs.

COMMON TRAPS

Durations measured with the wall clock. An NTP correction mid-request produces a negative or huge elapsed time.

Last write wins by timestamp. A machine with a fast clock silently erases other people's writes.

Trusting millisecond precision across machines. NTP's error is usually larger than the gaps you're comparing.

Reconstructing incidents from log timestamps across services, as if they shared one clock.

EXERCISES

EXERCISE 3.1 Timestamp the diagram

Every clock starts at 0. Assign a Lamport timestamp to each event in the diagram above, where a2 sends m1 and b3 sends m2. Then say which of these pairs are ordered and which are concurrent: a1 and c3; a3 and b2; b1 and c2.

SOLUTION

A: a1 = 1, a2 = 2, a3 = 3. B: b1 = 1, b2 = max(1, 2) + 1 = 3, b3 = 4. C: c1 = 1, c2 = 2, c3 = max(2, 4) + 1 = 5.

- **a1 and c3:** ordered. a1 comes before a2 on A, and a2, m1, b2, b3, m2 lead to c3.
- **a3 and b2:** concurrent, even though both have timestamp 3. Equal or unequal Lamport timestamps never prove causality either way.

- **b1 and c2:** concurrent. No message ever travels between them.

EXERCISE 3.2 Compare the vectors

For each pair of vector clocks, say whether the first is before, after, equal to, or concurrent with the second.

- `{A: 2, B: 1}` and `{A: 3, B: 1}`
- `{A: 2, B: 1}` and `{A: 1, B: 2}`
- `{A: 1, B: 0, C: 4}` and `{A: 1, C: 4}`
- `{A: 3}` and `{A: 2, B: 1}`

Then: two replicas hold concurrent versions of an invoice draft. What should the system do?

SOLUTION

- Before.
- Concurrent: each is ahead in one entry.
- Equal: a missing entry counts as zero.
- Concurrent: the first is ahead on A, the second on B.

Keep both versions and merge them: for example, take each field from whichever version changed it, union the added line items, and recompute the total. If both versions changed the same field, show the conflict to a person. Picking one version by timestamp would silently throw work away.

EXERCISE 3.3 Find the clock bugs

Explain what's wrong with each snippet, and fix it.

```
# A: measuring how long a request took
started = time.time()
handle(request)
metrics.observe("request_seconds", time.time() - started)

# B: two servers each save an edit; keep the latest
def resolve(edit_a, edit_b):
    return edit_a if edit_a.updated_at > edit_b.updated_at else edit_b
```

SOLUTION

A uses the wall clock for a duration. If NTP adjusts the clock during the request, the measurement can be negative or wildly large. Use `time.monotonic()` for both readings.

B compares wall-clock timestamps from two machines, so whichever server's clock runs ahead wins, even for the earlier edit. Give the invoice a version number that the database increments on every change, and require each save to state the version it started from, as optimistic concurrency does in *The Engineer's Shipyard*. Two edits based on the same version are then detected as a conflict rather than resolved by accident.

CHECKLIST

- ☐ Are durations and timeouts measured with a monotonic clock?
- ☐ Is any conflict resolved by comparing wall-clock times from different machines?
- ☐ Do you use versions or logical clocks where order matters?
- ☐ Do you detect concurrent writes rather than silently discarding one?

GO DEEPER

PAPER Time, Clocks, and the Ordering of Events in a Distributed System Leslie Lamport, 1978

PAPER Spanner: Google's Globally-Distributed Database Corbett et al., 2012 · TrueTime and commit wait

ESSAY Living without atomic clocks Cockroach Labs

PART II

Data on many machines

State is where distribution gets hard. Four chapters: keeping copies of data, splitting data across machines, the contracts that say what a read may return, and what CAP and PACELC actually say about the trade-offs.

Replication

3 exercises

Replication keeps copies of the same data on several machines, for availability, read capacity and nearness to users. Every design trades between how fresh the copies are, how fast a write can be acknowledged, and what happens when the machine taking writes dies.

Single-leader replication

The most common arrangement, used by PostgreSQL, MySQL and most relational databases. One node, the **leader** or primary, accepts all writes and records them in a log. **Followers** apply the leader's log in the same order, so they hold the same data, a little later. Reads can go to any node.

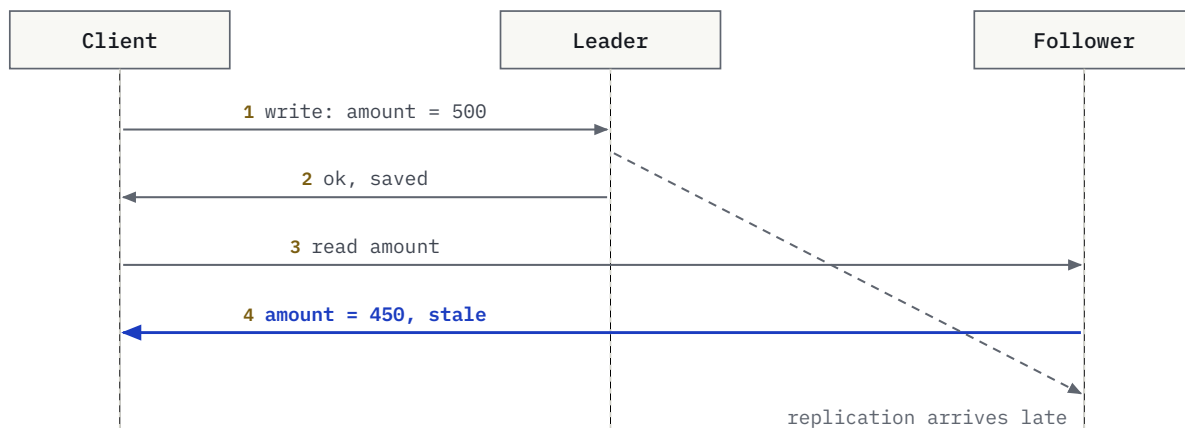
The crucial choice is when the leader tells the client "done".

- **Synchronous:** after at least one follower has the write. An acknowledged write survives the leader's death, but every write waits for a follower, and a slow or dead follower stalls writes.
- **Asynchronous:** as soon as the leader has it. Writes are fast, but if the leader dies, writes it acknowledged and hadn't yet shipped are lost.
- **Semi-synchronous:** one follower synchronous, the rest asynchronous. The usual compromise: no acknowledged write lives on only one machine.

Failover, promoting a follower when the leader dies, is where the trouble lives. You must decide the leader is really dead (chapter 2), choose the most up-to-date follower, redirect every client, and make sure the old leader can't come back and keep accepting writes. Get the last part wrong and you have **split brain**: two leaders, both taking writes, diverging. Chapters 8 and 9 cover how consensus and fencing prevent it.

Replication lag and what users see

With asynchronous followers, reads from a follower may return data that's seconds or, under load, minutes old. Usually harmless, but it produces three specific anomalies that users notice and report as bugs.



The leader acknowledged the write before the follower had it. A read routed to the follower in that gap returns the old amount, and the user concludes their edit was lost.

- **Read-your-writes.** A user saves a change, and the next page load doesn't show it, as above. Fix: read data the user has recently changed from the leader, or remember the log position of their last write and read only from a follower that has reached it.
- **Monotonic reads.** A user refreshes and sees a new comment, refreshes again and it's gone, because the second read hit a follower further behind. Fix: route each user's reads to the same replica, for example by hashing their id.
- **Consistent prefix.** An observer sees a reply before the question it answers, when the two were written to different partitions that replicate independently. Fix: write causally related data to the same partition, or track causal dependencies.

EXAMPLE reads.py · read-your-writes by log position

```

def save_invoice(session, leader, invoice):
    position = leader.execute_returning_log_position(
        "UPDATE invoices SET ... WHERE id = %s", (invoice.id,))
    session["min_position"] = max(session.get("min_position", 0), position)

def load_invoice(session, followers, leader, invoice_id):
    needed = session.get("min_position", 0)
    for follower in followers:
        if follower.replayed_position() >= needed:           # it has seen our last write
            return follower.get_invoice(invoice_id)
    return leader.get_invoice(invoice_id)                   # nobody has caught up yet
  
```

In PostgreSQL, the leader's position comes from `pg_current_wal_lsn()` and a follower's from `pg_last_wal_replay_lsn()`. The same idea works with any database that exposes a log position.

Multi-leader replication

Sometimes more than one node must accept writes: a leader in each region so writes stay local, an app that works offline and syncs later, or a document several people edit at once. Every leader then accepts writes independently, and **conflicts** are inevitable: the same field changed in two places before either change reached the other.

Conflicts must be resolved somehow, and the options get better as they get more work. Last write wins by timestamp throws data away (chapter 3). Merging per field keeps both changes when they touch different fields. Application-specific rules can merge meaningfully, such as taking the union of line items added in two places. And **conflict-free replicated data types**, CRDTs, are data structures designed so that replicas can always merge, in any order, any number of times, and converge on the same value. They power collaborative editors and offline-first apps.

EXAMPLE `crdt.py` · a counter any replica can increment

```
class GCounter:
    """A grow-only counter. Each replica increments only its own slot."""

    def __init__(self, replica_id: str):
        self.replica_id = replica_id
        self.counts: dict[str, int] = {}

    def increment(self, amount: int = 1) -> None:
        self.counts[self.replica_id] = self.counts.get(self.replica_id, 0) + amount

    def value(self) -> int:
        return sum(self.counts.values())

    def merge(self, other: "GCounter") -> None:
        for replica, count in other.counts.items():
            self.counts[replica] = max(self.counts.get(replica, 0), count)
```

Merging takes the maximum of each slot, an operation that gives the same answer whatever the order and however often it's repeated. So replicas that exchange state over an unreliable network, with duplicates and reordering, still converge on the true total. Richer CRDTs handle sets, maps and text in the same spirit.

Leaderless replication

Amazon's 2007 Dynamo paper popularised a design with no leader at all, used today by Cassandra and others. A write goes to all N replicas of a key and succeeds when W of them acknowledge it. A read asks R replicas and takes the newest version. If $W + R > N$, every read set overlaps every write set, so at least one replica consulted on a read has the latest acknowledged write. A common setting is $N = 3$, $W = 2$, $R = 2$, which tolerates one replica being down for both reads and writes.

Stale replicas are repaired in the background: on reads, when a coordinator notices an old version, and by **anti-entropy** processes that compare replicas using Merkle trees. When some replicas are unreachable, a **sloppy quorum** may store writes on substitute nodes and hand them back later, which is called hinted handoff; that keeps writes available but

weakens the overlap guarantee. And even strict quorums don't make a system linearizable on their own: concurrent writes, partially failed writes and clock-based conflict resolution all leave gaps. Quorums give you a high probability of fresh reads, not a promise.

COMMON TRAPS

Treating asynchronous replication as durability. An acknowledged write can still vanish in a failover.

Failover without fencing. The old leader returns and keeps accepting writes.

Reading from followers with no session guarantees. Users see their own edits disappear.

Last write wins between leaders. Silent data loss, discovered months later.

Believing $W + R > N$ means strong consistency. It means overlapping quorums, which is less.

EXERCISES

EXERCISE 4.1 Name the anomaly

Name the anomaly and a fix for each report.

- "I changed the invoice amount and the list still shows the old one."
- "A payment appeared on the invoice, then disappeared when I refreshed, then came back."
- "In the support thread, the agent's answer appeared above the customer's question."

SOLUTION

- Read-your-writes.** Read the user's recently changed data from the leader, or from a follower that has reached their last write's log position.
- Monotonic reads.** Pin each user's reads to one replica.
- Consistent prefix.** Keep a thread's messages in one partition, or order them by a sequence number assigned in one place.

EXERCISE 4.2 Configure the quorum

With $N = 5$, which of these settings guarantee that read and write sets overlap, and what does each trade? (W, R): (3, 3), (5, 1), (1, 5), (2, 2), (3, 2).

SOLUTION

- (3, 3): overlaps. Tolerates two replicas down for both reads and writes. The balanced choice.
- (5, 1): overlaps. Very fast reads, but a single unreachable replica blocks every write.
- (1, 5): overlaps. Fast writes, but one unreachable replica blocks every read, and a write acknowledged by one machine is fragile.
- (2, 2): doesn't overlap, since 4 isn't greater than 5. Reads can miss the latest write.
- (3, 2): doesn't overlap, since 5 isn't greater than 5.

EXERCISE 4.3 Resolve the conflict

The invoicing app runs a leader in Europe and one in the US. While the link between them is down, a user in Europe edits an invoice draft's notes and adds a line item, and a colleague in the US changes the due date and also edits the notes. Design the merge.

SOLUTION

- **Due date:** changed on one side only, so take that change.
- **Line items:** treat them as a set with unique ids, so additions from both sides are kept. Removals need care: record them as tombstones, so a removal on one side isn't undone by the other side's older copy.
- **Notes:** changed on both sides. Keep both versions and ask a person, or use a text CRDT if notes are edited collaboratively often enough to justify one.
- **Total:** never merged directly. Recompute it from the merged line items.

And ask whether the draft needs multi-leader writes at all. Giving each team a home region where its writes go would remove the conflict entirely, at the cost of slower writes for travelling users.

- ☐ Do you know which acknowledged writes a failover can lose?
- ☐ Is failover protected against split brain?
- ☐ Which reads go to followers, and which session guarantees do they need?
- ☐ With several leaders, is every conflict resolved deliberately rather than by timestamp?
- ☐ Do quorum settings match the failures you must tolerate?

GO DEEPER

PAPER **Dynamo: Amazon's Highly Available Key-value Store** DeCandia et al., 2007

GUIDE **crdt.tech** papers, talks and implementations of CRDTs

BOOK **Designing Data-Intensive Applications** Kleppmann & Riccomini · the replication chapter

BOOK **Database Internals** Alex Petrov · replication and consistency from the implementer's side

Partitioning and sharding

3 exercises

When data outgrows one machine, split it into partitions and spread them across nodes. The partition key decides almost everything: which queries stay cheap, which become expensive, and where the hot spots will form.

Partitioning and replication together

Partitioning, also called sharding, splits a dataset so that each piece lives on a different node, letting you store more data and handle more writes than one machine can. It's independent of replication: in practice each partition is also replicated, so every node holds several partitions, leading some and following others.

By key range

Assign contiguous ranges of keys to each partition: invoices A to F here, G to M there. Range scans stay efficient, because neighbouring keys live together. Bigtable, HBase, Spanner and CockroachDB work this way, splitting a range automatically when it grows too large.

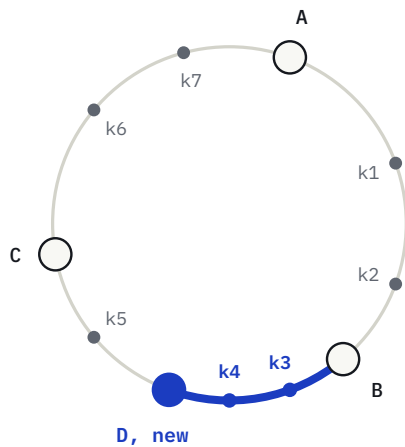
The danger is **hot spots** from keys that arrive in order. If the key starts with a timestamp, every new write lands in the newest range, and one node takes all of today's traffic while the rest sit idle. Put something with spread, such as a tenant id, before the timestamp.

By hash

Hash each key and assign partitions by hash value. Load spreads evenly whatever the keys look like, at the cost of range scans, which now touch every partition. But never place keys with `hash(key) % N`, where N is the number of nodes. Change N from 4 to 5 and about 80% of all keys move to a different node, all at once.

Consistent hashing

Consistent hashing, introduced at MIT in 1997, places both nodes and keys on a circle by hashing them. Each key belongs to the first node clockwise from it. Adding a node moves only the keys it takes over from its neighbour; removing one hands its keys to the next.



KEY	BEFORE	AFTER
k1	B	B
k2	B	B
k3	C	D
k4	C	D
k5	C	C
k6	A	A
k7	A	A

Each key belongs to the first node clockwise from it. Adding D takes over only the keys between B and D, which C used to own; every other key stays where it was.

With only a few points on the circle, the arcs between nodes come out uneven, so one node may own twice as much as another. The fix is **virtual nodes**: each physical node appears at many points. That evens out the load, and when a node leaves, its keys scatter across many survivors rather than landing on one neighbour.

EXAMPLE ring.py

```
import bisect
import hashlib

def _hash(value: str) -> int:
    return int.from_bytes(hashlib.sha256(value.encode()).digest()[:8], "big")

class HashRing:
    def __init__(self, nodes=(), vnodes: int = 100):
        self.vnodes = vnodes
        self._ring: list[tuple[int, str]] = []          # sorted (position, node) pairs
        for node in nodes:
            self.add(node)

    def add(self, node: str) -> None:
        for i in range(self.vnodes):
            bisect.insort(self._ring, (_hash(f"{node}#{i}"), node))

    def remove(self, node: str) -> None:
        self._ring = [(position, n) for position, n in self._ring if n != node]

    def node_for(self, key: str) -> str:
        index = bisect.bisect(self._ring, (_hash(key),)) # first point clockwise
        return self._ring[index % len(self._ring)][1]
```

Run it on 20,000 team ids with four nodes, then add a fifth. About 16% of keys move, all of them to the new node, against about 80% with the modulo approach. Alternatives with similar properties include rendezvous hashing, where each key goes to the node with the highest hash of the key and node combined, and Google's jump consistent hash, which needs no stored ring at all.

Many systems skip rings entirely and use a **fixed number of partitions**, far more than the number of nodes, say 1,024. Keys map to partitions by hash, and partitions map to nodes through a small table. Adding a node moves whole partitions, and the key-to-partition mapping never changes. Kafka topics, Elasticsearch indexes and Redis Cluster's 16,384 hash slots all work this way.

Hot keys

Even perfect hashing can't split a single key. One enormous tenant, one celebrity account, or one viral invoice sends a disproportionate share of traffic to one partition. Remedies, roughly in order of preference: cache the hot key's reads; give the whale its own partition or database, as *The Engineer's Shipyard* suggested for large tenants; and, for write-heavy counters, split the key into several sub-keys such as `views:inv_1#0` to `#9`, writing to a random one and summing all of them on read.

Secondary indexes

Partitioning by team id makes "all invoices of team X" cheap. What about "all overdue invoices across every team", or "invoices for this client email"? There are two designs.

- **Local indexes:** each partition indexes its own data. Writes stay simple, but a query by the secondary attribute must ask every partition and combine the results, called scatter-gather, and it's only as fast as the slowest partition.
- **Global indexes:** the index is partitioned by the indexed value itself. Reads touch one index partition, but a single write may update index partitions on other nodes, usually asynchronously, so the index lags.

Often the better answer is to not ask the operational database at all: stream changes into a search engine or analytics warehouse built for cross-partition questions (chapter 13).

Rebalancing and routing

Moving partitions between nodes is expensive: it copies data over the network while the system carries live traffic. Do it gradually and with throttling, and be wary of fully automatic rebalancing triggered by failure detection. A node that's merely slow can be declared dead, trigger a flood of data movement, and make everything slower, a feedback loop chapter 14 returns to. Clients need to know where each partition lives. They can ask a routing tier, ask any node, which forwards the request, or cache the mapping themselves. The authoritative mapping is usually kept in a consensus-backed store (chapter 8).

COMMON TRAPS

hash(key) % N. Every resize reshuffles almost everything.

Monotonic keys under range partitioning. Timestamps and sequential ids pile onto one node.

Ignoring the whales. One tenant larger than a partition breaks every assumption.

Cross-partition queries on the hot path. Scatter-gather is as slow as the slowest partition.

Automatic rebalancing on failure detection. A slow node triggers a storm of data movement.

EXERCISES

EXERCISE 5.1 Choose the partition key

The invoicing app must shard its invoices table. Evaluate three keys: invoice id, team id, and creation date. Which do you choose, and what problems does it leave?

SOLUTION

- **Invoice id:** perfectly even spread, but listing a team's invoices, the most common query, becomes scatter-gather across every shard.
- **Creation date:** today's shard takes every write. A classic hot spot.
- **Team id:** the right choice. A team's data lives together, so its lists, joins and transactions stay on one shard, and it doubles as tenant isolation.

