

PRINT-FLOW-360 · RESEARCH LIBRARY

System Design

21 research documents · each with plain-language key takeaways

Compiled 2026-06-26

Table of Contents

1. System Design — Complete Learning Curriculum
2. Foundations & Back-of-the-Envelope Estimation
3. Networking & Protocols
4. API Design & Service Communication
5. Database Internals: Storage Engines, Indexes, Transactions
6. 05 — Data Modeling: SQL vs NoSQL & Polyglot Persistence
7. 06 — Caching Deep Dive
8. 07 · Load Balancing, Scaling & Stateless Design
9. 08 — Replication & Partitioning (Sharding)
10. 09 — CAP, PACELC & Consistency Models
11. 10 — Consensus & Distributed Coordination
12. Module 11 — Distributed Transactions, Sagas & Idempotency
13. 12 · Messaging, Queues & Event-Driven Architecture
14. 13 — Event Sourcing & CQRS
15. 14 · Stream Processing & Real-Time Systems
16. 15 — Probabilistic Data Structures & Algorithms at Scale
17. 16 — Rate Limiting, Resiliency & Fault Tolerance
18. 17 · Observability, SRE & Operating Systems at Scale
19. Module 18 — Architecture Patterns: Monolith → Microservices → Cells
20. Module 19 — Specialized Systems: Search, Geospatial, Time-Series & Analytics
21. 20 — Case Studies & the System Design Interview Framework

System Design — Complete Learning Curriculum

Source: [readme/research/system-design/00-INDEX-learning-roadmap.md](#)

IN SHORT — KEY TAKEAWAYS

This is a 20-lesson study plan for software developers who want to get very good at designing large, reliable computer systems — the kind that handle millions of users without breaking or losing data. It goes step by step from basic ideas all the way to the hardest problems that companies like Amazon and Google have to solve every day. If your developer is working through this, it will sharpen how they think when the platform needs to grow or when something goes wrong at 3 a.m.

The main parts explained simply:

- **Tier 1 — The Basics (Modules 1, 2, 3, 6)** — Start here. Learn how to estimate how big a system needs to be before you build it, how the internet actually moves data between computers, how apps talk to each other, and how a “cache” works — storing copies of popular data nearby so the system doesn’t keep fetching the same thing over and over.
- **Tier 2 — Databases and Data (Modules 4, 5, 7, 8)** — Learn how a database actually finds and stores your data under the hood, when to use different types of storage (not all data belongs in the same kind of database), and how to spread work across many servers so the system stays fast as it grows.
- **Tier 3 — Distributed Systems Core (Modules 9–15)** — The hardest part. When your data lives on many computers at once, keeping it correct and consistent is very tricky. These lessons explain the honest trade-offs: a system usually cannot be both always available AND always perfectly up-to-date at the same time. You have to pick.
- **Tier 4 — Running Systems Reliably (Modules 16–19)** — How to stop a system from being overwhelmed by too many requests, how to notice problems before customers do, and how to decide when to break a big app into smaller separate pieces — or when that would make things worse.
- **Tier 5 — Interview Prep (Module 20)** — A 7-step thinking method for designing any system, plus 10 worked examples such as how to build a chat app, a URL shortener, or a payment system.
- **12 Core Mental Models** — Big ideas that come up in every topic, such as “every choice is a trade-off,” “failure is normal at scale, not rare — design for it,” and “don’t over-engineer: a single well-run database handles more than most teams ever need.”

What to do with this:

Read the modules in order, starting with Module 01. Do the practice questions at the end of each section before moving on — if you cannot explain the idea in plain words, you have not learned it yet. Budget roughly one focused evening per module for the early tiers, and two evenings for the harder distributed-systems modules.

This is a **self-study course**, not a stack of cheat-sheets. The goal is to take an experienced web developer — someone comfortable shipping a CRUD app against one database — all the way to **staff-level reasoning about distributed systems**: why linearizability is expensive, why every cross-service “transaction” is a lie you have to engineer around, why the speed of light shows up in your p99, and how to make defensible trade-offs under uncertainty.