What it leaves: very large teams can outgrow a shard, so plan to move whales to dedicated shards; and cross-team questions, such as finance's "total overdue across all customers", should run against a warehouse fed by change data capture, not against the shards.

EXERCISE 5.2 Count the moved keys

A cache cluster uses `hash(key) % N` across 10 nodes and grows to 11. Roughly what fraction of keys now map to a different node, and what happens to the database behind the cache? What would consistent hashing change?

SOLUTION

A key stays put only if its hash gives the same remainder modulo 10 and modulo 11, which happens for about 1 key in 11. So about 91% of keys move. Almost every cache lookup misses at once, the full read load lands on the database, and the database probably falls over: the cache was load-bearing (chapter 16).

With consistent hashing, only about 1 in 11 keys moves, the share the new node takes over. Scaling the cache becomes a routine operation instead of a self-inflicted outage.

EXERCISE 5.3 Cool the hot key

Every invoice has a public "viewed" counter, incremented each time a client opens it. One invoice, sent to 40,000 people, receives 3,000 increments a second, all hitting one partition. Design a fix, including how reads work.

SOLUTION

- Split the counter into, say, 16 sub-keys: `views:inv_42#0` to `#15`. Each increment picks one at random, so the writes spread over 16 keys, and potentially 16 partitions.
- A read sums the 16 sub-keys, and caches the total for a few seconds, since nobody needs a view count accurate to the millisecond.
- Better still, batch in the application: count increments in memory for a second and write one increment of n . It's accurate enough and takes thousands of writes down to one per server per second.

Split only keys you've measured as hot. Doing it to every key multiplies read cost for no benefit.

CHECKLIST

- ☐ Does the partition key keep the common queries on one partition?
- ☐ Is placement by consistent hashing or fixed partitions, never modulo the node count?
- ☐ Do you know your largest tenants and hottest keys, and have a plan for each?
- ☐ Are cross-partition questions answered by a system built for them?
- ☐ Is rebalancing gradual, throttled and deliberate?

GO DEEPER

PAPER **Dynamo consistent hashing and virtual nodes in production**

PAPER **A Fast, Minimal Memory, Consistent Hash Algorithm** Lamping & Veach, 2014
· jump consistent hash

DOCS **Redis Cluster specification** hash slots in practice

Consistency models

3 exercises

A consistency model is a contract about what a read may return when data is replicated and accessed concurrently. Stronger models are easier to program against and cost more in latency and availability, so decide which one each part of your system actually needs.

Why we need contracts

With several clients and several replicas, a stream of reads and writes can interleave in many ways. A consistency model says which of those interleavings, or **histories**, are allowed. Engineers talk loosely about "strong" and "eventual" consistency, but the precise models differ in ways that matter, and databases' marketing often claims more than the database delivers. Jepsen, Kyle Kingsbury's testing project, has repeatedly found popular databases violating the models their documentation promised.

Linearizability

The strongest single-object model, and the one people usually mean by "strong consistency". Every operation appears to take effect instantaneously at some moment between its start and its end, and once any read has returned a new value, every later read returns it too. The system behaves as if there were a single copy of the data, even though there are several.

That's what makes certain things possible at all: a lock that only one client can hold, a leader that only one node can be, a username only one person can claim, a balance that can't go below zero. The cost is coordination on every operation, and, during a partition, the side that can't reach a majority must refuse requests (chapter 7).

EXAMPLE a history that is not linearizable

```
client A: write(x = 1)  |-----| (0 to 10 ms)
client B:                read(x) -> 1 |-----| (5 to 15 ms)
client C:                                read(x) -> 0 (20 to 25 ms)
```

C started after A's write had finished and after B had already seen 1, yet C read the old value. A single copy could never behave this way.

Weaker, cheaper models

MODEL	GUARANTEE	FORBIDS, FOR EXAMPLE	WORKS THROUGH A PARTITION?
Linearizable	Single-copy	Reading an old value after	No, on the minority side

	behaviour, in real time	anyone has read the new one	
Sequential	One total order that respects each client's own order	Two clients seeing the same writes in different orders	No
Causal	Causally related operations are seen in order by everyone	Seeing a reply before the message it answers	Yes, if clients stay with one replica
Read-your-writes	A client always sees its own writes	Your edit vanishing when you refresh	Yes, if clients stay with one replica
Eventual	If writes stop, replicas converge	Almost nothing while writes continue	Yes

Causal consistency deserves particular attention. Operations related by cause and effect, in the sense of chapter 3, are seen in the same order everywhere; unrelated operations may be seen in different orders. That rules out the anomalies users find most confusing, and it's the strongest model that can keep working on both sides of a partition. **Session guarantees**, identified by the Bayou project in 1994, are smaller promises made to each client: read-your-writes, monotonic reads, monotonic writes, and writes-follow-reads. They fix most of the anomalies from chapter 4 cheaply.

Eventual consistency guarantees very little on its own, but CRDTs strengthen it to **strong eventual consistency**: replicas that have received the same updates hold the same state, immediately, whatever order the updates arrived in.

Consistency is not isolation

Two vocabularies grew up separately, and they're easy to confuse. **Isolation levels**, such as serializable, snapshot isolation and read committed, describe how *transactions over several objects* interleave. **Consistency models** such as linearizability describe *single objects* in real time. A database can be serializable without being linearizable: its transactions behave as if they ran one at a time, but not necessarily in real-time order, so a read-only transaction might see a slightly old snapshot. The combination of both is called **strict serializability**, which is roughly what Spanner calls external consistency.

You may already have what you need

A single-leader database gives you linearizable operations on the leader for free, including uniqueness. Claiming a team's public URL slug takes one statement, and exactly one caller can ever win.

EXAMPLE claiming a unique slug in PostgreSQL

```
INSERT INTO team_slugs (slug, team_id)
VALUES ('studio-north', 'team_42')
ON CONFLICT (slug) DO NOTHING
RETURNING team_id;      -- a row back means you have it; no row means someone else does
```

The same operation against an eventually consistent store is genuinely hard, because two replicas can each accept the slug before hearing about the other. That's often the deciding question when choosing a datastore: which of your operations need a single authoritative answer?

Choosing per operation

Most systems need several models at once. Money, uniqueness, and permissions changes need linearizable or serializable operations. A user's own profile and settings need read-your-writes. Comments and conversations need causal order. Like counts, view counts and activity feeds are fine with eventual consistency. Paying the cost of the strongest model everywhere is as much a design mistake as using the weakest one for payments.

COMMON TRAPS

"Strongly consistent" in the marketing. Ask which model, under which failures, and look for a Jepsen analysis.

Check-then-act against a replica. Reading a stale copy, then writing based on it.

Confusing serializable with linearizable. One is about transactions, the other about real time.

Strong consistency everywhere. Latency and availability paid for data that didn't need it.

EXERCISES

EXERCISE 6.1 Linearizable or not?

For each history, say whether it could come from a linearizable register that starts at 0, and why.

```
History 1:  A writes x = 1 from 0 to 10 ms.
            B reads x from 5 to 15 ms and gets 1.
            C reads x from 20 to 25 ms and gets 0.
```

```
History 2:  A writes x = 1 from 0 to 10 ms.
            B reads x from 2 to 4 ms and gets 0.
            C reads x from 5 to 8 ms and gets 1.
```

History 3: A writes $x = 1$ from 0 to 10 ms.
B reads x from 2 to 6 ms and gets 1.
B reads x again from 7 to 9 ms and gets 0.

SOLUTION

- **History 1: not linearizable.** C begins after the write has completed, so it must see 1.
- **History 2: linearizable.** Both reads overlap the write. Place the write's instant of effect between 4 and 5 ms: B's read comes before it, C's after.
- **History 3: not linearizable.** Once B has read 1, the write has taken effect, so a later read can't return 0. It also breaks the weaker monotonic-reads guarantee.

EXERCISE 6.2 Pick the model

Choose the weakest model that's safe for each, and justify it.

- Claiming a team's URL slug.
- The number of times an invoice has been viewed.
- A user changing their own display name.
- Comments on an invoice's discussion thread.
- A prepaid credit balance that must never go negative.

SOLUTION

- Linearizable.** Two teams must never both believe they own the slug.
- Eventual**, ideally a CRDT counter. Being briefly behind hurts nobody.
- Read-your-writes.** The user must see their own change; others can see it a moment later.
- Causal.** Replies must never appear before what they reply to.
- Serializable transactions, or a linearizable compare-and-set.** Two concurrent spends must not both pass the balance check.

EXERCISE 6.3 Fix the check

Occasionally two teams end up with the same slug. Explain the two bugs in this code, and fix them.

```
def claim_slug(slug, team_id):
    if not replica_db.slug_exists(slug):           # read from a follower
        leader_db.insert_slug(slug, team_id)
        return True
    return False
```

SOLUTION

A stale read: the follower may not have seen a slug claimed a moment ago. A race: even on the leader, two requests can both check, both see nothing, and both insert, because the check and the insert are separate steps.

Delete the check. Rely on a unique constraint on the leader and a single atomic statement: `INSERT ... ON CONFLICT (slug) DO NOTHING RETURNING team_id`. If a row comes back, this team won; if not, someone else did. The database performs the check and the write as one linearizable operation.

CHECKLIST

- ☐ For each important operation, can you name the consistency model it needs?
- ☐ Do uniqueness and money rely on a single authoritative, atomic operation?
- ☐ Do reads that follow a user's own writes have read-your-writes?
- ☐ Have you checked what your datastore really guarantees, not what it advertises?

GO DEEPER

GUIDE [Consistency Models Jepsen](#) · a clickable map of how the models relate

ESSAY [Strong consistency models](#) Kyle Kingsbury

REPORTS [Jepsen analyses what real databases actually do under failure](#)

CAP and PACELC, stated correctly

3 exercises

CAP says only this: when the network partitions, a replicated system must choose between answering every request and keeping every answer consistent. PACELC adds the trade-off you face far more often: even without a partition, every request trades latency against consistency.

What CAP actually says

Eric Brewer proposed the idea in 2000, and in 2002 Seth Gilbert and Nancy Lynch proved it with precise definitions. **C** means linearizability, chapter 6's strongest model. **A** means every request received by a node that hasn't failed gets a non-error response. **P** means the network may lose any number of messages between nodes. The theorem: during a partition, a system can't have both C and A. A node that can't reach the others must either answer, possibly with stale data, or refuse, and lose availability.

What it doesn't say

- **"Pick two out of three" is misleading.** You don't get to choose whether the network partitions. The only real choice is what to do when it does.
- **"CA" isn't an option for a distributed system.** A system can be consistent and available only while it runs on one node or ignores partitions, and then the first partition makes the choice for you, usually badly.
- **CAP's definitions are narrow.** Its availability means every non-failed node answers, which few real systems attempt. Its consistency is only linearizability. Many useful systems are neither CAP-consistent nor CAP-available: a single-leader database with asynchronous failover can lose acknowledged writes and is unavailable on the old leader's side. Martin Kleppmann's 2015 essay makes this case under the blunt title *Please stop calling databases CP or AP*.
- **The choice isn't global.** Brewer's own 2012 retrospective stressed that partitions are rare, that systems can choose differently for different operations and data, and that the interesting work is in how you detect a partition, operate during it, and recover afterwards.

PACELC: the trade-off you make every day

Daniel Abadi pointed out in 2010 that CAP describes a rare situation and ignores a constant one. His formulation: if there's a **P**artition, choose between **A**vailability and **C**onsistency; Else, even when everything is healthy, choose between **L**atency and **C**onsistency. Waiting for replicas to agree takes round trips, and round trips take time, on every single request.

SYSTEM, AS USUALLY CONFIGURED	DURING A PARTITION	NORMALLY	CLASS
Dynamo-style stores such as Cassandra, with low consistency levels	Keep answering	Answer fast, possibly stale	PA/EL
DynamoDB with default, eventually consistent reads	Keep answering	Fast reads, possibly stale	PA/EL
Spanner, CockroachDB	Refuse on the minority side	Pay coordination latency	PC/EC
A single-leader database reading only from the leader	Unavailable if the leader is cut off	Consistent reads	PC/EC

The classifications are about defaults. Many databases let you choose per request, which is the real lesson: consistency is a property of an operation, not of a product.

EXAMPLE choosing per query, with the Cassandra and DynamoDB Python clients

```
from cassandra import ConsistencyLevel
from cassandra.query import SimpleStatement

balance = SimpleStatement(
    "SELECT balance_minor FROM credits WHERE team_id = %s",
    consistency_level=ConsistencyLevel.QUORUM)    # a majority must agree: slower, fresher

activity = SimpleStatement(
    "SELECT * FROM activity WHERE team_id = %s LIMIT 20",
    consistency_level=ConsistencyLevel.ONE)       # any replica: fast, maybe stale

# DynamoDB: eventually consistent by default; opt in per read.
table.get_item(Key={"team_id": team_id}, ConsistentRead=True)
```

A partition, from the inside

Suppose the invoicing app runs a five-node consensus-backed database across three regions, and the link to one region fails. The two nodes in that region can't reach a majority. They refuse writes, and consistent reads, until the link returns: consistency is kept, and availability is lost for users routed there. The other three nodes form a majority and carry on normally. A leaderless store in the same situation would let both sides keep accepting writes, and reconcile the conflicting versions afterwards: availability is kept, and consistency must be repaired later.

Neither behaviour is right in general. The useful question is, for each operation, which would hurt the business more during a partition: an error, or a wrong answer?

COMMON TRAPS

"Our system is CA." Only until the first partition.

Labelling a whole database CP or AP. Behaviour depends on configuration and on the operation.

Treating eventual consistency as "wrong data". With session guarantees and good design, users rarely notice it.

Designing only for partitions. The latency cost of consistency is paid every day; the partition cost, rarely.

EXERCISES

EXERCISE 7.1 True or false?

Say whether each statement is true, and correct the false ones.

- a. "Our database is CA."
 - b. "A CP system is completely unavailable during a partition."
 - c. "Eventual consistency means users see wrong data."
 - d. "A single-leader database with asynchronous replication and automatic failover is CP."
-

SOLUTION

- a. **False** for any distributed deployment. Partitions aren't optional, so the only question is how it behaves during one.
- b. **False.** Only nodes that can't reach a majority refuse requests. The majority side carries on.
- c. **False.** Replicas may briefly disagree. With session guarantees and good design, most users never notice.
- d. **False.** Failover can lose acknowledged writes, so it isn't linearizable, and the old leader's side is unavailable. It's neither CP nor AP in CAP's terms, which is exactly Kleppmann's point.

EXERCISE 7.2 Choose during the partition

The link between the invoicing app's European and US regions fails for twenty minutes. For each operation, choose to keep answering or to refuse, and say why.

- a. Viewing the invoice list.
- b. Creating a new invoice draft.
- c. Recording a payment against an invoice.
- d. Transferring ownership of a team to another person.
- e. Sending scheduled reminder emails.

SOLUTION

- a. **Answer**, from the local copy, perhaps with a "may be out of date" notice. Stale is far better than nothing.
- b. **Answer**. A new draft conflicts with nothing, and a unique id generated locally avoids collisions.
- c. **Accept it and reconcile**, as long as payments are recorded as append-only events rather than balance updates. Money really arrived, and refusing to record it helps nobody. Duplicates are removed later by payment id.
- d. **Refuse**, or queue it for after the partition. Ownership decides who can do everything else, and two conflicting transfers would be very hard to untangle.
- e. **Carry on per region**, using idempotency keys, so that if both sides decide to send the same reminder, the email provider or a later check suppresses the duplicate.

EXERCISE 7.3 Classify the configuration

Classify each configuration using PACELC, per operation where it matters.

- a. DynamoDB reads with the default settings.
- b. DynamoDB reads with `ConsistentRead=True`.
- c. A PostgreSQL primary with asynchronous replicas, where the app reads from the replicas.
- d. Cassandra with QUORUM for both reads and writes.

SOLUTION

- a. **PA/EL**: fast and possibly stale, and it keeps answering.

- b. **PC/EC for these reads:** they cost more and take longer, and can fail when consistency can't be guaranteed.
- c. **EL for reads:** replicas answer fast and may be stale. During a partition, replicas keep serving stale reads while writes fail if the primary is unreachable, so the classification differs for reads and writes.
- d. **Close to PC/EC for those operations:** a majority must respond, so the minority side fails, and every request waits for several replicas. It still isn't strictly linearizable, for the reasons chapter 4 gave.

The answers that needed "for these operations" are the point of the exercise.

CHECKLIST

- ☐ For each operation, have you decided: error or stale answer during a partition?
- ☐ Have you counted the latency cost of consistency on the normal path?
- ☐ Do you choose consistency per query where the datastore allows it?
- ☐ Is there a plan for recovering and reconciling after a partition heals?

GO DEEPER

ESSAY CAP Twelve Years Later: How the "Rules" Have Changed Eric Brewer, 2012

ESSAY Please stop calling databases CP or AP Martin Kleppmann, 2015

REFERENCE PACELC Daniel Abadi's formulation, with examples

PART III

Agreement

How do machines that can fail, over a network that loses messages, agree on anything? Three chapters: consensus and the Raft algorithm, the leases and locks built on it, and what to do when one business operation spans several services.

Consensus and Raft

3 exercises

Consensus lets a group of machines agree on a sequence of decisions even when some of them fail. You'll rarely implement it, but you'll depend on it constantly, and understanding Raft tells you exactly what the systems built on it can and can't promise.

The problem

Several nodes must agree on a value, or on an ordered log of values, despite crashes and lost messages. A solution needs two properties. **Safety**: nodes never decide differently. **Liveness**: they eventually decide. In 1985, Fischer, Lynch and Paterson proved that no deterministic algorithm can guarantee both in a fully asynchronous system if even one node may crash. This FLP result doesn't make consensus impossible in practice. It means real algorithms keep safety always, and use timeouts to make progress during the periods when the network behaves, which in the partially synchronous world of chapter 2 is nearly always.

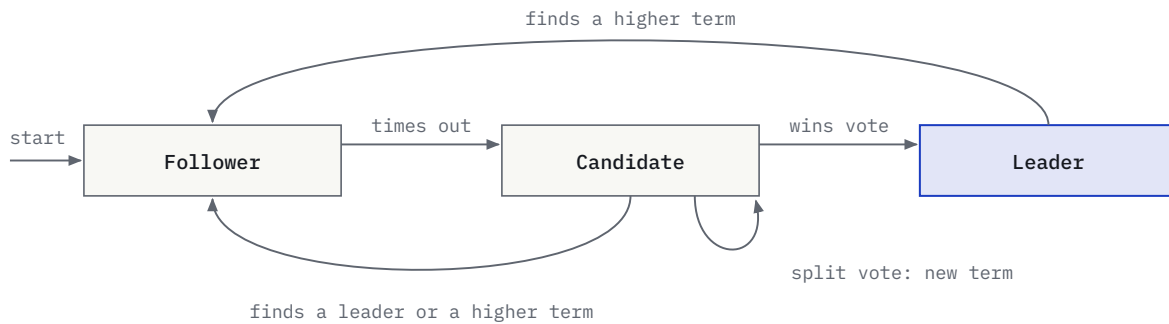
You depend on consensus whenever you use etcd, which stores Kubernetes' state, ZooKeeper, Consul, the metadata layer of Kafka, or distributed databases such as CockroachDB and TiDB, which run a consensus group for every range of data. Since Kafka 4.0 in March 2025, Kafka no longer needs ZooKeeper at all; it runs its own Raft-based metadata quorum, called KRaft.

Majorities

Consensus algorithms decide by **majority**. With $2f + 1$ nodes, any f can fail and the rest still form a majority. And any two majorities of the same group always share at least one node, so two conflicting decisions can never both be approved: some node would have had to vote for both. Three nodes tolerate one failure; five tolerate two. Adding a fourth node to three doesn't help, because a majority of four is three, so it still tolerates only one failure, with more nodes that can fail.

Raft

Leslie Lamport's Paxos, from 1989, was the first practical solution, and is famously hard to understand. Diego Ongaro and John Ousterhout designed Raft in 2014 with understandability as an explicit goal, and it has become the default in new systems. It has three parts.



Every node starts as a follower. A follower that hears nothing from a leader before its randomised timeout becomes a candidate and asks for votes; a candidate that wins a majority becomes leader. Any node that learns of a higher term steps back down.