Each of the 20 modules is **teaching material**: it builds intuition first (plain analogies), then precise mechanics, then the failure modes that show up in a 3 a.m. incident or a hard interview question. Modules include back-of-the-envelope math, ASCII diagrams, comparison tables, war stories, **Test-Yourself questions**, and pointers to the canonical papers. They are meant to be *read in order and worked through*, not skimmed.

How to use this curriculum

- **Read in tier order.** Later modules assume the vocabulary of earlier ones. The Distributed Systems Core (Tier 3) is the hard, load-bearing middle — don’t skip to the interview module before it.
- **Do every Test-Yourself question before moving on.** If you can’t answer in your own words, re-read — recognition is not understanding.
- **Build small versions.** A toy consistent-hash ring (300 lines), a token-bucket limiter in Redis, a tiny event-sourced ledger that rebuilds state by folding events. You learn replication lag by *causing* it, not by reading about it.
- **Read the cited papers.** The summaries here compress 30-page papers into paragraphs. Read Dynamo, Raft, and the Spanner paper in full at least once — they’re more readable than their reputation.
- **Re-read after a real incident.** The modules land much harder once you’ve watched a retry storm take down your own service.

Budget roughly **one focused evening per module** for the fundamentals, **two** for the Tier 3 distributed-systems modules. The whole curriculum is a multi-week effort — that’s expected.

Suggested learning path

The 20 modules group into five tiers. Within a tier, modules listed as “parallel-OK” can be read in any order; across tiers, finish the earlier tier first.

Tier 1 — Fundamentals (*everyone, first*)

01 → 02 → 03 → 06 Estimation, networking, API design, and caching. This is the shared vocabulary and the mental math (latency numbers, QPS, bytes) the whole rest of the course leans on. Module 01 is the absolute starting point.

Tier 2 — Data & State

04 → 05 → 07 → 08 How a single database actually works (storage engines, indexes, transactions), how to model data across SQL/NoSQL, then how to scale the serving tier (load balancing, stateless design) and the data tier (replication, partitioning). **08 is the bridge into distributed systems.**

Tier 3 — Distributed Systems Core (*the advanced heart*)

09 → 10 → 11 → 12 → 13 → 14 → 15 The conceptual center of gravity. CAP/PACELC and consistency models (09) and consensus (10) are prerequisites for almost everything after. Distributed transactions/sagas (11), messaging (12), event sourcing + CQRS (13), and stream processing (14) build directly on them. Probabilistic structures (15) is more standalone and can be read in parallel with the others once you have 09.

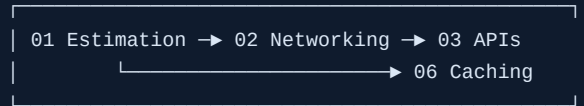
Tier 4 — Architecture & Operating at Scale

16 → 17 → 18 → 19 Resiliency/rate-limiting (16) and observability/SRE (17) — parallel-OK with each other. Then architecture patterns (18, monolith → microservices → cells) and specialized stores (19, search/geo/time-series/OLAP), which assume Tiers 2 and 3.

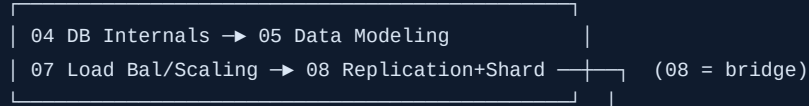
Tier 5 — Applied / Interview

20 The 7-step interview framework + ten worked case studies. Read it **last** — it's where every prior concept gets exercised under a time budget. (You *can* skim it first to see where the course is going, then return to it for real.)