Leader election. Time is divided into numbered **terms**, each with at most one leader. A leader sends regular heartbeats. A follower that hears nothing before its election timeout, randomised, for example between 150 and 300 milliseconds in the original paper, so that nodes rarely time out together, starts a new term, votes for itself, and asks the others for votes. Each node grants at most one vote per term, and only to a candidate whose log is at least as up to date as its own. A candidate with votes from a majority becomes leader. If the vote splits, the candidates time out and try again in a new term.

Log replication. Clients send commands to the leader, which appends them to its log and sends them to followers. Each message includes the index and term of the entry just before the new ones, and a follower rejects entries that don't follow on from what it has, so the leader backs up until the logs match and then overwrites any divergent tail. An entry is **committed** once the leader has stored it on a majority, and only then is it applied to the state machine and acknowledged to the client. Every node applies the same commands in the same order, so every node reaches the same state. That's a **replicated state machine**.

Safety. The voting rule is what ties it together. A committed entry is on a majority; a new leader needs votes from a majority; the two majorities overlap; and no node votes for a candidate whose log is behind its own. So every new leader already holds every committed entry.

EXAMPLE raft_votes.py · the heart of Raft's safety

```

def log_is_up_to_date(cand_last_term, cand_last_index, my_last_term, my_last_index):
    if cand_last_term != my_last_term:
        return cand_last_term > my_last_term      # a later term wins, whatever the length
    return cand_last_index >= my_last_index        # same term: the longer log wins

def handle_request_vote(state, req):
    if req.term < state.current_term:
        return VoteReply(term=state.current_term, granted=False)  # a stale candidate
    if req.term > state.current_term:
        state.current_term, state.voted_for = req.term, None      # a new term: step down
        state.role = "follower"
  
```

```

already_voted = state.voted_for not in (None, req.candidate_id)
if not already_voted and log_is_up_to_date(
    req.last_log_term, req.last_log_index,
    state.last_log_term(), state.last_log_index()):
    state.voted_for = req.candidate_id
    state.persist() # the vote must survive a crash before we reply
    state.reset_election_timer()
    return VoteReply(term=state.current_term, granted=True)
return VoteReply(term=state.current_term, granted=False)

```

Note the `persist()` before replying. If a node could forget its vote in a crash, it might vote twice in the same term, and two leaders could be elected. Details like this are why the Raft paper's one-page summary of the rules, its Figure 2, is worth reading line by line.

What consensus is for, and what it costs

Use a consensus-backed store for the small amount of data that must be agreed exactly: who the leader is, which node owns which partition, configuration, locks and leases (chapter 9). Every committed write costs at least one round trip to a majority, plus a disk write on each, so consensus groups are small, usually three or five nodes, and are kept out of the high-volume data path unless, as in distributed databases, the design shards data across many independent groups.

Across regions, that round trip is tens or hundreds of milliseconds, which is why globally consistent databases feel slower than local ones: that's PACELC's latency cost, made concrete.

COMMON TRAPS

Even-sized clusters. Four nodes tolerate no more failures than three.

Two availability zones. Losing the zone holding the majority loses the cluster. Use three.

Putting bulk data through a consensus store. etcd and ZooKeeper are for coordination, not for your invoices.

Writing your own consensus for production. Use a proven implementation; write your own to learn.

Forgetting that clients must retry. A write lost with a deposed leader was never committed, and the client must send it again, idempotently.

EXERCISES

EXERCISE 8.1 Size the cluster

How many nodes tolerate two simultaneous failures? How many failures does a four-node cluster tolerate? And why are consensus clusters usually spread over three availability zones rather than two?

SOLUTION

Five nodes, since $2 \times 2 + 1 = 5$. A four-node cluster needs three for a majority, so it tolerates only one failure, the same as three nodes.

With two zones, one zone must hold a majority, and losing that zone loses the cluster. With five nodes placed two, two and one across three zones, losing any whole zone leaves at least three nodes, which is still a majority.

EXERCISE 8.2 Grant the vote?

A node is in term 7, hasn't voted in it, and its last log entry has term 6 and index 42. It receives these vote requests in order. Which does it grant?

- a. Term 7, from a candidate whose last entry is term 6, index 40.
- b. Term 8, last entry term 6, index 42.
- c. Term 6, last entry term 9, index 99.
- d. Term 8, from a different candidate, last entry term 6, index 50.

And separately: a fresh node in the same starting state receives term 8, last entry term 7, index 10. Grant or not?

SOLUTION

- a. **Deny.** Same last term, but the candidate's log is shorter.
- b. **Grant.** A newer term, so the node moves to term 8, and the candidate's log is as up to date as its own.
- c. **Deny.** Term 6 is older than the node's current term, whatever the log says.
- d. **Deny.** The node has already voted in term 8, for the candidate in (b).

The fresh node **grants** the last request. The candidate's last entry has a later term, 7 against 6, and that decides it even though its log is much shorter: later terms beat longer logs.

EXERCISE 8.3 Partition the cluster

A five-node Raft cluster is split by a partition into a group of two, which contains the current leader, and a group of three. Describe what happens to client writes on each side, and what happens when the partition heals.

SOLUTION

- **The side with the old leader:** it still believes it's leader and accepts writes, but can't replicate them to a majority, so nothing commits. Clients' requests hang and time out.
- **The side with three nodes:** followers stop hearing heartbeats, one times out, wins three votes in a new, higher term, and becomes leader. Writes sent there commit normally.
- **When it heals:** the old leader sees the higher term and steps down. Its uncommitted entries are overwritten by the new leader's log.

No committed write is ever lost, but clients that sent writes to the old leader got no answer and must retry them. Those retries must be idempotent, because from the client's side the outcome was unknown (chapter 2).

CHECKLIST

- ☐ Are consensus clusters an odd size, spread across three failure domains?
- ☐ Is only coordination data in the consensus store?
- ☐ Do clients retry, idempotently, writes whose outcome is unknown?
- ☐ Do you know which of your dependencies rely on consensus, and what happens when they lose quorum?

GO DEEPER

- PAPER** In Search of an Understandable Consensus Algorithm Ongaro & Ousterhout, 2014 · with visualisations
- VISUAL** The Secret Lives of Data: Raft an animated walkthrough
- COURSE** MIT 6.5840 Distributed Systems the Raft labs are this stage's deliverable
- PAPER** Paxos Made Simple Leslie Lamport, 2001

Leases, locks and fencing

3 exercises

A lock that expires is safe only if the resource it protects can reject a holder whose time has run out. Pair every lease with a fencing token, and check the token at the place where the damage would be done.

Why distributed locks exist

You often need exactly one of something: one worker processing a particular job, one instance running the nightly scheduler, one node acting as primary for a shard. On a single machine, a mutex does this. Across machines, you need a lock that survives the holder crashing, so it must expire. A lock with an expiry is a **lease**: granted for, say, ten seconds, kept alive by renewing it, and automatically released if the holder stops renewing, so a dead holder can't block everyone forever.

The problem with expiry

A lease protects you only if the holder notices when it has expired. It often can't. A process can be frozen by a long garbage-collection pause, by its virtual machine being suspended and migrated, by heavy swapping, or simply by being descheduled, and then resume with no idea that time has passed. Network packets can be delayed. Clocks can jump. From the frozen process's point of view, it checked the lease a moment ago and it was valid.

EXAMPLE how two holders happen

```
t = 0 s    Client 1 acquires the lease, valid until t = 10 s
t = 1 s    Client 1 is frozen by a long garbage-collection pause
t = 10 s   The lease expires
t = 11 s   Client 2 acquires the lease and writes to the file
t = 15 s   Client 1 resumes, still believes it holds the lease, and writes too
           Two writers, one file: corrupted
```

No amount of care by Client 1 fixes this, because the pause can happen between any check and the write that follows it. The protection has to live in the resource.

Fencing tokens

Have the lock service hand out a number that increases every time the lease is granted: 33 to Client 1, 34 to Client 2. Every write to the protected resource carries the holder's token, and the resource remembers the highest token it has seen and rejects anything lower. When Client 1 wakes and writes with token 33, the storage has already seen 34, and refuses.

EXAMPLE `fencing.py` · the check lives in the resource

```
class StaleLeaseholder(Exception):
    pass

class FencedStore:
    def __init__(self):
        self.highest_token = 0
        self.data = {}

    def write(self, key: str, value, token: int) -> None:
        if token < self.highest_token:
            raise StaleLeaseholder(f"token {token} is older than {self.highest_token}")
        self.highest_token = token
        self.data[key] = value
```

In a database, the same check is a condition on the write itself, such as `UPDATE shard_owner SET ... WHERE shard = %s AND epoch <= %s`. The tokens must come from something linearizable: a consensus-backed store such as etcd, whose revision numbers only ever increase, ZooKeeper's versions, or a database sequence. Google's Chubby lock service, described in 2006, called them **sequencers**; primary-election schemes call them **epochs** or generation numbers. The idea is the same.

Efficiency or correctness?

Martin Kleppmann's 2016 essay *How to do distributed locking* made a distinction worth remembering. Ask what happens if the lock fails and two holders run at once.

- If it merely wastes effort, such as two workers generating the same report, a lock is an *efficiency* optimisation. A simple lease in Redis, with occasional duplicates, is fine.
- If it corrupts data or moves money twice, the lock is protecting *correctness*, and it needs a consensus-backed service plus fencing tokens checked by the resource.

The same essay criticised Redlock, an algorithm for locking across several Redis nodes, for depending on bounded pauses and clock drift and for producing no fencing token; Redis's creator published a response, and the debate is worth reading for how carefully each side states its assumptions. The practical conclusion hasn't changed: timing assumptions are exactly what real systems violate.

The same bug, at planet scale

On 19 and 20 October 2025, a race of exactly this shape took down a large part of Amazon Web Services' US-East-1 region for around fifteen hours. Two independent automation workers updated the DNS records for DynamoDB's regional endpoint. One checked that its plan was newer than the one already applied, then ran into unusual delays. By the time it applied the plan, its check was stale, and it overwrote a newer plan with an older one; a cleanup process then deleted that old plan, leaving the endpoint with no DNS records at all.

Services that depended on DynamoDB, including parts of EC2, failed in cascade. The general lesson is the one in this chapter: a check made at one moment protects nothing at a later moment. The version comparison must happen atomically, at the point of the write.

Better still: don't need a lock

- **Conditional writes.** Compare-and-set with a version number (*The Engineer's Shipyard*, chapter 2) makes the database the arbiter, with no lock at all.
- **Idempotent operations.** If running twice is harmless, two runners are harmless.
- **Single-writer ownership.** Assign each partition to exactly one consumer, as Kafka's consumer groups do (chapter 11), so concurrent writers can't arise.
- **A platform primitive.** Kubernetes controllers elect leaders with Lease objects stored in etcd; use what your platform already trusts rather than inventing a scheme.

COMMON TRAPS

Checking the lease, then writing. A pause between the two makes the check worthless.

Locks without fencing protecting anything that matters.

Timing assumptions. "Pauses never exceed a second" holds until the day it doesn't.

Using a lock where a conditional write would do. More moving parts, weaker guarantee.

EXERCISES

EXERCISE 9.1 Trace the fence

Replay the timeline above with fencing: Client 1's lease comes with token 33 and Client 2's with token 34. What does the storage do at $t = 11$ and at $t = 15$? What would happen if the storage checked tokens but the lock service reused numbers after a restart?

SOLUTION

At $t = 11$, Client 2 writes with 34, higher than anything seen, so it's accepted and the highest token becomes 34. At $t = 15$, Client 1 writes with 33, lower than 34, so it's rejected, and Client 1 learns it lost the lease.

If the lock service forgot its counter and handed out 1, 2, 3 again, a new holder's valid writes would be rejected as stale, or worse, an old holder with a high number would beat a new holder with a low one. Tokens must increase forever, which is why they come from a durable, consensus-backed counter.

EXERCISE 9.2 Efficiency or correctness?

For each use, is a simple Redis lease with an expiry good enough?

- Stop two workers rendering the same monthly PDF, which is idempotent anyway.
- Make sure only one scheduler instance enqueues the daily reminders, which already carry deduplication keys.
- Make sure only one process deducts from a team's prepaid credit.
- Make sure exactly one node is primary for a shard of invoices.

SOLUTION

- Yes.** A rare duplicate costs a few seconds of CPU.
- Yes.** The deduplication keys already make a double run harmless; the lock just saves work.
- No, and it doesn't need a lock at all.** Use a transaction with a conditional update, `UPDATE credits SET balance = balance - %s WHERE team_id = %s AND balance >= %s`, so the database enforces the rule.
- No.** Use consensus-based leadership, and fence every write with the primary's epoch, so a deposed primary that wakes up can't write.

EXERCISE 9.3 Replace the lock

Rewrite this without a lock, so that the payment can be recorded at most once and never on an invoice that isn't in the "sent" state.

```
with distributed_lock(f"invoice:{invoice_id}", ttl=10):
    invoice = db.get_invoice(invoice_id)
    if invoice.status == "sent":
        db.execute("UPDATE invoices SET status = 'paid' WHERE id = %s", (invoice_id,))
        db.insert_payment(invoice_id, amount_minor)
```

SOLUTION

```

with db.transaction():
    updated = db.execute(
        "UPDATE invoices SET status = 'paid', version = version + 1 "
        "WHERE id = %s AND status = 'sent'",
        (invoice_id,)).rowcount
    if updated:
        db.insert_payment(invoice_id, amount_minor, payment_id=payment_id)

```

The condition and the change happen in one statement, so no pause, crash or second worker can slip between them. Only one transaction can move the invoice from "sent" to "paid". A unique constraint on `payment_id` adds a second guarantee against recording the same payment twice.

CHECKLIST

- ☐ For each lock: does failure waste effort, or corrupt data?
- ☐ Does every correctness-critical lease come with a fencing token?
- ☐ Does the protected resource itself reject stale tokens?
- ☐ Could a conditional write, idempotency or single ownership replace the lock?

GO DEEPER

ESSAY How to do distributed locking Martin Kleppmann, 2016

PAPER The Chubby lock service for loosely-coupled distributed systems Mike Burrows, 2006

REPORT A single DNS race condition brought AWS to its knees The Register, on the October 2025 outage

DOCS Leases Kubernetes

Transactions across services

3 exercises

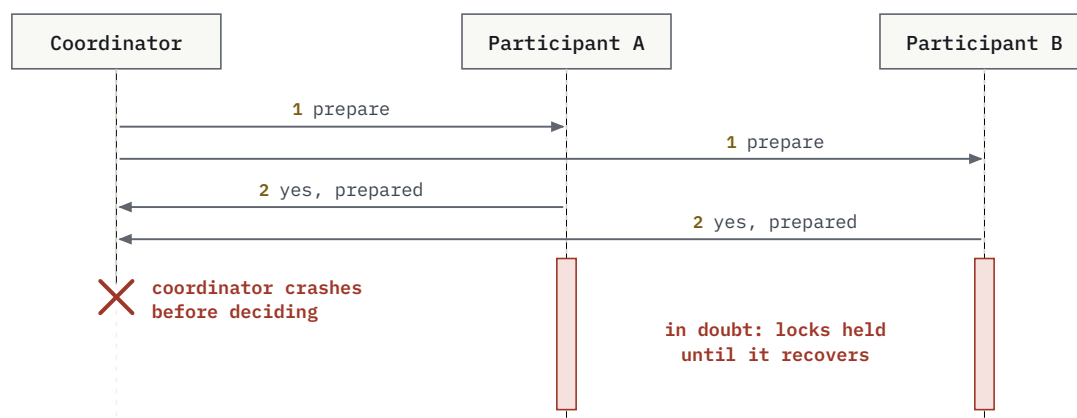
Atomic transactions stop at the edge of a database. Across services, use two-phase commit only where you control every participant and can live with it blocking. Everywhere else, break the work into local transactions joined by messages, and use sagas to undo what can't be committed together.

The problem

Upgrading a team's subscription in the grown-up invoicing app touches several services, each with its own database: billing charges the card, entitlements unlocks the features, notifications sends the receipt, and the CRM records the upgrade. No single database transaction covers all four. If the charge succeeds and the entitlement update fails, the customer has paid for nothing. Something has to make the whole operation all-or-nothing, or close enough.

Two-phase commit

The classic protocol has a **coordinator** and several **participants**. In the first phase, the coordinator asks each participant to *prepare*: do all the work, write it durably, take the necessary locks, and vote. A yes vote is a promise: "I can commit, and I will if you tell me to, even if I crash and restart first." In the second phase, if every vote was yes, the coordinator records its decision and tells everyone to commit; if any vote was no, everyone aborts.



Once a participant votes yes, it has promised to commit if told to, so it can neither commit nor abort on its own. If the coordinator dies at that moment, the participants wait, holding their locks, until it comes back.

That window is the protocol's central weakness: two-phase commit is a **blocking** protocol. Rows stay locked while participants are in doubt, so other transactions queue behind them, and a coordinator that never recovers leaves them stuck until someone intervenes by hand. It also costs at least two round trips and several durable writes per transaction, and every participant must support it. Your payment provider won't join your transaction, and nor will an email service.

Two-phase commit is still used successfully where those problems are solved. Distributed databases such as Spanner run it between their own shards, with each participant and the coordinator replicated by consensus, so a crashed machine doesn't stop the protocol. Inside one such database, let it do this for you. What you shouldn't do is build it yourself across independent services.

Sagas

A saga, a term from a 1987 paper by Hector Garcia-Molina and Kenneth Salem, replaces one big transaction with a sequence of local transactions, each committed on its own. Every step has a **compensating action** that semantically undoes it: a refund for a charge, a release for a reservation, a cancellation for a booking. If a step fails permanently, the saga runs the compensations for the steps already completed, in reverse order.

EXAMPLE `saga.py` · an orchestrated saga, simplified

```
from dataclasses import dataclass
from typing import Callable

@dataclass
class Step:
    name: str
    action: Callable[[dict], None]
    compensate: Callable[[dict], None]

def run_saga(steps: list[Step], context: dict) -> bool:
    completed: list[Step] = []
    for step in steps:
        try:
            step.action(context)
            completed.append(step)
        except PermanentFailure:
            for done in reversed(completed):
                done.compensate(context)  # must be idempotent, and retried until it works
            return False
    return True

upgrade_subscription = [
    Step("record pending upgrade", record_pending_upgrade, cancel_pending_upgrade),
    Step("charge prorated amount", charge_card, refund_charge),
    Step("unlock plan features", apply_entitlements, revert_entitlements),
]
```

A production orchestrator persists the saga's progress after every step, so a crash mid-saga resumes rather than starting over, and it retries transient failures before declaring a step failed. Workflow engines such as Temporal exist to provide exactly this durability.

Two design rules make sagas manageable.

- **Put irreversible steps last.** Some actions can't be compensated: an email can't be unsent, a parcel can't be unshipped. Order the saga so that everything before a certain step can be undone, and everything after it is guaranteed to succeed eventually with retries. That step is called the **pivot**. The confirmation email belongs after it.
- **Design for visible intermediate states.** A saga isn't isolated: other requests can see the half-finished state, a charge without features, for example. Use explicit pending states that the rest of the system understands, rather than pretending the operation is atomic.

Orchestration or choreography?

In an **orchestrated** saga, a central coordinator tells each service what to do, as in the code above. The flow is explicit and easy to follow, test and monitor. In a **choreographed** saga, each service reacts to events from the others: billing emits `PaymentCaptured`, entitlements reacts and emits `PlanUpgraded`, and so on. Coupling is lower, but the flow exists only as the sum of many subscriptions, and understanding it means reading all of them. Start with orchestration for anything with more than two or three steps.

The plumbing: outbox and idempotent consumers

Every step of a saga changes a local database and sends a message, which is the dual-write problem from *The Engineer's Shipyard*: use the transactional outbox. Messages are delivered at least once, so every consumer must be idempotent. The simplest way is a table of processed message ids, written in the same transaction as the effect.

EXAMPLE `consumer.py`

```
def handle_payment_captured(db, message):
    with db.transaction():
        first_time = db.execute(
            "INSERT INTO processed_messages (message_id) VALUES (%s) ON CONFLICT DO NOTHING",
            (message.id,)).rowcount
        if not first_time:
            return # a redelivery: already handled
        db.execute("UPDATE upgrades SET status = 'paid' WHERE id = %s", (message.upgrade_id,))
```

Pat Helland's essay *Life beyond Distributed Transactions* argues this is how large systems end up working anyway: entities that are each consistent on their own, communicating by messages, with idempotence and reconciliation instead of global atomicity.

COMMON TRAPS

Home-made two-phase commit across services. All of the blocking, none of the tooling.

Compensations that can fail for good. A refund that errors out leaves the saga stuck; compensations must retry until they succeed.

Irreversible steps early. The email went out; then the charge failed.

Pretending sagas are isolated. Other requests see every intermediate state.

Distributed transactions where one database would do. If the data can live together, keep it together.

EXERCISES

EXERCISE 10.1 Design the saga

Design the saga for upgrading a subscription: charge the prorated amount, unlock the new plan's features, send a confirmation email, and record the upgrade in the CRM. Give the order, each compensation, and the pivot.

SOLUTION

1. **Record a pending upgrade** locally. Compensation: mark it cancelled.
2. **Charge the card**, with an idempotency key. Compensation: refund. This is the **pivot**: once the charge succeeds, the saga must complete.
3. **Unlock the features**. No compensation needed after the pivot; retry until it succeeds, and alert if it keeps failing.
4. **Send the confirmation email**. Irreversible, so it comes after the pivot, and retries are safe with an idempotency key.
5. **Update the CRM**. Retry until it succeeds. Nothing depends on it, so it's last.

If the charge fails, only step 1 needs undoing, and the customer is never emailed about an upgrade that didn't happen.

EXERCISE 10.2 The report that didn't add up

A saga moves prepaid credit from one team to another, which live on different database shards: debit team A, then credit team B. A finance report that sums every balance occasionally comes out short. Explain why, and propose two fixes.

SOLUTION

Between the debit and the credit, the money is in neither balance. A report running at that moment sees it as missing. Sagas don't provide isolation.

- **Make the in-flight state visible:** record each transfer in a transfers table with a status, and have the report add pending transfers to the total.
- **Use a ledger:** write each transfer as one entry with two legs, debit and credit, in the same transaction, which is possible if ledger entries are stored together rather than beside each team's balance. Balances then become sums over the ledger, and they always add up.

EXERCISE 10.3 Two-phase commit or saga?

Choose the approach for each case.

- Moving credit between two teams whose rows are in the same PostgreSQL database.
- Booking a flight, a hotel and a car with three different external companies.
- Updating an order and publishing an "order updated" event.
- A distributed SQL database updating rows that live on two different nodes.

SOLUTION

- An ordinary local transaction.** Nothing distributed is needed.
- A saga,** with cancellations as compensations. External companies won't join your two-phase commit.
- The transactional outbox.** Write the event to an outbox table in the same transaction as the order.
- Let the database handle it,** with its own two-phase commit over consensus-replicated participants.

CHECKLIST

- ☐ Could this data live in one database, and avoid the problem entirely?
- ☐ Does every saga step have an idempotent compensation, or come after the pivot?
- ☐ Are irreversible steps placed after the pivot?
- ☐ Is saga progress persisted, so a crash resumes rather than restarts?
- ☐ Are intermediate states explicit and understood by the rest of the system?

GO DEEPER

PAPER Sagas Garcia-Molina & Salem, 1987

PATTERN Saga Chris Richardson, microservices.io

ESSAY Life beyond Distributed Transactions Pat Helland, ACM Queue

DOCS Temporal a durable workflow engine for orchestrated sagas

PART IV

Messages and streams

Most distributed systems are held together by messages. Three chapters: the difference between queues and logs and what "exactly once" really means, events as an architecture, and processing data in batches and in streams.

Queues, logs and delivery guarantees

3 exercises

A queue hands each message to one consumer and forgets it. A log keeps every message in order and lets any number of readers replay it. Both deliver at least once in practice, so "exactly once" is something you build out of idempotence, not something you buy.

Queues

In a message queue such as RabbitMQ or Amazon SQS, producers add messages and competing consumers take them. Each message goes to one consumer, which acknowledges it when done, and then it's deleted. A consumer that dies without acknowledging has its message redelivered to someone else. Queues are ideal for spreading independent tasks across workers: resize this image, send this email. They make weak promises about order, because several consumers work in parallel and redeliveries jump the queue, though some offer ordered variants, such as SQS FIFO queues with message groups.

Logs

A log, such as Apache Kafka, Redpanda or Amazon Kinesis, is an append-only sequence of messages that isn't deleted when read. Each topic is split into **partitions**. Within a partition, messages keep their order and are numbered by **offset**. Consumers don't remove messages; each consumer group records how far it has read, and can rewind to reprocess history. Messages are kept for a configured period, or forever with **log compaction**, which keeps only the latest message for each key, turning the log into a durable, replayable table.

Within a **consumer group**, each partition is read by exactly one member, which gives two properties at once: order within a partition, and parallelism across partitions. The number of partitions therefore caps the useful number of consumers. A producer chooses the partition by hashing the message key, so everything about one invoice, keyed by its id, stays in order.

Jay Kreps' 2013 essay *The Log* argued that this one abstraction, an ordered, replayable record of what happened, unifies databases, replication and data integration. That's why logs sit at the centre of so many architectures, and why the line between queues and logs keeps blurring: Kafka 4.0 added early support for queue-style consumption.

ASPECT	QUEUE	LOG
After reading	Message deleted	Message kept; reader's position recorded
Multiple	Compete for	Each group reads everything, independently

readers	messages	
Order	Weak	Strict within a partition
Replay	No	Yes, from any retained offset
Best for	Independent tasks	Event streams many systems consume; rebuilding state

Delivery guarantees

Where a consumer records its progress, relative to doing the work, decides what happens when it crashes.

- **At most once:** record progress, then process. A crash in between loses the message.
- **At least once:** process, then record progress. A crash in between processes the message again. This is the normal, sensible default.
- **Exactly once:** possible only inside a closed system. Kafka's transactions, introduced in 2017, let a consumer read from Kafka, write results to Kafka, and commit its offset, all atomically, so within Kafka nothing is lost or duplicated. The moment processing touches the outside world, such as sending an email or calling an API, you're back to at least once.

What you can always achieve is **effectively once**: at-least-once delivery plus idempotent processing (chapter 10), so a duplicate changes nothing.

EXAMPLE events.py, with the confluent-kafka client

```
from confluent_kafka import Consumer, Producer

producer = Producer({
    "bootstrap.servers": BROKERS,
    "enable.idempotence": True,      # the broker drops duplicates from producer retries
    "acks": "all",                  # wait for every in-sync replica
})
producer.produce("invoice-events", key=invoice_id, value=json.dumps(event))
producer.flush()

consumer = Consumer({
    "bootstrap.servers": BROKERS,
    "group.id": "reminder-service",
    "enable.auto.commit": False,    # we decide when a message counts as done
    "auto.offset.reset": "earliest",
})
consumer.subscribe(["invoice-events"])
while True:
    msg = consumer.poll(1.0)
    if msg is None or msg.error():
        continue
    handle(json.loads(msg.value())) # must be idempotent: a crash below means a replay
    consumer.commit(message=msg)
```

Operational realities

- **Consumer lag**, how far a group is behind the newest message, is the metric to alert on. Adding consumers beyond the partition count does nothing; add partitions early, because changing the count later changes which partition each key maps to.
- **Poison messages** that always fail will block their partition forever. After a few attempts, move them to a dead-letter topic with the error attached, and carry on.
- **Hot partitions** come from skewed keys, exactly as in chapter 5.
- **Schemas are contracts**. Many teams consume your events. Register schemas and enforce compatibility rules, as you would for a public API (*The Engineer's Shipyard*, chapter 3).

COMMON TRAPS

Buying "exactly once". Side effects outside the log are still at least once.

Auto-committing offsets. Progress is recorded before the work is done, so crashes lose messages.

Too few partitions. Consumer parallelism has a hard ceiling you chose on day one.

Keys that don't match the ordering you need. Events for one invoice split across partitions arrive out of order.

No dead-letter handling. One bad message stops everything behind it.

EXERCISES

EXERCISE 11.1 Queue or log?

Choose a queue or a log for each, and say why.

- Generating thumbnails for uploaded logos.
- Feeding every invoice change to the search index, the analytics warehouse and the audit service.
- Building a brand-new reporting view from the last year of invoice events.
- Sending one welcome email per sign-up.

SOLUTION

- a. **Queue.** Independent tasks, any worker, no need to keep history.
- b. **Log.** Three different consumers each need every change, in order, at their own pace.
- c. **Log.** Only a log can replay a year of history into a new consumer.
- d. **Queue,** or a log if sign-ups are already published there for other consumers. Either way, the email job must be idempotent.

EXERCISE 11.2 Choose the key

Invoice events, created, sent, paid and voided, must be processed in order for each invoice. The largest team has two million invoices. Should the message key be the invoice id or the team id?

SOLUTION

The invoice id. It gives exactly the order required, per invoice, and spreads load evenly across partitions. Keying by team id would give per-team order, which nobody asked for, and would send every event of the two-million-invoice team to a single partition, which becomes a hot spot. Only choose the team id if some consumer genuinely needs a team's events in one global order.

EXERCISE 11.3 The duplicate email

A consumer reads an `InvoiceSent` event, emails the client, and crashes before committing its offset. What happens next, and how do you stop the client receiving the same email twice?

SOLUTION

On restart, the group resumes from the last committed offset, so the event is delivered again and the email is sent again.

Make the handler idempotent. Before sending, record the event id in a table with a unique constraint, in the same transaction as marking the email as due; skip if it's already there. Pass an idempotency key derived from the event id to the email provider, so a send that happened just before the crash isn't repeated. As chapter 12 of *The Engineer's Shipyard* explains, if the provider has no idempotency keys, there's a tiny window you must choose to fail in one direction or the other.

CHECKLIST

- ☐ Are offsets or acknowledgements committed only after the work is done?
- ☐ Is every consumer idempotent?
- ☐ Do message keys give exactly the ordering you need, without hot spots?
- ☐ Is consumer lag monitored, with dead-letter handling for poison messages?
- ☐ Are event schemas versioned and checked for compatibility?

GO DEEPER

- ESSAY** The Log: What every software engineer should know about real-time data's unifying abstraction Jay Kreps, 2013

- ESSAY** Exactly-once semantics are possible: here's how Kafka does it Neha Narkhede, 2017

- NEWS** Apache Kafka 4.0 release announcement March 2025

Event-driven architecture, CQRS and event sourcing

3 exercises

Events let services react to what happened without calling each other; CQRS lets you shape a separate read model for each kind of query; event sourcing makes the history itself the source of truth. Each is powerful, each is a long-term commitment, and together they can multiply complexity.

Commands and events

A **command** asks for something to happen: `SendInvoice`. It's addressed to one handler, named in the imperative, and may be refused. An **event** records that something happened: `InvoiceSent`. It's named in the past tense, broadcast to anyone interested, and can't be refused, because it's already true. Keeping the two apart in names and in design avoids a lot of confusion about who is responsible for what.

Four things people mean by "event-driven"

Martin Fowler's 2017 essay separates four patterns that share the name.

- **Event notification.** A small event says something changed; consumers call back for details. Loose coupling, but the call-back couples them again.
- **Event-carried state transfer.** The event carries the data, so consumers keep their own copies and never call back. More autonomy and resilience, at the cost of duplicated, eventually consistent data.
- **Event sourcing.** The events are the system of record.
- **CQRS.** Separate models for writing and for reading.

The benefits are real. Producers don't wait for consumers. New consumers can be added without touching the producer. A slow consumer doesn't slow the producer. The costs are also real: the flow of the system is no longer visible in any one place, data is eventually consistent everywhere, events become contracts that are hard to change, and debugging a problem means tracing it across services (chapter 20).

CQRS

Command Query Responsibility Segregation separates the model that handles commands, which validates them and enforces the business rules, from one or more models that answer queries, each shaped for the query it serves. The invoicing app's write model is

normalised and strict. Its dashboard reads from a denormalised table of totals per team, its search reads from a search index, and finance reads from a warehouse, all kept up to date from the stream of changes. Each read model can use the storage that suits it and can be rebuilt from scratch.

You may already do a light version: a read replica, or a materialised view refreshed from the main tables. Full CQRS is worth it when read and write shapes differ sharply and read load dominates.

Event sourcing

Instead of storing each invoice's current state and overwriting it, store the sequence of events that happened to it, and compute the current state by replaying them. The event store is append-only; state is a fold over history.

EXAMPLE `invoice_events.py` · state is a fold over events

```
from dataclasses import dataclass

@dataclass
class Invoice:
    id: str
    status: str = "draft"
    total_minor: int = 0
    paid_minor: int = 0
    version: int = 0

def apply(invoice: Invoice, event: dict) -> Invoice:
    kind = event["type"]
    if kind == "InvoiceCreated":
        invoice.total_minor = event["total_minor"]
    elif kind == "InvoiceSent":
        invoice.status = "sent"
    elif kind == "PaymentRecorded":
        invoice.paid_minor += event["amount_minor"]
        invoice.status = "paid" if invoice.paid_minor >= invoice.total_minor else
            "partially_paid"
    elif kind == "InvoiceVoided":
        invoice.status = "void"
    invoice.version += 1
    return invoice

def load(store, invoice_id: str) -> Invoice:
    invoice = Invoice(id=invoice_id)
    for event in store.events_for(invoice_id):
        invoice = apply(invoice, event)
    return invoice

def record_payment(store, invoice_id: str, amount_minor: int) -> None:
    invoice = load(store, invoice_id)
    if invoice.status in ("draft", "void"):
        raise InvalidCommand(f"can't record a payment on a {invoice.status} invoice")
    store.append(invoice_id, {"type": "PaymentRecorded", "amount_minor": amount_minor,
                             expected_version=invoice.version})  # fails if someone appended meanwhile
```

What you gain: a complete audit trail for free, the ability to answer "what did this invoice look like last Tuesday?", and read models that can be rebuilt, or new ones created, by replaying history, which turns a buggy projection into a fix-and-replay rather than a data repair. The `expected_version` on append gives optimistic concurrency, exactly as in *The Engineer's Shipyard*.

What you pay: events are permanent, so every schema mistake lives forever and old versions must be translated as they're read, a technique called **upcasting**. Aggregates with long histories need **snapshots** to load quickly. Deleting personal data for privacy laws needs special techniques, such as encrypting each person's data with their own key and destroying the key, called crypto-shredding. And most developers need time to think this way. Event sourcing suits domains where history is the domain: ledgers, bookings, anything audited. For ordinary records, it's usually too much.

Change data capture

A lighter way to get a stream of changes is to read the database's own replication log with a tool such as Debezium and publish every row change. It needs no changes to application code. But raw row changes expose your internal schema to every consumer, so publishing deliberate events through an outbox table, which Debezium can also relay, keeps the contract under your control.

COMMON TRAPS

Events named like commands, or commands broadcast like events.

Event sourcing everything. CRUD data gains nothing from a permanent history, and pays the full cost.

Rewriting history in place instead of upcasting old events on read.

Raw change data capture as a public contract. Every schema change breaks consumers.

Invisible flows. Nobody can say what happens after an event is published, because nobody wrote it down.

EXERCISES

EXERCISE 12.1 Command or event?

Classify each name as a command or an event. For each command, name the event it produces when it succeeds.

`ChargeCard` , `CardCharged` , `SendReminder` , `InvoiceBecameOverdue` ,
`CancelSubscription` , `ReminderSent`

SOLUTION

- `ChargeCard` : command, producing `CardCharged` , or `CardChargeFailed` if refused.
- `CardCharged` : event.
- `SendReminder` : command, producing `ReminderSent` .
- `InvoiceBecameOverdue` : event, emitted by a scheduled check, not requested by anyone.
- `CancelSubscription` : command, producing `SubscriptionCancelled` .
- `ReminderSent` : event.

EXERCISE 12.2 Upcast an old event

Version 1 of `InvoiceCreated` stored `"amount"` as a float number of dollars. Version 2 needs `total_minor` as integer cents and an explicit `currency` . The store holds millions of version 1 events. Write an upcaster that converts them as they're read.

SOLUTION

```
def upcast(event: dict) -> dict:
    if event["type"] == "InvoiceCreated" and event.get("schema", 1) == 1:
        return {
            "type": "InvoiceCreated",
            "schema": 2,
            "total_minor": to_minor_units(str(event["amount"]), "USD"),
            "currency": "USD",          # every version 1 event was in dollars
        }
    return event
```

Every read passes events through `upcast` before `apply` , so the rest of the code only knows version 2. Converting through a string and `Decimal` avoids the float trap: 19.99 becomes 1999, not 1998. The stored events are never changed, so the conversion can be fixed later if it proves wrong.

EXERCISE 12.3 Design the read models

The dashboard shows each team's overdue invoices with their total, and each client's page shows their payment history. Which events feed which read model? And what do you do when you discover a bug in how the overdue total was calculated?

SOLUTION

- **Overdue per team:** consumes `InvoiceBecameOverdue` to add an invoice, and `PaymentRecorded` and `InvoiceVoided` to reduce or remove it.
- **Client payment history:** consumes `PaymentRecorded` and refund events, keyed by client.
- **Fixing the bug:** correct the projection code, create a new, empty read model, replay the full event history into it, and switch the dashboard over once it has caught up. Then delete the old one. No data repair scripts, no guessing what the right totals were: the history is the truth, and the projection is disposable.

CHECKLIST

- ☐ Are commands and events distinct in name and in handling?
- ☐ Is event sourcing used only where history is the domain?
- ☐ Are old events upcast on read rather than rewritten?
- ☐ Can every read model be rebuilt from the log?
- ☐ Is the flow of events documented, so someone can say what happens after each one?

GO DEEPER

ESSAY What do you mean by "Event-Driven"? Martin Fowler, 2017

ESSAY Event Sourcing Martin Fowler

ESSAY CQRS Martin Fowler

DOCS Outbox event router Debezium

Batch and stream processing

3 exercises

Batch jobs process a bounded dataset and can take their time. Stream processors handle unbounded data as it arrives, and must decide when they've seen enough. The hard part of streaming is time: events arrive late and out of order, and your windows must cope.

Batch processing

A batch job reads a fixed input, such as yesterday's events or a snapshot of a table, and produces an output: a report, a search index, training data. Google's 2004 MapReduce paper made the model famous: a **map** step transforms each record and emits key-value pairs, the framework groups everything by key across the cluster, called the **shuffle**, and a **reduce** step combines each group. Modern engines such as Apache Spark generalise it into arbitrary graphs of operations, and much batch work now happens as SQL inside a data warehouse.

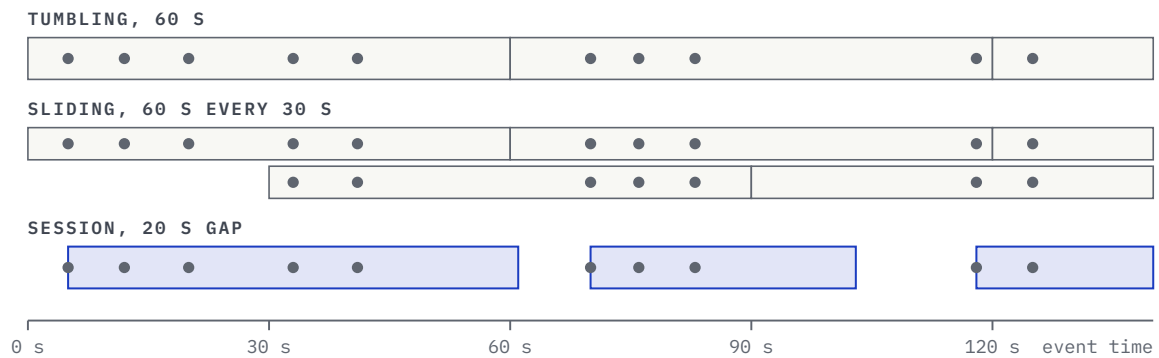
Batch has a wonderful property: inputs are immutable and jobs are deterministic, so recovery is simply running the job again. A bug in yesterday's report is fixed by fixing the code and re-running it.