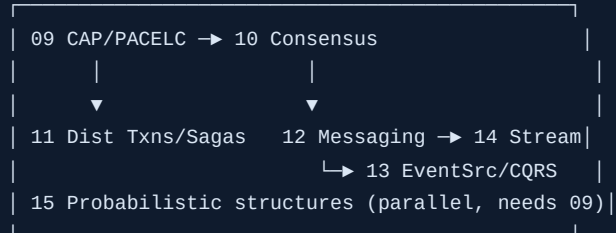
TIER 1 Fundamentals



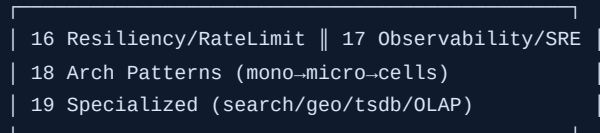
TIER 2 Data & State



TIER 3 Distributed Systems Core (the hard middle)



TIER 4 Architecture & Operating



TIER 5 Applied



Table of contents

#	Module	What you'll learn	Difficulty
01	Foundations & Back-of-the-Envelope Estimation	Latency/throughput/storage numbers every engineer should know; sizing a system from user counts; the napkin math that anchors every design	★☆☆☆☆
02	Networking & Protocols	TCP/UDP, TLS, HTTP/1.1 → 2 → 3, DNS, the OSI mental model, and where the speed of light bites your latency budget	★★☆☆☆
03	API Design & Service Communication	REST vs gRPC vs GraphQL, versioning, pagination, idempotency, sync vs async communication patterns	★★☆☆☆
04	Database Internals: Storage Engines, Indexes, Transactions	B-trees vs LSM-trees, the WAL, MVCC, isolation levels, how an index actually speeds a query	★★★☆☆
05	Data Modeling: SQL vs NoSQL & Polyglot Persistence	Normalization vs denormalization, document/wide-column/KV/graph models, picking the right store per access pattern	★★★☆☆
06	Caching Deep Dive	Cache patterns (aside/through/behind), eviction, invalidation, TTLs, stampedes, and the layers from CPU to CDN	★★★★☆
07	Load Balancing, Scaling & Stateless Design	L4 vs L7, balancing algorithms, health checks, externalized state, autoscaling, read-replicas vs sharding, load shedding	★★★★☆
08	Replication & Partitioning (Sharding)	Single/multi/leaderless replication, sync vs async durability, lag anomalies, consistent hashing, rebalancing, routing	★★★★☆
09	CAP, PACELC & Consistency Models	CAP stated correctly, PACELC, the linearizable → eventual spectrum, quorum math, isolation vs consistency	★★★★☆
10	Consensus & Distributed Coordination	FLP impossibility, Paxos/Raft/ZAB, ZooKeeper/etcd, distributed locks + fencing tokens, leases, clocks	★★★★★
11	Distributed Transactions, Sagas & Idempotency	Why cross-service ACID fails, 2PC/3PC blocking, sagas + compensations, the outbox+CDC fix, delivery semantics	★★★★★
12	Messaging, Queues & Event-Driven Architecture	Queues vs logs, Kafka internals, ordering/delivery guarantees, backpressure, retries, DLQs, exactly-once limits	★★★★☆
13	Event Sourcing & CQRS	State as an append-only event log, folding + snapshots, read projections, versioning/upcasting, when it's overkill	★★★★☆
14	Stream Processing & Real-Time Systems	Batch vs stream, Lambda vs Kappa, event vs processing time, windowing, watermarks, checkpointing, exactly-once	★★★★★
15	Probabilistic Data Structures & Algorithms at Scale	Bloom/Cuckoo filters, HyperLogLog, Count-Min, t-digest, Merkle trees, consistent hashing, geo indexes, skip lists	★★★★☆