Event time and processing time

A stream processor works on data that never ends, and must produce results before it has seen everything. Two clocks matter. **Event time** is when something happened, stamped at the source. **Processing time** is when the system gets to it. They can differ by milliseconds or by days: a phone that records usage offline uploads it when it finds a signal, and a failed consumer catches up on an hour of backlog. Any result about *when things happened*, such as invoices sent per hour, must use event time, or it changes depending on how busy the system was.

Windows

To aggregate an unbounded stream, you group it into finite windows of event time.



The same ten events grouped three ways. Tumbling windows never overlap. Sliding windows overlap, so an event can count in two. Session windows are shaped by the data itself, closing after 20 seconds without an event.

- **Tumbling** windows are fixed and consecutive: invoices sent per hour.
- **Sliding** or hopping windows overlap: payments in the last five minutes, updated every minute.
- **Session** windows group bursts of activity separated by quiet gaps: one user's visit to the dashboard.

Watermarks and late data

When is the window from 10:00 to 11:00 complete? Never, strictly: an event from 10:59 might still be on a phone somewhere. A **watermark** is the system's running estimate that no more events older than some time T will arrive. When the watermark passes the end of a window, the window's result is emitted. An event that arrives after its window was emitted is **late**, and must be dropped, sent to a side output for correction, or used to update the published result. Google's 2015 Dataflow paper, the basis of Apache Beam, framed this as a three-way trade-off between correctness, latency and cost, which you choose per pipeline.

EXAMPLE windows.py · tumbling counts with a watermark

```
from collections import defaultdict

class TumblingCounter:
    """Counts events per window of event time, tolerating stragglers."""

    def __init__(self, window: int = 60, allowed_lateness: int = 30):
        self.window, self.lateness = window, allowed_lateness
        self.open = defaultdict(int)      # window start -> running count
        self.newest = 0                    # highest event time seen so far
        self.late = []                     # events that arrived after their window closed

    def watermark(self) -> int:
        return self.newest - self.lateness    # we expect nothing older than this

    def add(self, event_time: int) -> list[tuple[int, int]]:
        start = event_time - event_time % self.window
```

```

    if start + self.window <= self.watermark():
        self.late.append(event_time)      # its window has already been emitted
        return []
    self.open[start] += 1
    self.newest = max(self.newest, event_time)
    return self._emit_closed()

def _emit_closed(self) -> list[tuple[int, int]]:
    closed = sorted(s for s in self.open if s + self.window <= self.watermark())
    return [(s, self.open.pop(s)) for s in closed]

```

This watermark is the simplest kind: "the newest event minus a fixed allowance". Real engines compute watermarks from their sources, and can emit early, speculative results and then refine them as late data arrives.

State and exactly-once in streams

Stream processors keep state: counts, windows, joins. To survive crashes, engines such as Apache Flink periodically snapshot every operator's state together with the input offsets it corresponds to, using a variant of the Chandy–Lamport distributed snapshot algorithm. After a crash, everything rolls back to the last snapshot and replays from those offsets. That gives exactly-once updates to the *state*. Results written to the outside world are exactly-once only if the sink is transactional or idempotent, which is the lesson of chapter 11 again.

Do you need a stream processor?

Probably not yet. Many "real-time" needs are satisfied by a scheduled query every five minutes over a warehouse fed by change data capture. Reach for stream processing when freshness is worth the operational cost: fraud detection that must act within seconds, alerting, live operational dashboards, or feeding derived data to user-facing features. Architectures once kept a batch layer and a streaming layer side by side, called the lambda architecture; many now use a single streaming path with replay from the log for recomputation, called the kappa architecture.

COMMON TRAPS

Windows on processing time. Results shift whenever the system falls behind.

No plan for late data. It's silently dropped, and reports quietly undercount.

Unbounded state. Session windows that never close, or joins that keep everything forever.

Assuming the engine's exactly-once covers your database. It covers its own state.

Streaming what a nightly query could do. Freshness nobody needed, operations everybody pays for.

EXERCISE 13.1 Which clock?

Should each use event time or processing time?

- Invoices sent per hour, for a monthly billing report.
- An alert when the payment-events backlog exceeds ten minutes.
- Daily active users of the mobile app, whose phones often sync hours later.
- Blocking a card after five failed payment attempts within one minute.

SOLUTION

- Event time.** The report is about when invoices were sent.
- Processing time.** The alert is about the system itself, right now.
- Event time**, with generous lateness or later corrections, since much of the data arrives hours late.
- Event time**, measured when the attempts happened. Using arrival time would let a delayed batch of attempts look spread out, or bunch up innocent ones.

EXERCISE 13.2 Run the counter by hand

Feed `TumblingCounter`, with a 60-second window and 30 seconds of allowed lateness, the events 10, 20, 65, 50, 95, 55 and 130, in that order of arrival. What does it emit, what's late, and what's still open at the end?

SOLUTION

- 10 and 20 open the window starting at 0. The watermark is still negative.
- 65 opens the window at 60 and moves the watermark to 35.
- 50 is out of order but its window, 0 to 60, hasn't closed, since 60 is above 35, so it's counted: three events in window 0.
- 95 moves the watermark to 65, which passes the end of window 0, so the counter emits (0, 3).
- 55 belongs to window 0, which has been emitted, so it's **late**.
- 130 opens window 120 and moves the watermark to 100, which hasn't passed 120, the end of window 60.

Emitted: (0, 3). Late: 55. Still open: window 60 with 2 events, and window 120 with 1.

EXERCISE 13.3 Choose the lateness

Mobile usage events arrive with 99% within two minutes of happening, and 99.9% within six hours. Product wants hourly usage charts that are both fresh and accurate. What do you do?

SOLUTION

Neither single setting works. Two minutes of lateness gives fresh charts that miss about 1% of events; six hours gives accurate charts that are six hours stale.

Emit each hour's result about two minutes after the hour, labelled provisional, and keep the window's state for six hours, updating the published figure as late events arrive. Or publish the fresh figure from the stream, and correct it with a nightly batch job over the complete data. Either way, show the chart's freshness and completeness honestly.

CHECKLIST

- ☐ Are time-based results computed on event time?
- ☐ Is there an explicit policy for late data?
- ☐ Is all stream state bounded?
- ☐ Are sinks transactional or idempotent?
- ☐ Would a scheduled query be good enough?

GO DEEPER

ESSAY Streaming 101 Tyler Akidau · event time, windows and watermarks

BOOK Streaming Systems Akidau, Chernyak & Lax

PAPER MapReduce: Simplified Data Processing on Large Clusters Dean & Ghemawat, 2004

PART V

Surviving failure

Three chapters on keeping a distributed system up when parts of it misbehave: the reaction to slowness and overload, the layer that spreads load and contains damage, and the caches that become load-bearing without anyone deciding they should.

Timeouts, retries and overload

3 exercises

Most large outages aren't caused by a component failing, but by the system's reaction to it: retries that multiply load, queues that grow without limit, and slow calls that tie up everything behind them. Design the reaction to failure as carefully as the happy path.

How failures cascade

A dependency gets slow. Every caller's requests to it now take longer, so each caller holds more threads and connections open at once. Callers run out of them and slow down in turn, and their callers pile up behind them. Within minutes, one slow database makes the whole product unresponsive, including features that never touch it. Slowness spreads further than failure does, because a fast error releases resources and a slow success holds them.

Timeouts and deadlines

Every call to another machine needs a timeout (chapter 2). Better still, pass a **deadline** down the chain of calls: not "each hop gets five seconds" but "the user's request must finish by 12:00:03.200". Each service subtracts its own work and passes the remainder on, and anything that would finish after the deadline is abandoned rather than wasted. gRPC propagates deadlines this way by default; with HTTP, pass the remaining budget in a header.

Retries, with a budget

The previous volumes covered retrying transient failures with exponential backoff and jitter. At the scale of many services, two further rules matter.

- **Retry at one layer.** If four layers each make up to four attempts, one user request can become $4 \times 4 \times 4 \times 4 = 256$ requests to the bottom service, arriving precisely when it's already failing.
- **Use a retry budget.** Allow retries only while they're a small fraction of traffic, for example no more than 10% of requests from a client. When the dependency is healthy, occasional failures get retried; when it's down, retries stop adding load. Google's SRE book describes this as a per-client retry budget.

Circuit breakers

Michael Nygard's *Release It!* popularised the circuit breaker. It watches calls to a dependency. While things are healthy, the circuit is **closed** and calls pass through. After enough consecutive failures it **opens**, and calls fail immediately without touching the dependency, which gets room to recover while callers stop wasting threads on it. After a cooldown it lets a probe call through, the **half-open** state: success closes the circuit, failure opens it again.

EXAMPLE breaker.py

```
import time

class CircuitOpen(Exception):
    pass

class CircuitBreaker:
    def __init__(self, failure_threshold=5, cooldown_s=30.0, clock=time.monotonic):
        self.failure_threshold, self.cooldown_s, self.clock = failure_threshold, cooldown_s, clock
        self.failures = 0
        self.opened_at = None  # None means the circuit is closed

    def call(self, fn, *args, **kwargs):
        if self.opened_at is not None and self.clock() - self.opened_at < self.cooldown_s:
            raise CircuitOpen()  # fail fast; don't pile on
        # Closed, or cooling down is over (half-open): let this call through.
        # A production breaker admits one probe at a time while half-open.
        try:
            result = fn(*args, **kwargs)
        except Exception:
            self.failures += 1
            if self.opened_at is not None or self.failures >= self.failure_threshold:
                self.opened_at = self.clock()  # trip, or re-trip after a failed probe
                raise
            self.failures, self.opened_at = 0, None  # success closes the circuit
        return result
```

When the circuit is open, have a fallback ready: cached data, a default, a degraded feature, or a clear error. An open breaker with no fallback just moves the failure.

Bulkheads, load shedding and backpressure

- **Bulkheads**, named after a ship's watertight compartments, give each dependency its own pool of threads or connections. A slow PDF service can exhaust its own pool, and invoice pages keep working.
- **Load shedding** means refusing work you can't finish, early and cheaply, with `503` and `Retry-After`, rather than accepting it and timing out later. Shed by priority: keep payments and sign-ins working, drop analytics and exports first. Adaptive concurrency limits, which adjust the number of requests in flight based on observed latency, do this automatically.

- **Backpressure** means a slow consumer pushes back on its producer instead of silently buffering. Every queue in the system should be bounded. An unbounded queue turns overload into ever-growing latency and, eventually, running out of memory.

Hedged requests

Jeff Dean and Luiz André Barroso's 2013 article *The Tail at Scale* described a cheap cure for slow outliers. Send a request to one replica; if it hasn't answered within the 95th-percentile latency, send the same request to a second replica, and use whichever answers first. Only about 5% of requests are sent twice, and the slowest requests become dramatically faster; in their measurements on Bigtable, the 99.9th-percentile latency fell from over a second to well under a hundred milliseconds. Use it only for idempotent reads, and cancel the loser.

Metastable failures

Sometimes a system stays down after the cause has gone. A 2021 paper by Nathan Bronson and colleagues named these **metastable failures**. A brief trigger, such as a load spike or a cache node restarting, pushes the system into overload, and something then sustains the overload even after the trigger ends: retries from timed-out clients, a cache that can't refill because the database is too busy to answer, or work queued for requests nobody is waiting for any more. The system is stable in a bad state. Recovery needs a deliberate cut in load, by shedding traffic, disabling retries or draining queues, below the level that triggered it, then bringing traffic back gradually.

COMMON TRAPS

Retries at every layer. Load multiplies exactly when the system is weakest.

Unbounded queues. Overload becomes latency, then memory exhaustion.

Circuit breakers without fallbacks. Fast failure, but still failure.

Hedging writes. A duplicated charge is not a latency optimisation.

Restoring full traffic at once after an outage, and re-triggering it.

EXERCISES

EXERCISE 14.1 Count the retries

A request passes through an API gateway, the invoice service, the pricing service and the database driver. Each layer retries a failed call three times. In the worst case, how many requests reach the database for one user action? What would you change?

SOLUTION

Each layer makes up to four attempts, the original plus three retries, so the database sees up to $4 \times 4 \times 4 \times 4 = 256$ requests for one click.

Retry in one place, normally the layer closest to the user that knows whether the operation is idempotent, and make every other layer fail fast. Add a retry budget so retries stop when failures are widespread, propagate the deadline so no retry outlives the user's patience, and put a circuit breaker in front of the database's callers.

EXERCISE 14.2 Trace the breaker

Using `CircuitBreaker` above with a threshold of 3 and a 30-second cooldown, trace this sequence of calls, giving the time of each, and say what the caller sees: fail at 0 s, fail at 1 s, succeed at 2 s, fail at 3 s, fail at 4 s, fail at 10 s, a call at 20 s, a failing call at 45 s, a succeeding call at 80 s.

SOLUTION

1. 0 s, 1 s: failures 1 and 2. Still closed.
2. 2 s: success resets the count.
3. 3 s, 4 s, 10 s: failures 1, 2, 3. The third trips the breaker, open from 10 s.
4. 20 s: only 10 seconds into the cooldown, so `CircuitOpen` is raised and the dependency isn't called.
5. 45 s: the cooldown is over, so the probe goes through, fails, and the breaker re-opens from 45 s.
6. 80 s: 35 seconds later, a probe goes through and succeeds, and the breaker closes.

EXERCISE 14.3 Survive the spike

A customer's buggy script suddenly sends the invoice API ten times its normal traffic. Design the protections, from the edge inwards, so that other customers barely notice.

SOLUTION

1. **Per-tenant rate limits** at the edge, returning 429 with `Retry-After` , so one tenant can't consume shared capacity (*The Engineer's Shipyard*, chapter 16).
2. **Concurrency limits** per service, shedding low-priority work first: exports and analytics before invoice views, and never payments.
3. **Bulkheads** around slow dependencies such as PDF rendering, so their pools fill without starving the rest.
4. **Bounded queues** for background jobs, with the tenant's jobs limited separately.
5. **An alert** on the tenant's traffic anomaly, and a person who contacts the customer.

Shuffle sharding, from chapter 15, would further limit how many other customers share capacity with the noisy one.

CHECKLIST

- ☐ Do deadlines propagate through every call chain?
- ☐ Do retries happen at one layer, within a budget?
- ☐ Does every critical dependency have a circuit breaker and a fallback?
- ☐ Is every queue bounded, and is load shed by priority?
- ☐ Is there a plan for bringing traffic back gradually after an outage?

GO DEEPER

PAPER The Tail at Scale Jeff Dean & Luiz André Barroso, 2013

PAPER Metastable Failures in Distributed Systems Bronson et al., 2021

BOOK Addressing Cascading Failures Google SRE book

BOOK Release It!, 2nd edition Michael Nygard · stability patterns and antipatterns

Load balancing, discovery and isolation

3 exercises

Spreading requests evenly is the easy part. The hard parts are knowing which instances are healthy, removing them gracefully, and arranging the system so that one bad component, customer or deploy can only take down a small part of it.

Where balancing happens

A **layer 4** load balancer works on connections: it picks a backend for each TCP connection and forwards bytes without understanding them. It's fast and simple, and it balances connections, not requests, which matters when one long-lived connection carries thousands of requests. A **layer 7** balancer understands HTTP: it can route by path or header, balance each request separately, retry, and terminate TLS. Most application traffic goes through layer 7 proxies, such as Envoy, NGINX or a cloud load balancer, or through a sidecar proxy in a service mesh.

Choosing a backend

- **Round robin** takes turns. Fine when every request costs about the same.
- **Least outstanding requests** sends each request to the backend with the fewest in flight, which adapts to slow requests and slow machines.
- **Consistent hashing** sends the same key to the same backend, for cache affinity, at the cost of hot spots (chapter 5).
- **The power of two choices**: pick two backends at random, and send to the less loaded of the two. Michael Mitzenmacher showed that this tiny change makes the busiest backend's load dramatically closer to the average than purely random choice, and it avoids the stampede that happens when every balancer independently sends everything to the one "least loaded" backend. Envoy's least-request balancing uses it.

EXAMPLE `balancing.py` · random against two choices

```
import random

def max_load(servers: int, requests: int, choose, rng) -> int:
    load = [0] * servers
    for _ in range(requests):
        load[choose(load, rng)] += 1
    return max(load)

def pick_random(load, rng):
    return rng.randrange(len(load))
```

```
def pick_two(load, rng):
    a, b = rng.randrange(len(load)), rng.randrange(len(load))
    return a if load[a] <= load[b] else b

rng = random.Random(1)
max_load(100, 10_000, pick_random, rng)  # typically 120 to 135: some servers 30% over the
mean
max_load(100, 10_000, pick_two, rng)      # typically 101 or 102: almost perfectly even
```

Health: active, passive and gray

Active health checks probe each backend on a schedule and remove those that fail. **Passive** checks, called outlier detection in Envoy, watch real traffic and eject a backend whose error rate or latency is far worse than its peers', which catches the gray failures of chapter 2 that pass every probe. Use both, and cap how many backends can be ejected at once, so a problem that affects every backend, such as a bad dependency, doesn't make the balancer eject all of them.

Service discovery

Instances come and go constantly, so clients must look them up. The options range from DNS, which is simple but cached, sometimes for longer than you'd like; to a registry such as Consul or etcd that clients query or subscribe to; to platform-managed discovery such as Kubernetes Services; to a service mesh, where sidecar proxies handle discovery, balancing, mutual TLS and retries. A mesh is convenient, but its retries are one more layer of retries, so configure them with chapter 14 in mind.

Graceful shutdown

Every deploy removes instances, and doing it carelessly drops requests. The order matters: stop advertising the instance as ready, wait long enough for balancers to notice and stop sending new work, finish the requests already in flight within a deadline, close connections to databases and queues, and only then exit.

Containing the blast radius

The most important architectural question about failure isn't whether something will break, but how much breaks with it. Four techniques limit the damage.

- **Zones.** Run in at least three availability zones. Amazon's Builders' Library describes **static stability**: provision enough capacity in the surviving zones in advance that losing a whole zone needs no new machines and no action by the control plane, which may itself be struggling during the incident.

- **Cells.** Split the whole stack, services and data alike, into independent copies called cells, each serving a subset of customers. A bad deploy, a poison request or a corrupted database affects one cell.
- **Shuffle sharding.** Give each customer a random combination of a few workers from a larger pool. Two customers rarely share their *whole* combination, so a customer whose requests crash workers takes down its own shard while almost everyone else keeps at least one healthy worker.
- **Staged rollouts,** one cell or zone at a time, as in *The Engineer's Shipyard*, chapter 17.

And when parts are down, **degrade gracefully:** serve cached or slightly stale data, hide recommendations, switch to read-only mode, queue writes for later. A product that's partly working is very different, for its users, from one that's down.

COMMON TRAPS

Every balancer picks the least-loaded backend. They all pick the same one at the same moment.

Health checks that eject everything when a shared dependency fails.

Long DNS caching during a failover.

Killing instances without draining. Every deploy drops in-flight requests.

One big shared pool. One tenant's poison traffic reaches every worker.

EXERCISES

EXERCISE 15.1 Choose the algorithm

Choose a balancing approach for each.

- Stateless API servers whose requests vary from 5 ms to 2 seconds.
- Cache servers, where each key should always go to the same server.
- Servers holding long-lived WebSocket connections.
- A pool mixing large and small machines.

SOLUTION

- Least outstanding requests, or two random choices,** which adapt to uneven request costs.

- b. **Consistent hashing**, with virtual nodes for balance.
- c. **Least connections**, plus a way to rebalance, such as asking some clients to reconnect, when new servers join, since existing connections never move on their own.
- d. **Weighted** balancing, proportional to each machine's capacity.

EXERCISE 15.2 Do the shuffle-sharding arithmetic

A pool has 16 workers, and each tenant is assigned 4 of them at random. How many different combinations are there? If one tenant's requests crash all four of its workers, what share of other tenants lose all of theirs? Compare with splitting the 16 workers into four fixed groups of four.

SOLUTION

There are $C(16, 4) = 1,820$ combinations. A tenant loses all its workers only if it has exactly the same four, so about 1 in 1,820 of other tenants are fully affected, while many others lose one or two workers and carry on with the rest.

With four fixed groups, the crashing tenant takes down its whole group, and every tenant in that group, a quarter of all tenants, is fully affected. Same hardware, a very different blast radius.

EXERCISE 15.3 Order the shutdown

Put these shutdown steps in the right order for an instance being removed during a deploy: close database connections; receive the termination signal; wait for load balancers to stop sending traffic; mark the instance not ready; exit; finish in-flight requests, up to a deadline.

SOLUTION

1. Receive the termination signal.
2. Mark the instance not ready, so health checks start failing.
3. Wait for load balancers to notice and stop sending traffic, typically a few seconds.
4. Finish in-flight requests, up to a deadline.
5. Close database connections and other resources.
6. Exit.

The most common mistake is closing the listener immediately on the signal, before balancers have stopped routing to the instance, which turns every deploy into a burst of connection errors.

CHECKLIST

- ☐ Does balancing adapt to uneven request costs and slow instances?
- ☐ Do you use both active checks and outlier detection, with a cap on ejections?
- ☐ Do instances drain gracefully on shutdown?
- ☐ Can you survive the loss of a zone without new capacity or control-plane actions?
- ☐ What fraction of customers can one bad tenant, deploy or request affect?

GO DEEPER

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

ESSAY Workload isolation using shuffle-sharding [Amazon Builders' Library](#)

DOCS Supported load balancers [Envoy](#)

Caching at scale

3 exercises

A cache starts as an optimisation and quietly becomes load-bearing: one day the database can no longer survive without it. Design for the cache being wrong, cold or gone, because at scale it will be all three.

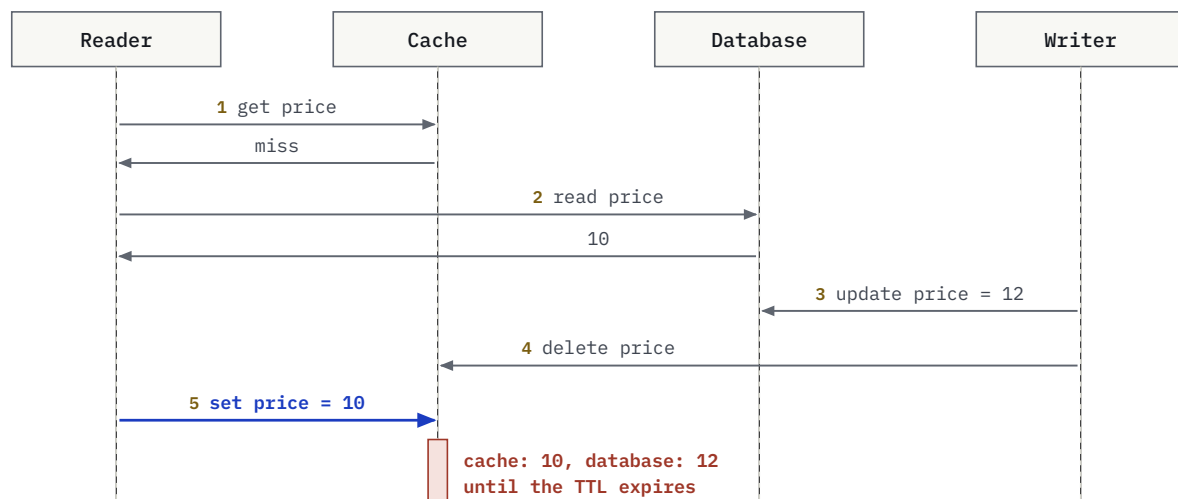
Where caches sit

Between a user and your database there may be five caches: the browser's, a CDN at the edge, an in-process cache inside each application server, a shared cache such as Redis or Memcached, and the database's own buffer pool. Each one closer to the user is faster and serves more traffic, and is harder to invalidate. This chapter is mostly about the shared cache, where the distributed problems live.

Reading and writing through a cache

The common pattern is **cache-aside**: on a read, try the cache; on a miss, read the database and store the result. The alternatives are **read-through**, where the cache itself loads from the database, **write-through**, where writes go to the cache and the database together, and **write-behind**, where the cache acknowledges writes and flushes them to the database later, which is fast and loses data when the cache fails.

With cache-aside, when data changes, **delete** the cache entry rather than setting it to the new value. Two writers that each set the cache can finish in the opposite order to their database writes and leave the older value cached. Deletes don't have that problem, and Facebook's Memcache paper chose them for the same reason: a delete can be repeated safely. Deleting isn't enough on its own, though, as the figure shows.



The reader fetched the old price before the update and stored it after the delete, so the cache now holds a value the database no longer has. A lease, or a version check when setting, rejects step 5; a TTL limits how long the damage lasts if everything else fails.

Leases

The Memcache paper, from Facebook in 2013, fixes this with **leases**. On a miss, the cache hands the reader a token for that key. A delete for the key cancels any outstanding token, and a set is accepted only with a token that's still valid. In the figure, the delete in step 4 cancels the reader's token, so step 5 is rejected and the next reader loads the new price.

The same tokens solve a second problem. The cache issues a token for a given key at most once every few seconds, ten by default in the paper, and tells other readers who miss in the meantime to wait briefly and try again. So when a popular key is deleted, one reader goes to the database while the rest wait a few milliseconds for it to refill the cache, instead of all of them querying the database at once.

Stampedes

That rush of simultaneous misses is a **cache stampede**, also called a thundering herd or dog-pile. It happens whenever a hot entry expires or is deleted, and it can overload the database at exactly the moment the cache has nothing to give it. There are four standard defences, and they combine well.

- **Coalesce requests.** Within one process, let the first miss for a key do the work and have concurrent misses wait for its result. Go's `singleflight` package does exactly this. Across processes, leases or a short lock in the cache do the same.
- **Serve stale while refreshing.** Keep the expired value, return it immediately and refresh in the background. HTTP has the same idea built in: RFC 5861's `stale-while-revalidate`, plus `stale-if-error` for serving the old copy when the origin is failing.

- **Refresh early, at random.** Let a request occasionally refresh a value shortly before it expires, with a probability that rises as expiry approaches, so one request refreshes it and the others never miss.
- **Add jitter to TTLs**, so that entries written together, for example after a deploy or a bulk import, don't all expire together.

EXAMPLE xfetch.py · probabilistic early refresh

```
import math
import random
import time

def xfetch(cache: dict, key, recompute, ttl_s: float, beta: float = 1.0,
           now=time.monotonic, rand=random.random):
    """Refresh a cached value early, with a probability that rises near its expiry."""
    entry = cache.get(key)
    if entry is not None:
        value, delta, expires_at = entry
        # log() of a number in (0, 1] is at most zero, so this acts as if it were a little
        # later than it is: a random head start, scaled by delta, the recompute time.
        if now() - delta * beta * math.log(1.0 - rand()) < expires_at:
            return value
    start = now()
    value = recompute()
    delta = now() - start
    cache[key] = (value, delta, now() + ttl_s)
    return value
```

This is **XFetch**, from a 2015 paper by Andrea Vattani, Flavio Chierichetti and Keegan Lowenstein. With a recomputation that takes 200 ms, a single request made one second before expiry refreshes the value with a probability under 1%; half a second before, about 8%; 200 ms before, 37%; 50 ms before, 78%. On a key read hundreds of times a second, some request almost certainly refreshes it just before it expires, while every other request keeps reading the cached value. Raising `beta` above 1 refreshes earlier.

When the cache becomes load-bearing

A cache with a 97% hit rate means the database sees only 3% of reads, and the database gets sized for that. Now the arithmetic runs the other way. If the hit rate falls to 90%, the database's load more than triples; if the cache is emptied by a restart, a flush or a change to the key format, the load rises more than thirtyfold. The database was never built for that, so it falls over, and the cache can't refill from a database that's down. This is the metastable failure of chapter 14 in its most common form.

- **Protect the database** with a limit on concurrent queries from cache misses, so a cold cache causes slow responses rather than a dead database.

- **Warm up gradually.** Facebook let a new, empty cache cluster read from a warm cluster's cache instead of from the database until its hit rate recovered. Restart cache nodes one at a time, and never flush everything at once.
- **Replace failed nodes carefully.** Rehashing a failed node's keys onto the survivors can overload them too. The Memcache paper sent the failed node's traffic to a small, separate pool called **Gutter**, whose entries expire quickly, instead.
- **Change key formats with a version prefix and a gradual rollout, never all at once.**

Hot keys and missing keys

A **hot key**, such as a celebrity's profile or the home page's configuration, can receive more traffic than one cache node can serve, and consistent hashing puts every request for it on that one node. Keep very hot keys in a small in-process cache with a short TTL, or store several copies under different suffixes, such as `home:1` to `home:8`, and have readers pick one at random.

At the other extreme, requests for keys that don't exist miss the cache every time and go to the database, which a scraper or an attacker probing random IDs will do deliberately. Cache "not found" results too, with a short TTL, or keep a Bloom filter of the keys that do exist and turn away the rest before they reach the database.

Invalidating across machines and regions

An application that writes to the database and then deletes the cache entry can crash between the two steps, and a region that deletes its own cache entry can't easily reach the caches in other regions. Facebook's answer was to drive invalidations from the database's commit log: a daemon beside each database reads committed changes and sends the matching deletes to the caches in its region. That's change data capture, from chapter 12, applied to caches. Invalidations then happen only after the change is committed, and survive application crashes. In other regions the same daemons run beside the database replicas, so caches there are invalidated only once the change has replicated, and no reader can refill them with the old value from a replica that hasn't caught up. Whatever the mechanism, give every entry a TTL, as the safety net that eventually corrects whatever the invalidations missed.

COMMON TRAPS

Setting the cache on write. Concurrent writers leave the older value behind.

A database sized for the hit rate. One cache flush takes it down.

Identical TTLs on entries written together. They all expire at the same moment.

Cooking entries with a common TTL. They all expire at the same moment.

Caching only hits. Missing keys go to the database every time.

Deciding from the cache. Balances, permissions and stock levels read from a cache can be stale.

EXERCISES

EXERCISE 16.1 Do the miss arithmetic

A service handles 50,000 reads a second through a cache with a 97% hit rate. The database handles 5,000 reads a second comfortably.

- How many reads a second reach the database?
 - A deploy changes some cache keys and the hit rate falls to 90%. Now how many?
 - The cache cluster restarts empty. Now how many, and what should have been in place beforehand?
-

SOLUTION

- 3% of 50,000: **1,500 a second**, well within capacity.
- 10% of 50,000: **5,000 a second**, more than three times as many and right at the limit. A seven-point drop in hit rate is a large change in database load.
- All **50,000 a second**, ten times what the database can handle.
Beforehand: a cap on concurrent miss queries so the database degrades rather than dies; restarting nodes one at a time; warming a new cluster from a warm one; and alerts on the hit rate itself, not just on latency.

EXERCISE 16.2 Find the race

This runs whenever a price changes. Describe how two concurrent calls can leave the wrong price in the cache for an hour, then fix it.

EXERCISE prices.py

```
def update_price(product_id, new_price):
    db.execute("UPDATE products SET price = %s WHERE id = %s", (new_price, product_id))
    cache.set(f"price:{product_id}", new_price, ttl=3600)
```

SOLUTION

Call A sets the price to 10 and call B sets it to 12. A updates the database, then B updates it, so the database says 12. Then B sets the cache to 12, and A, delayed by a pause or a slow network, sets it to 10. The cache says 10 for the next hour.

The fix is to delete the entry after the update, `cache.delete(f"price:{product_id}")`, and let the next reader load it. To close the remaining race from the figure, use leases if your cache supports them, or store a version alongside the value and only accept a set with a newer version, and keep a shorter TTL as the safety net. Better still, send the delete from the database's change stream, so it happens even if the application crashes after its update.

EXERCISE 16.3 Choose the strategy

Choose a caching approach for each.

- A product page viewed millions of times a day and edited a few times a day.
- A user's account balance, shown in the app and used to approve payments.
- Exchange rates, fetched every minute from a provider and used in every checkout.
- Lookups by user ID, where most of the IDs requested by scrapers don't exist.

SOLUTION

- The CDN**, with a long TTL, purged when the product is edited, for example by tagging responses with the product's ID, plus `stale-while-revalidate`. Keep per-user parts of the page out of the cached response.
- Display it from a short-lived cache if you must, but approve payments only from the database**, inside the same transaction that records the payment. A stale balance that allows an overdraft is a real loss.
- A cache refreshed ahead of expiry**, by a background job or early refresh, so no checkout waits for the provider. If the provider fails, keep using the last rate for a bounded time, alert, and stop quoting once the rate is too old to honour.

- d. **Negative caching** of "not found" with a short TTL, or a Bloom filter of existing IDs in front of the database, together with rate limits on the scrapers.

CHECKLIST

- ☐ Can the database survive an empty cache, and what limits its load while the cache refills?
- ☐ Do writes delete entries rather than set them, and does every entry have a TTL?
- ☐ Are hot keys protected from stampedes by coalescing, leases, early refresh or stale serving?
- ☐ Are "not found" results cached, briefly?
- ☐ Is every decision about money, permissions or stock made from the source of truth?

GO DEEPER

PAPER **Scaling Memcache at Facebook** Nishtala et al., NSDI 2013

PAPER **Optimal Probabilistic Cache Stampede Prevention** Vattani, Chierichetti & Lowenstein

ESSAY **Caching challenges and strategies** Amazon Builders' Library

SPEC **RFC 5861: stale-while-revalidate and stale-if-error** IETF

PART VI

Architecture and design

Four chapters on the decisions above any single mechanism: how to divide a system into services, how to find the boundaries in the business itself, how to design a system from a blank page, and how to find out whether the result really works.

Architecture styles

3 exercises

Architecture is the set of decisions that are expensive to change. Splitting a system into services buys independence for teams and pays for it on every request, in latency, failure modes and lost transactions, so split only where the independence is worth the price.

The main styles

STYLE	WHAT IT IS	SUITS	COSTS
Monolith	One deployable application with one database.	Small teams and new products.	Grows tangled without discipline; every change ships everything.
Modular monolith	One deployable, divided into modules with enforced boundaries, each owning its own data.	Most products, for most of their lives.	Boundaries need tooling to stay real; scales as one unit.
Microservices	Many small services, each deployed on its own and owning its own data.	Many teams that need to ship independently.	Network failures, no cross-service transactions, heavy operations.
Serverless functions	Code run per event by the platform, scaling to zero.	Spiky workloads and event-driven glue.	Cold starts, time limits, cost at steady high load.

What distribution costs

Everything in the first sixteen chapters is the price of putting a network between two pieces of code. A function call that always worked becomes a request that can fail, time out or succeed without telling you (chapter 2). Availabilities multiply along a chain of calls (chapter 1). A join becomes an API call, and a transaction becomes a saga (chapter 10). Data gets copied between services and is eventually consistent (chapter 12). Every interface between services needs versioning, as in *The Engineer's Shipyard*, chapter 3, and every service needs its own deploys, dashboards, alerts and on-call rotation.

What it buys

In return, each service can be deployed without coordinating with the others, scaled on its own, written in a different language, and isolated for security or compliance, so that only the payment service, for example, is in scope for a card-data audit. Nearly all of these benefits are really about people: microservices are mainly a way for many teams to work without waiting for each other. A team of eight rarely has that problem, and pays the full cost anyway.

Conway's law

In 1968 Melvin Conway observed that the systems an organisation designs end up mirroring its communication structure: four teams building a compiler produce a four-pass compiler. You can resist this or use it. The "inverse Conway manoeuvre" shapes the teams to match the architecture you want. *Team Topologies*, by Matthew Skelton and Manuel Pais, builds on the idea: most teams should be stream-aligned, owning one slice of the product end to end, supported by platform teams that make shared infrastructure self-service, with service boundaries drawn so that each team can hold its part in its head.

The modular monolith

Martin Fowler observed that almost every successful microservice system he'd heard of started as a monolith that grew too big, while systems built as microservices from scratch often got into serious trouble. The practical middle ground is a **modular monolith**: one deployable, divided into modules that each own their tables and expose a small public interface, with no reaching into another module's internals. Shopify took this route with its very large Rails application, and built a tool, Packwerk, to enforce the boundaries. If a module later needs to become a service, its boundary is already there.

The boundaries only stay real if a machine checks them. Tools exist for most languages, such as import-linter for Python, Packwerk for Ruby, ArchUnit for Java and dependency-cruiser for JavaScript, but the idea fits in a page.

EXAMPLE check_boundaries.py · run in CI

```
import ast
import pathlib
import sys

# Each package under app/ is a module. This lists which others each may use.
ALLOWED = {
    "billing": {"customers"},
    "customers": set(),
    "notifications": {"billing", "customers"},
}

def imports(path: pathlib.Path):
    tree = ast.parse(path.read_text(), filename=str(path))
    for node in ast.walk(tree):
        if isinstance(node, ast.Import):
            for alias in node.names:
                yield alias.name, node.lineno
        elif isinstance(node, ast.ImportFrom) and node.module and node.level == 0:
```

```

        for alias in node.names:
            yield f"{node.module}.{alias.name}", node.lineno

def violations(root: pathlib.Path):
    for path in sorted(root.rglob("*.py")):
        rel = path.relative_to(root).parts
        if len(rel) < 2:
            continue
        owner = rel[0]
        for name, line in imports(path):
            parts = name.split(".")
            if parts[0] != root.name or len(parts) < 2 or parts[1] == owner:
                continue
            target = parts[1]
            if target not in ALLOWED.get(owner, set()):
                yield f"{path}:{line}: {owner} may not depend on {target}"
            elif len(parts) > 2 and parts[2] != "api":
                yield f"{path}:{line}: {owner} imports {name}; use {target}.api"

if __name__ == "__main__":
    found = list(violations(pathlib.Path("app")))
    print("\n".join(found) or "module boundaries respected")
    sys.exit(1 if found else 0)

# app/billing/models.py:1: billing imports app.customers.models.Customer; use customers.api
# app/customers/api.py:1: customers may not depend on billing

```

The distributed monolith

The worst outcome is services that can't change independently: they share a database and read each other's tables, they must be deployed together, or one request passes through a long chain of synchronous calls. That's all the cost of distribution with none of the independence. If two services always change together, they're one service.

How services talk

Synchronous calls, over HTTP or gRPC, are simple to write and reason about, but they couple the caller's availability and latency to the callee's. **Asynchronous** messages and events, through a queue or a log (chapter 11), decouple services in time, so the consumer can be down for an hour without the producer noticing, at the cost of eventual consistency and harder debugging. Use synchronous calls when the caller needs the answer to continue, and events for everything that can happen afterwards. When a service keeps needing another's data, it can keep its own read-only copy, updated from that service's events, instead of calling it on every request.

Serverless, and choosing the grain

Serverless functions remove servers from your concerns and cost nothing when idle, which suits spiky and event-driven work. They have cold starts, time limits of minutes rather than hours, and a per-call price that can exceed a server's at steady high load. In 2023, Prime Video's engineers described moving one monitoring service, which checks video streams for defects, from serverless functions orchestrated by AWS Step Functions,

passing video frames between steps through S3, into a single process that passes them in memory. Its infrastructure cost fell by over 90%. The lesson isn't that one style is right, but that the split must match how data moves: steps that exchange large amounts of data many times a second belong in one process.

Migrating with a strangler fig

Rewriting a large system from scratch rarely works. Fowler's **strangler fig**, named after a vine that grows around a tree until it replaces it, migrates one piece at a time: put a routing layer in front of the old system, rebuild one capability in the new one, send that capability's traffic there, and delete the old code. Each step is small and reversible. Running the new code in shadow, on copies of real traffic with its results compared but not used, catches differences before any user sees them.

COMMON TRAPS

Splitting by technical layer, with a "database service" and a "validation service", instead of by business capability.

A shared database behind supposedly independent services.

Services that must be deployed together.

Long synchronous chains on the most important requests.

Microservices for a single small team, paying the whole cost for none of the benefit.

EXERCISES

EXERCISE 17.1 Monolith or services?

Recommend a structure for each.

- A four-person startup still searching for product-market fit.
- 300 engineers in 30 teams, sharing one application whose deploys queue behind each other and whose test suite takes two hours.
- A product where one feature, video transcoding, needs GPUs and scales very differently from the rest.
- A business that handles card numbers in one small part of its system.

SOLUTION

- a. **A modular monolith.** The product will change shape many times, and moving a boundary inside one codebase is a refactoring, while moving it between services is a migration.
- b. **Split, along the boundaries of the teams and the business,** starting with the parts that change most often or have their own scaling needs, using the strangler fig. The problem here is organisational, and that's exactly what services solve.
- c. **Extract that one component** as a separate worker service fed by a queue, and keep the rest together.
- d. **Isolate the card handling** in its own small service, or better, hand it to a payment provider, so that the audit scope is that one component rather than the whole system.

EXERCISE 17.2 Spot the distributed monolith

Six services have these symptoms. For each, explain the problem and propose a fix.

- a. Three services read and write the same Postgres schema.
- b. Deploying the orders service needs a same-day deploy of the pricing service.
- c. Loading the order page makes 14 sequential calls, each to a service with 99.9% availability and a 20 ms median latency.

SOLUTION

- a. **The schema is the real interface**, so no service can change its tables without breaking the others. Give each table a single owning service, and have the others use its API or a copy fed by its events.
- b. **The contract between them isn't backward compatible.** Make changes additive, as in *The Engineer's Shipyard*, chapter 3, and add contract tests that fail a build that breaks a consumer. If they still always change together, merge them.
- c. **Availability is $0.999^{14} \approx 98.6\%$** , so about 14 times the downtime of any single service, and the median latency is at least 280 ms, with the tails worse. Make independent calls in parallel, and serve the page from a read model built from events (chapter 12), so it needs one or two calls instead of fourteen.

EXERCISE 17.3 Plan a strangler fig

A legacy billing monolith generates invoice PDFs itself, slowly, and the team wants to move this to a new service. Plan the steps so that each one can be reversed.

SOLUTION

1. Put a routing layer in front of PDF generation, initially sending everything to the old code.
2. Build the new service, fed with invoice data from the monolith's events or change stream rather than its tables.
3. Run it in shadow: generate both PDFs, keep serving the old one, and compare them automatically until they match.
4. Route a small percentage of real traffic to the new service, then more, watching errors and latency, with the old path one switch away.
5. Once all traffic has run on the new service for a while, delete the old code and the routing rule.

CHECKLIST

- ☐ Does every service have one owning team and its own data?
- ☐ Can each service be deployed on its own, without a coordinated release?
- ☐ How many synchronous hops does your most important request make?
- ☐ Are module boundaries enforced by a tool rather than by memory?
- ☐ For each split, can you name the benefit it bought?

GO DEEPER

ESSAY [MonolithFirst](#) Martin Fowler

ESSAY [Deconstructing the Monolith](#) Shopify Engineering

ESSAY [Strangler Fig Application](#) Martin Fowler

ESSAY [How Do Committees Invent?](#) Melvin Conway, 1968

BOOK [Team Topologies](#) Matthew Skelton & Manuel Pais · team types and the boundaries between them

Domain-driven design

3 exercises

The hardest boundaries to get right are in the business, not in the code. Domain-driven design gives you tools for finding them: a shared language with the people who know the domain, contexts in which each word means one thing, and aggregates that say what must be consistent at once.

Start from the domain

Eric Evans's *Domain-Driven Design*, published in 2003, starts from an observation: in most business software, the hard part isn't the technology but the domain, meaning the rules of lending, logistics, insurance or whatever the software is for. The design has two halves. **Strategic design** finds the large boundaries in the business. **Tactical design** is a set of patterns for modelling inside one boundary. The strategic half matters most for distributed systems, because it tells you where the seams are.

A shared language

Developers and domain experts should use the same words for the same things, in conversation, in documents and in the code. Evans calls this the **ubiquitous language**. If the business says "the account is on hold" and the code says `user.status_flag = 3`, every conversation needs a translation, and translations are where misunderstandings get in. When the code uses the experts' words, an expert can read a test and say whether it's right.

Bounded contexts

A shared language breaks down at the scale of a whole company, because the same word means different things in different departments. To sales, a "customer" is a lead with a stage in the pipeline; to billing, a legal entity with a tax ID and a payment method; to support, a person with open tickets. One `Customer` class serving all three becomes a huge object that every team edits and no team owns.

A **bounded context** is a boundary within which one model applies and each word has one meaning. Sales, billing and support each have their own customer model, sharing only an identifier, and translate explicitly where they meet. Bounded contexts are the best candidates for the modules and services of chapter 17, because a boundary drawn where the language changes is one the business already respects.

How contexts relate

Evans's **context map** names the relationships between contexts, and naming them makes the politics visible.

- **Customer and supplier.** The upstream team plans its changes around the downstream team's needs.
- **Conformist.** The downstream team simply adopts the upstream model, typically because it has no influence over it.
- **Anticorruption layer.** The downstream team translates the upstream model into its own at the boundary, so a legacy system's or a vendor's quirks don't leak into its code. This is the right default for any third-party API.
- **Open host service and published language.** The upstream team offers one well-documented protocol for everyone, instead of a custom integration per consumer.
- **Shared kernel.** Two teams co-own a small piece of the model. Keep it very small, since every change needs both teams.

Where to invest

DDD sorts the parts of a business into three kinds of subdomain. The **core** is what makes this business different from its competitors, and deserves your best engineers and your most careful modelling. **Supporting** subdomains are specific to the business but don't set it apart, so build them simply. **Generic** subdomains, such as authentication, payments and email, are the same everywhere, so buy them or use open source. Much wasted effort goes into building a generic subdomain lovingly while the core is neglected.

Event storming

Alberto Brandolini's **event storming** workshop is a fast way to find the contexts. Domain experts and engineers stand at a long wall and write **domain events**, things that happened, in the past tense, such as "Order placed" or "Payment failed", on sticky notes, in time order. Then they add the commands that cause each event, the people who issue them, the policies that react ("whenever a payment fails, notify the customer"), and the open questions. Boundaries show up where the vocabulary changes and where work passes from one department to another.

Inside a context: the tactical patterns

- **Entities** have an identity that persists as their attributes change, such as a customer or an order.
- **Value objects** are defined entirely by their value and are immutable, such as an amount of money, an address or a date range. Two equal amounts are interchangeable.
- **Aggregates** are clusters of objects that are changed together, as one unit, through a single root object. The aggregate is the boundary of consistency: its rules hold at the end of every transaction.
- **Domain events** record that something the experts care about has happened.

- **Repositories** load and save whole aggregates, so the rest of the code never handles half of one.

EXAMPLE orders/model.py · a value object and an aggregate

```
from dataclasses import dataclass
from decimal import Decimal

@dataclass(frozen=True)
class Money:                                # value object: immutable, equal by value
    amount: Decimal
    currency: str

    def __add__(self, other: "Money") -> "Money":
        if other.currency != self.currency:
            raise ValueError("cannot add different currencies")
        return Money(self.amount + other.amount, self.currency)

    def times(self, n: int) -> "Money":
        return Money(self.amount * n, self.currency)

@dataclass(frozen=True)
class OrderPlaced:                          # domain event, named in the past tense
    order_id: str
    total: Money

class Order:                                # aggregate root: every change goes through it
    MAX_LINES = 50

    def __init__(self, order_id: str, currency: str):
        self.id, self.currency = order_id, currency
        self.lines: list[tuple[str, int, Money]] = []
        self.status = "draft"
        self.events: list[OrderPlaced] = []

    def add_line(self, sku: str, qty: int, unit_price: Money) -> None:
        if self.status != "draft":
            raise ValueError("order can no longer be changed")
        if qty <= 0 or len(self.lines) >= self.MAX_LINES:
            raise ValueError("invalid line")
        self.lines.append((sku, qty, unit_price))

    def total(self) -> Money:
        total = Money(Decimal("0"), self.currency)
        for _, qty, price in self.lines:
            total = total + price.times(qty)
        return total

    def place(self) -> None:
        if self.status != "draft" or not self.lines:
            raise ValueError("only a draft with lines can be placed")
        self.status = "placed"
        self.events.append(OrderPlaced(self.id, self.total()))

# The repository saves the order and its new events in one transaction (the outbox of
# chapter 10). Payment and stock are separate aggregates, which react to OrderPlaced.
```

Adding a line to a placed order, or adding euros to dollars, fails inside the model, wherever the call comes from. The rules live in one place instead of being re-checked, or forgotten, in every handler.

Aggregates in a distributed system

Aggregates matter far beyond object modelling, because an aggregate is the natural unit for almost everything in the earlier chapters. It's the unit of a transaction, and of optimistic locking with a version number. It's the natural partition key (chapter 5), since everything one transaction touches then lives on one shard. It's one stream in an event-sourced system (chapter 12). Rules that span aggregates, such as "reserve stock when an order is placed", are enforced eventually, through domain events, sagas and the outbox of chapter 10. Vaughn Vernon's advice for designing them is short: keep aggregates small, refer to other aggregates by ID, change one aggregate per transaction, and use eventual consistency between them.

When not to use it

An application that mostly stores and displays records, with few rules, doesn't need aggregates and repositories, and neither do generic subdomains. The tactical patterns are worth their weight only where the rules are complex. The strategic ideas, a shared language and clear contexts, are worth using almost everywhere.

COMMON TRAPS

One model for the whole company. Every "Customer" field is someone's, and every change is contested.

Aggregates as big as the database, which turn every change into a contended lock.

Invariants spread across services with no single place that enforces them.

Carefully modelling a generic subdomain while the core gets the leftovers.

All the tactical patterns on a simple CRUD application.

EXERCISES

EXERCISE 18.1 Find the contexts

In an online shop, four teams use the word "product". The catalogue team means a title, description, images and category. Pricing means a list price, discounts and a tax class. Inventory means stock levels per warehouse and a reorder point. Shipping means weight, dimensions and whether it's hazardous. What goes wrong with a single `Product` model, and what would you do instead?

SOLUTION

A single model couples four teams: every change needs agreement, every service loads fields it doesn't use, and words drift apart in meaning, until nobody knows whether "available" means in stock or published.

Instead, give each context its own model with only its own fields, sharing the product's identifier, such as the SKU: a catalogue entry, a price list entry, a stock item and a shipping specification. When one context needs to know about changes in another, it subscribes to events such as `ProductPublished` or `PriceChanged`, and keeps its own copy of what it needs.

EXERCISE 18.2 Draw the aggregate boundary

A conference booking system has one hard rule: a session never has more bookings than seats. Which aggregate design would you choose, and why?

- a. One `Conference` aggregate containing all its sessions and bookings.
- b. Each `Booking` as its own aggregate.
- c. A `Session` aggregate holding its capacity and its seat reservations.

SOLUTION

(c). It holds exactly the data the rule needs, so one transaction can check and update it.

Design (a) also enforces the rule, but every booking for the whole conference contends for one aggregate, so bookings for different sessions block each other. Design (b) can't enforce the rule at all, because checking capacity would mean reading other aggregates, which isn't atomic. Keep attendee details in their own aggregate, referenced by ID, and send confirmation emails in reaction to a `SeatReserved` event rather than inside the transaction.

EXERCISE 18.3 Classify the subdomains

A food delivery startup's system includes: dispatch and route optimisation; restaurant menu management; card payments; login and accounts; customer support ticketing; delivery-time prediction. Classify each as core, supporting or generic, and say what that means for how you'd build it.

SOLUTION

- **Core:** dispatch and route optimisation, and delivery-time prediction. The business competes on fast, reliable delivery, so these get the strongest engineers and careful modelling.
- **Supporting:** restaurant menu management. It has to work well and is specific to the business, but it won't win customers on its own, so build it simply.
- **Generic:** card payments, login and support ticketing. Use a payment provider, an identity provider and an existing helpdesk product, and put an anticorruption layer between each one and your own model.

CHECKLIST

- ☐ Does the code use the domain experts' words, each with one meaning per context?
- ☐ Can you draw the bounded contexts and how each pair relates?
- ☐ Is each invariant enforced inside one aggregate, in one transaction?
- ☐ Do aggregates refer to each other only by ID?
- ☐ Are you building the core yourself and buying the generic parts?

GO DEEPER

REFERENCE Domain-Driven Design Reference Eric Evans · the patterns in brief, free

ESSAY Bounded Context Martin Fowler

ESSAY Effective Aggregate Design Vaughn Vernon

BOOK Learning Domain-Driven Design Vlad Khononov · the most practical modern introduction

BOOK Introducing EventStorming Alberto Brandolini

System design method

3 exercises

Designing a system is a sequence of decisions, each justified by a requirement and a number. Clarify what's needed, estimate the load, sketch the simplest thing that meets it, then find where that breaks and deepen the design only there.

The method

Whether it's a design review at work or a system design interview, the same steps work, in this order.

- 01 Clarify the requirements.** The functional ones, what the system does, and the non-functional ones: scale, latency, availability, consistency, durability, cost and compliance. Ask which operations matter most, and which failures are unacceptable.
- 02 Estimate.** Requests per second at average and peak, data size and growth, bandwidth. The numbers decide which problems you actually have.
- 03 Define the interface and the data.** The main API calls and the main entities, with their access patterns: what's looked up by what.
- 04 Sketch the simplest design that meets the numbers.** Often that's one application tier, one database and a cache.
- 05 Find what breaks, and deepen there.** The bottleneck, the hot spot, the consistency requirement, the operation that can't be lost. This is where the earlier chapters come in.
- 06 Plan for failure and operations.** For each component: what happens when it's slow, down or wrong? How will you know? How do you deploy it?
- 07 State the trade-offs.** Every choice gives something up. Say what, and why that's acceptable here.

Estimating

Estimates only need to be right to within a factor of a few, which makes them quick. A day has 86,400 seconds, so a million events a day is about 12 a second. Peaks run at two to ten times the average. Here's an invoicing platform that sends 60 million invoices a month, three emails per invoice, and keeps a 150 KB PDF of each invoice for seven years, with a peak of five times the average.

QUANTITY

WORKING

RESULT

Invoices a day	$60\text{M} \div 30$	2 million
Average rate	$2\text{M} \div 86,400 \text{ s}$	$\approx 23 \text{ a second}$
Peak rate	$5 \times \text{average}$	$\approx 116 \text{ a second}$
Emails	$3 \times 2\text{M a day}$	6M a day, $\approx 69 \text{ a second}$
PDF storage	$60\text{M} \times 150 \text{ KB}$	9 TB a month, 108 TB a year
Seven years of PDFs	$7 \times 108 \text{ TB}$	$\approx 756 \text{ TB}$

The write rate is small, and one database handles it easily, so there's no case for sharding the invoice data. The email rate needs a queue and a provider, nothing heroic. The number that shapes the design is storage: three-quarters of a petabyte of PDFs, before replication. That belongs in object storage, with lifecycle rules that move old files to cheaper tiers, and it raises a question for the business: must every PDF be kept, or can it be regenerated from the invoice data when someone asks for it?

A worked design: a distributed rate limiter

Requirements. A public API runs on 40 servers behind a load balancer. Each API key may make 100 requests a second on average, with bursts of up to 200. The check must add at most a millisecond or two, should be accurate to within a few percent, and must not take the API down if it fails.

Estimate. At a peak of 50,000 requests a second, that's 50,000 checks a second, each a small read and write on one key: comfortably within a small Redis cluster, with keys spread across shards.

Where the state lives. Each server could enforce its own share, $100 \div 40 = 2.5$ requests a second per key, with no network call at all. But that's wrong whenever a client's traffic is spread unevenly across servers, which is common with long-lived connections, and the share changes whenever the fleet scales. A central store is accurate, at the cost of one round trip, about half a millisecond within a data centre, and of a new dependency.

The algorithm. A fixed window counter, one counter per key per second, is simplest, but allows twice the limit across a window boundary. A sliding log of timestamps is exact but stores every request. A **token bucket** stores two numbers per key and expresses exactly this requirement: a steady rate plus a burst.

Atomicity. The check reads the bucket, computes and writes it back. If two servers do that concurrently, both can spend the last token, a lost update from chapter 6. Redis runs a Lua script atomically, so the whole check goes into one script, which also uses Redis's clock instead of each server's.

EXAMPLE token_bucket.lua

```
-- token_bucket.lua    KEYS[1]: the bucket    ARGV: rate per second, burst, cost
local rate = tonumber(ARGV[1])
local burst = tonumber(ARGV[2])
local cost = tonumber(ARGV[3])

-- Use the Redis server's clock, so API servers with skewed clocks agree (chapter 3).
local t = redis.call("TIME")
local now = tonumber(t[1]) * 1000 + math.floor(tonumber(t[2]) / 1000)

local state = redis.call("HMGET", KEYS[1], "tokens", "ts")
local tokens = tonumber(state[1]) or burst
local ts = tonumber(state[2]) or now

-- Refill for the time since the last call; never go backwards if the clock does.
tokens = math.min(burst, tokens + math.max(0, now - ts) / 1000 * rate)

local allowed, wait_ms = 0, 0
if tokens >= cost then
    tokens = tokens - cost
    allowed = 1
else
    wait_ms = math.ceil((cost - tokens) / rate * 1000)
end

redis.call("HSET", KEYS[1], "tokens", tokens, "ts", now)
-- A full bucket is the same as no bucket, so let idle keys expire.
redis.call("PEXPIRE", KEYS[1], math.ceil(burst / rate * 1000))
return {allowed, wait_ms}
```

EXAMPLE limiter.py · the caller

```
import redis

r = redis.Redis(host="ratelimit.internal", socket_timeout=0.005) # 5 ms
token_bucket = r.register_script(open("token_bucket.lua").read())

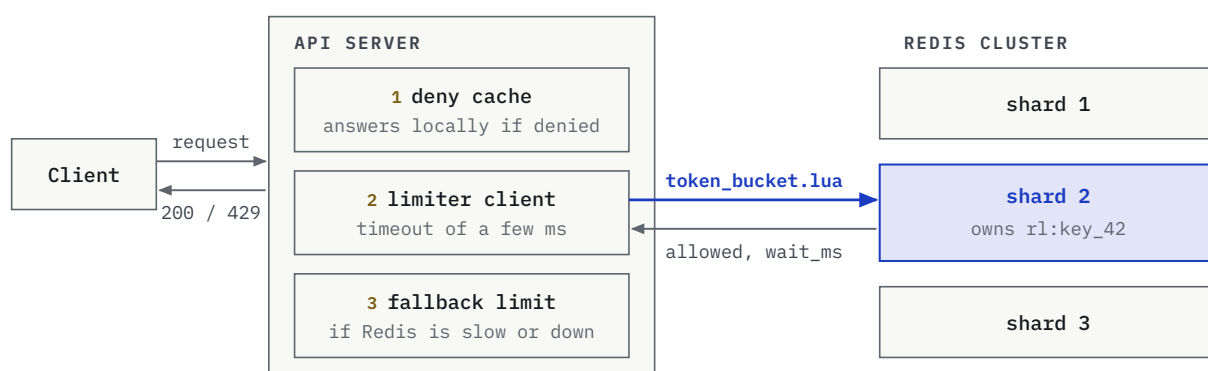
def check(api_key: str, local_limiter, rate=100, burst=200) -> tuple[bool, int]:
    """Returns (allowed, milliseconds until a token is available)."""
    try:
        allowed, wait_ms = token_bucket(keys=[f"rl:{api_key}"], args=[rate, burst, 1])
        return allowed == 1, wait_ms
    except redis.RedisError: # slow or down: fail open, up to a local cap
        ok = local_limiter.allow(api_key)
        return ok, 0 if ok else 1000
```

Deepening. With the basic design in place, the questions that matter are about the edges.

- **Abusive clients.** A rejected request still costs a round trip to Redis, so a client hammering the API with an exhausted key loads the shard that owns its key. Each server remembers denials locally until `wait_ms` has passed, and turns those requests away without asking Redis.
- **Redis failing.** The short timeout bounds the added latency. On an error, the server falls back to its local share of the limit, fails open within that cap, and alerts. An outage of the

limiter becomes a period of rough limits, not an outage of the API.

- **Failover.** Redis replicates asynchronously (chapter 4), so a failover can lose the last few updates and briefly give some clients extra requests. For a rate limiter that's acceptable. For an account balance it wouldn't be, which is why that would need a different design.
- **More regions.** A single global count would put a cross-region round trip, tens to hundreds of milliseconds (chapter 1), on every request. Give each region its own share of the limit instead, or synchronise regional counts in the background and accept brief overshoot.
- **Telling clients.** Reject with `429 Too Many Requests` and a `Retry-After` taken from `wait_ms`, as in *The Engineer's Shipyard*, chapter 16.



The finished design. Most requests make one round trip, to the shard that owns their key. Clients already over their limit are turned away inside the API server, and if Redis fails, each server enforces its own share of the limit.

Trade-offs. This design gives up a little accuracy, during failovers and fallbacks, for speed and availability, which is the right choice for a rate limiter and would be the wrong one for a ledger. Saying so explicitly is the last step of the method, and the one most often skipped.

COMMON TRAPS

Designing before asking. The requirement you didn't ask about is the one that decides the design.

No numbers. Sharding a database that handles 23 writes a second, or not sharding one that needs 50,000.

Starting complicated. Every box you add needs a reason from the requirements.

Only the happy path. The design review's first question is what happens when each part fails.

Hiding the trade-offs. A design that claims to give up nothing hasn't been examined.

EXERCISES

EXERCISE 19.1 Estimate a URL shortener

A URL shortener creates 100 million links a month, each read 100 times on average. Each record takes 500 bytes and is kept for ten years. Estimate the average write and read rates, the storage needed, and how many characters a base-62 short code needs. What do the numbers tell you about the design?

SOLUTION

- **Writes:** $100\text{M} \div (30 \times 86,400) \approx 39$ a second.
- **Reads:** 100 times that, $\approx 3,900$ a second on average, and several times more at peak.
- **Storage:** $100\text{M} \times 500 \text{ B} = 50 \text{ GB}$ a month, 600 GB a year, 6 TB over ten years.
- **Codes:** ten years is 12 billion links. $62^6 \approx 57$ billion, so six characters would be 21% full, which makes random codes collide increasingly often. $62^7 \approx 3.5$ trillion, so **seven characters** leave plenty of room.

The system is read-heavy and its data fits on a few machines, so the design centres on caching: a cache in front of the database, and redirects served at the edge. One subtlety: a 301 redirect is cached by browsers, so repeat visits never reach you, which is good for load and bad for click counting. Many shorteners use 302 for that reason.

EXERCISE 19.2 Pick the algorithm

Choose a rate-limiting algorithm for each, and say why.

- Five login attempts per account per 15 minutes.
- API requests per key, allowing short bursts.
- A daily quota of text messages per user, which users expect to reset at midnight.

SOLUTION

- a. A **sliding log**, or a sliding window counter. A fixed window would allow five attempts just before a boundary and five more just after, twice the limit in a few seconds, which matters for security. The log is only five timestamps per account.
- b. A **token bucket**, which expresses a steady rate plus a burst directly.
- c. A **fixed window aligned to the calendar**, in the user's time zone. Here the boundary effect is what users expect, not a flaw.

EXERCISE 19.3 Design a notification service

Sketch a service that sends push notifications, emails and text messages for other teams' services: around 20 million notifications a day, peaking at ten times the average, where some are urgent, such as login codes, and some are bulk, such as a weekly digest. Work through the method: requirements, estimate, the simplest design, then what breaks.

SOLUTION

Requirements: an API to send one notification or a batch, templates, user preferences and opt-outs, delivery status, no duplicates to users, and urgent messages delivered within seconds even while bulk sends are running.

Estimate: $20M \div 86,400 \approx 230$ a second on average, about 2,300 at peak: modest, but provider rate limits and slowness will dominate.

Simplest design: the API validates the request, stores it and writes an event in the same transaction (the outbox), and returns. Workers consume from a queue, apply preferences and templates, call the providers, and record the result.

What breaks: bulk sends delay login codes, so use **separate queues and workers by priority**. Providers are slow or down, so use **timeouts, retries with backoff and a circuit breaker per provider**, with a second provider for urgent traffic. Retries could send twice, so give each notification an **idempotency key** and pass it to providers that accept one. One team's runaway job could flood everyone, so apply **per-sender rate limits**. And set a **deadline** on each message: a login code that arrives after twenty minutes is worse than none.

- ☐ Did you write down the requirements, including the non-functional ones, before designing?
- ☐ Is every major decision backed by a number: peak rate, data size or growth?
- ☐ Is the first design the simplest one that meets those numbers?
- ☐ For each component, do you know what happens when it's slow, down or wrong?
- ☐ Have you said what each choice gives up?

GO DEEPER

ESSAY [Scaling your API with rate limiters Stripe](#)

GUIDE [The System Design Primer Donne Martin](#)

DOCS [Scripting with Lua Redis](#)

BOOK [System Design Interview, volumes 1 and 2 Alex Xu · worked designs of common systems](#)

Testing and verifying

3 exercises

Distributed bugs hide in rare orderings: the message that arrives late, the node that pauses at the worst moment. An ordinary test explores one ordering. The tools in this chapter explore thousands, replay the one that fails, or prove that none can.

Why ordinary tests miss them

A distributed system's behaviour depends on things a test doesn't control: thread scheduling, network timing, clock readings and random choices. A bug that needs one particular interleaving might show up once in a million runs, in production, and never again while you watch. The first step is making the code controllable. Every example in this book took its clock, and sometimes its random number generator, as a parameter, so that a test could supply fake ones. Do the same with the network: put calls to other machines behind an interface that a test can replace with one that delays, drops, duplicates or reorders.

Test the failure paths

Most tests check that things work. Distributed systems need tests that things fail well: that a timeout is handled, a duplicate is ignored, an out-of-order message is harmless, and a crash between two steps leaves nothing half done. **Property-based testing**, with a library such as Hypothesis for Python, generates hundreds of inputs and shrinks any failure to the smallest example, which suits properties such as "merging in any order gives the same result" from chapter 4. For integration tests, Testcontainers runs real databases and brokers, and Toxiproxy sits between your service and them to add latency, cut connections or limit bandwidth on demand.

Jepsen: checking what databases promise

Kyle Kingsbury's **Jepsen** tests a real cluster from the outside. Clients perform operations while a "nemesis" partitions the network, kills and pauses nodes, and skews clocks. Every operation's start, end and result goes into a history, and a checker then searches for an order of the operations that's consistent with the database's claimed consistency model: Knossos for linearizability, Elle for transaction isolation. If no such order exists, the database broke its promise, and the history shows how. Jepsen has found violations in many well-known databases, often where the documentation promised more than the system delivered. Before trusting a database with data that matters, read its Jepsen analysis if one exists.

Deterministic simulation testing

FoundationDB's engineers built their database to run inside a simulator. In simulation, the network, the disks and the clock are fakes, many nodes run in a single thread, and every choice, from message delays to which node crashes next, comes from one random seed. So a whole cluster's run is a function of its seed. The simulator injects faults far more often than reality does and runs time faster than a real clock, so it covers years of bad luck in hours. When an invariant fails, rerunning the same seed reproduces the failure exactly, as many times as debugging needs.

TigerBeetle's simulator, the VOPR, works the same way: a seed and a commit reproduce any failure it finds. Antithesis, a company founded by FoundationDB engineers, runs ordinary software inside a deterministic hypervisor, to get the same property without rewriting it. The core idea fits in a few lines.

EXAMPLE `simulate.py` · the idea behind simulation testing

```
import random

def simulate(seed: int, dedupe: bool) -> bool:
    """One run: deliver five payments over a network that duplicates and reorders."""
    rng = random.Random(seed) # every random choice comes from the seed
    payments = [f"pay_{i}" for i in range(5)]
    in_flight = []
    for p in payments:
        in_flight.append(p)
        if rng.random() < 0.2: # 20% of messages are delivered twice
            in_flight.append(p)
    rng.shuffle(in_flight) # and in any order
    applied, seen = [], set()
    for message in in_flight:
        if dedupe and message in seen:
            continue
        seen.add(message)
        applied.append(message)
    return sorted(applied) == payments # invariant: each payment applied exactly once

failing = [s for s in range(10_000) if not simulate(s, dedupe=False)]
len(failing) # 6761: about two runs in three break the invariant
simulate(failing[0], dedupe=False) # False, and False on every rerun: it replays exactly
all(simulate(s, dedupe=True) for s in range(10_000)) # True
```

A real simulator runs the system's actual code against simulated networks and disks, not a toy model, but the properties are the same ones shown here: a search over thousands of seeds, a clear invariant, and exact replay of any failure found.

Formal methods: checking the design

Simulation tests the code you've written. TLA+, created by Leslie Lamport, checks a design before you write it. You describe the system's states and the steps it can take, and the TLC model checker explores every reachable state of a small instance, checking your invariants

in each one. Engineers at AWS have used it since 2011. On DynamoDB's replication and group-membership system it found three bugs whose shortest demonstrations took up to 35 steps, orderings no test suite would stumble on.

Here is the lost update from chapter 6 as a model: two processes each read a shared counter and then write back one more than they read.

EXAMPLE Counter.tla

```
----- MODULE Counter -----
EXTENDS Naturals

Procs == {"p1", "p2"}
VARIABLES counter, copy, pc
vars == <<counter, copy, pc>>

Init == /\ counter = 0
        /\ copy = [p \in Procs |-> 0]
        /\ pc = [p \in Procs |-> "read"]

Read(p) == /\ pc[p] = "read"
            /\ copy' = [copy EXCEPT ![p] = counter]
            /\ pc' = [pc EXCEPT ![p] = "write"]
            /\ UNCHANGED counter

Write(p) == /\ pc[p] = "write"
            /\ counter' = copy[p] + 1
            /\ pc' = [pc EXCEPT ![p] = "done"]
            /\ UNCHANGED copy

AllDone == \A p \in Procs : pc[p] = "done"
Done      == AllDone /\ UNCHANGED vars          \* finished: stutter rather than deadlock

Next == (\E p \in Procs : Read(p) \/ Write(p)) \/ Done
Spec == Init /\ [][Next]_vars

\* The invariant TLC checks in every reachable state.
NoLostUpdate == AllDone => counter = 2
=====
```

Checking `NoLostUpdate`, TLC reports a violation almost immediately and prints the trace: p1 reads 0, p2 reads 0, p1 writes 1, p2 writes 1. The model is small enough to check every ordering, and that's the point: a model of a real protocol, such as a replication scheme or a payment workflow, is usually a few hundred lines, far smaller than the code, and can be checked exhaustively before the code exists. AWS also uses lighter-weight methods on the code itself, such as property-based tests and model checking of real implementations, as it did for a storage node in S3.

Chaos engineering

Netflix's Chaos Monkey, from around 2011, randomly terminated production instances during working hours, so every team had to build services that survived losing a machine. The discipline that grew from it is careful rather than random. Define the system's steady state by a business metric, such as successful checkouts a minute. Form a hypothesis that it

holds under some fault. Inject the fault in a controlled way, with a small blast radius and an abort switch, and look for a difference. **Game days** rehearse larger scenarios, such as losing a zone, with the people and runbooks involved. AWS Fault Injection Service, Gremlin and Chaos Mesh provide the tooling.

Which tool, when

SITUATION	USE
Every service	Injected clocks and randomness; tests of timeouts, duplicates and crashes; fault injection in integration tests
Services that must stay up	Chaos experiments and game days
A protocol at the heart of the system	A TLA+ model, deterministic simulation, or both
Choosing a database	Its Jepsen analysis, and your own tests of the guarantees you rely on
Whatever still gets through	Distributed tracing and good telemetry, to debug in production

COMMON TRAPS

Only testing the happy path, where distributed systems rarely fail.

Reading the clock and the random generator directly, so tests can't control them.

Flaky tests dismissed as noise. A test that fails one run in fifty may be showing you a real race.

Chaos without a hypothesis, which breaks things without teaching anything.

Believing the documentation about a database's guarantees without evidence.

EXERCISES

EXERCISE 20.1 Make it testable

This function can't be tested without real waiting, real randomness and a real server. Change it so that a test can control all three, then write a test that it gives up after five attempts. What does the test reveal?

EXERCISE sender.py

```
import random
import time

import requests

def send_with_retry(url, payload):
    for attempt in range(5):
        try:
            return requests.post(url, json=payload, timeout=2)
        except requests.RequestException:
            time.sleep(min(30, 2 ** attempt) * random.random())
    raise RuntimeError("gave up")
```

SOLUTION

Pass the three dependencies in, with the real ones as defaults so that callers don't change.

SOLUTION sender.py

```
def send_with_retry(url, payload, post=requests.post, sleep=time.sleep,
                    rand=random.random, attempts=5):
    for attempt in range(attempts):
        try:
            return post(url, json=payload, timeout=2)
        except OSError:
            # requests' exceptions are OSError's too
            if attempt == attempts - 1:
                raise
            sleep(min(30, 2 ** attempt) * rand())

def test_gives_up_after_five_attempts():
    calls, sleeps = [], []

    def failing_post(url, **kwargs):
        calls.append(url)
        raise ConnectionError("down")

    with pytest.raises(ConnectionError):
        send_with_retry("https://example.test", {}, post=failing_post,
                       sleep=sleeps.append, rand=lambda: 1.0)
    assert len(calls) == 5
    assert sleeps == [1, 2, 4, 8]
```

Writing the test exposes a bug in the original: it slept after the final attempt too, up to 16 seconds spent waiting before giving up anyway. It also replaced the real error with a generic one, which the new version re-raises instead.

EXERCISE 20.2 Read the history

A Jepsen-style test records these operations on one register, which starts at 0. Times are in milliseconds.

- Client A writes 1, starting at 0 and completing at 10.
- Client B reads, starting at 2 and completing at 4, and gets 1.
- Client C reads, starting at 5 and completing at 7, and gets 0.

Each read overlaps the write, so each on its own could see either value. Is the history linearizable?

SOLUTION

No. B's read returned 1, so the write must take effect before B's read, which finished at 4. C's read started at 5, after B's read had completed, so it must take effect after B's and therefore after the write, and it must return 1. It returned 0.

This is the rule from chapter 6: once any read has seen the new value, every read that starts afterwards must see it too. The history would be linearizable if C had returned 1, or if B had returned 0. It's a typical anomaly of reads served by lagging replicas.

EXERCISE 20.3 Plan a chaos experiment

Checkout calls a recommendations service to show "you might also like". Plan an experiment to test whether checkout survives that service becoming slow.

SOLUTION

- **Steady state:** checkout success rate at or above 99.5%, with p99 latency under 800 ms.
- **Hypothesis:** if recommendations responds two seconds late, checkout is unaffected, because its call has a 150 ms timeout and falls back to showing nothing.

- **Method:** inject two seconds of latency into calls to recommendations, first in staging, then in production for 5% of traffic in one cell, during working hours with the team watching.
- **Abort:** stop automatically if checkout success drops below 99% or p99 exceeds one second.
- **Afterwards:** record what happened. If the hypothesis failed, perhaps because the timeout was never set or the fallback threw an exception, fix it, then automate the experiment so it keeps running.

CHECKLIST

- ☐ Can your code run with an injected clock, random source and network?
- ☐ Do tests cover timeouts, duplicates, reordering and crashes between steps?
- ☐ Have you injected a real fault into a real environment this quarter?
- ☐ Is the protocol at the heart of your system modelled or simulated, not just unit-tested?
- ☐ When a rare failure happens, can you reproduce it?

GO DEEPER

REPORTS Jepsen analyses [Kyle Kingsbury](#) · read two or three

DOCS Deterministic simulation testing [Antithesis](#)

DOCS The VOPR [TigerBeetle](#)

PAPER How Amazon Web Services Uses Formal Methods [Newcombe et al., 2015](#)

COURSE Learn TLA+ [Hillel Wayne](#)

ESSAY Principles of Chaos Engineering

One per chapter, each linked back to it. Read them before a design review, and against every design document that sends a message between two machines.

-
- 01 Distribute only for a reason you can name, and count what every extra hop costs in latency and availability.
-
- 02 Put a timeout on every remote call, and design for not knowing whether it happened.
-
- 03 Never order events on different machines by their wall clocks; use logical clocks, or bound the uncertainty.
-
- 04 Decide how fresh each read must be, and send it to a copy that can promise that.
-
- 05 Choose the partition key from your queries, and plan for the hot key before it arrives.
-
- 06 Name the consistency each operation needs, and enforce invariants with constraints, not with a check followed by a write.
-
- 07 Ask what a system does during a partition, and what it costs in latency without one; "CP" and "AP" answer neither.
-
- 08 Build on proven consensus rather than your own, and remember it needs a majority to make progress.
-
- 09 Pair every lease with a fencing token, checked where the damage would be done.
-
- 10 Keep each transaction inside one database; across services, use the outbox, idempotent consumers and compensating sagas.
-
- 11 Assume every message arrives at least once, and make every consumer idempotent.
-
- 12 Treat events as permanent contracts: version them from the start, and adopt event sourcing only where the history is worth its lifelong cost.

-
- 13 Try a scheduled query before a stream processor; when you need one, window by event time and decide how late is too late.
-
- 14 Propagate deadlines, retry at one layer within a budget, and shed load before it sheds you.
-
- 15 Balance on real load, drain before stopping, and limit the blast radius with zones, cells and shuffle sharding.
-
- 16 Size the database for an empty cache, delete rather than set on writes, and protect hot keys from stampedes.
-
- 17 Start with a modular monolith, and split out a service only for a benefit you can name.
-
- 18 Draw boundaries where the language changes, and keep each invariant inside one small aggregate.
-
- 19 Clarify, estimate, build the simplest design that meets the numbers, then say what it gives up.
-
- 20 Make time, randomness and the network injectable, then search for the orderings that break you.
-

READING LIST

The books behind this volume, and the chapters each one deepens. If you read only one, make it *Designing Data-Intensive Applications*, and read it slowly, with this volume's exercises alongside.

BOOK	AUTHOR	DEEPENS
Designing Data-Intensive Applications, 2nd edition	Martin Kleppmann & Chris Riccomini	Ch. 1–13
Understanding Distributed Systems	Roberto Vitillo	Ch. 1–3, 14–16
Distributed Systems	Maarten van Steen & Andrew Tanenbaum	Ch. 2–9
Database Internals	Alex Petrov	Ch. 4–8
Streaming Systems	Tyler Akidau, Slava Chernyak & Reuven Lax	Ch. 13
Release It!, 2nd edition	Michael Nygard	Ch. 14–15
Site Reliability Engineering	Betsy Beyer et al., Google	Ch. 14–16
Building Microservices, 2nd edition	Sam Newman	Ch. 17
Team Topologies	Matthew Skelton & Manuel Pais	Ch. 17
Learning Domain-Driven Design	Vlad Khononov	Ch. 18
System Design Interview, volumes 1 and 2	Alex Xu	Ch. 19
Practical TLA+	Hillel Wayne	Ch. 20

Compiled September 2026 as the companion to stage 11 of The Engineer's Syllabus, following *The Engineer's Craft* and *The Engineer's Shipyard*. Research drew on the second edition of DDIA, Martin Kleppmann's Cambridge lecture notes, Lamport's 1978 paper on clocks, the Raft paper, CockroachDB on clocks, Kleppmann on distributed locking, PACELC, Jepsen's consistency models, GitHub's October 2018 incident analysis, reporting on the October 2025 AWS outage, the Kafka 4.0 announcement, The Tail at Scale, the metastable failures paper, the Amazon Builders' Library, the Memcache paper, the XFetch paper, a summary of the Prime Video case, whose original post is no longer online, TigerBeetle's VOPR and AWS's report on formal methods.

The self-contained examples, including the availability arithmetic, Lamport and vector clocks, the G-Counter, quorum overlap, consistent hashing, the Raft vote rule, fencing, the saga runner, the idempotent consumer, the event-sourced fold and upcaster, windows with a watermark, the circuit breaker, two-choice balancing, early cache refresh, the boundary checker, the order aggregate, the estimates and the simulation, were run before publication to confirm they behave as the text says. The Lua token bucket was run against a real Redis server. The TLA+ model's counterexample was confirmed by an exhaustive search of the same states in Python, not by TLC itself. Examples that depend on a framework or a running cluster are illustrative. Re-checked in October 2026 with Python 3.14.7 and Redis 8.8: every self-contained example ran, and a test of each claim the text makes about it, from the share of keys that move when a node joins the ring to XFetch's refresh probabilities, passed.