#	Module	What you'll learn	Difficulty
16	Rate Limiting, Resiliency & Fault Tolerance	The five rate-limiters, distributed Redis variants, timeouts, jittered retries, circuit breakers, bulkheads, chaos	★★★★☆
17	Observability, SRE & Operating Systems at Scale	Metrics/logs/traces, RED/USE, tracing + sampling, SLI/SLO/error budgets, burn-rate alerts, canary/blue-green deploys	★★★★☆
18	Architecture Patterns: Monolith → Microservices → Cells	When (not) to split, DDD bounded contexts, strangler-fig, gateway/mesh, DB-per-service, distributed-monolith trap, cells	★★★★☆
19	Specialized Systems: Search, Geo, Time-Series & Analytics	Inverted indexes + BM25, geohash/quadtrees/S2, TSDB compression, columnar OLAP, polyglot persistence via CDC	★★★★☆
20	Case Studies & the System Design Interview Framework	A 7-step design loop + ten worked designs (URL shortener, feed, chat, rideshare, payments...) and their load-bearing decisions	★★★★☆

Mental models / first principles

These big ideas recur across nearly every module. If you internalize only ten things, make it these:

1. **There is no free lunch — every choice is a trade-off.** Latency vs consistency, space vs accuracy, throughput vs durability. “Best” is meaningless without a workload.
2. **You can’t beat the speed of light.** Cross-region round trips cost tens of milliseconds no matter how fast your code is. Geography sets the floor on your p99.
3. **Failure is the normal case, not the exception.** At scale, *something* is always broken. Design for partial failure, not for the happy path.
4. **State is the enemy of scaling.** Stateless services scale trivially; the hard problems all live where state must be shared, replicated, or agreed upon.
5. **Consistency, availability, and partitions force a choice** (CAP/PACELC) — and even without a partition you still trade latency against consistency on every request.
6. **Coordination is expensive; avoid it when you can.** Consensus, distributed locks, and linearizable reads cost round trips. The cheapest distributed system is one that needs no agreement.
7. **The dual-write problem is everywhere.** You cannot atomically update two systems. The answer is almost always *one* atomic write plus the outbox/log/CDC pattern.
8. **Idempotency is the antidote to “exactly-once.”** True exactly-once delivery doesn’t exist across a network; at-least-once + idempotent handlers is how production gets it.
9. **Indexes (and data layouts) are shaped like queries.** The right store/index is the one whose structure mirrors your read pattern — that’s why polyglot persistence exists.

10. **Bound everything: timeouts, retries, queues, blast radius.** Unbounded anything becomes a cascading or metastable failure. Backpressure and isolation keep failures local.
 11. **Eventual consistency is a UX problem as much as a data problem.** Read-your-writes and monotonic reads are often what users actually need — not full linearizability.
 12. **Don't over-engineer.** A modular monolith on one Postgres handles more than most teams ever reach. Distribute only when a concrete forcing function demands it.
-

Canonical resources

The course stands on these. Read the book; read the papers at least once each.

Books

- **Designing Data-Intensive Applications** (Kleppmann) — *the* companion to Tiers 2–3. If you read one book, read DDIA.
- **Google SRE Book** (+ *The SRE Workbook*) — backs Module 17; SLOs, error budgets, on-call.
- **The System Design Primer** (GitHub, donnemartin) — breadth review + interview drilling.

Foundational papers

- **Dynamo** (Amazon, 2007) — leaderless replication, quorums, consistent hashing, vector clocks (Modules 08, 09, 15).
- **Spanner / TrueTime** (Google, 2012) — external consistency via bounded-uncertainty clocks (Modules 09, 10).
- **Raft — In Search of an Understandable Consensus Algorithm** (Ongaro & Ousterhout, 2014) (Module 10).
- **MapReduce** (Google, 2004) and the **Kafka** paper (LinkedIn, 2011) — batch + the log (Modules 12, 14).
- **Bigtable** (Google, 2006) — wide-column/LSM design (Modules 04, 05).
- *Also worth it:* the **Paxos Made Simple** note, the **CAP** retrospective (Brewer, 2012), the **FLP impossibility** result, and the **HyperLogLog** paper.

Courses

- **MIT 6.824 Distributed Systems** — lectures + labs (build Raft yourself). The single best practical complement to Tier 3.

Self-assessment milestones

You've internalized the curriculum when you can honestly check these off:

- [] I can **size a system for 10M users** — estimate QPS, storage, and bandwidth on a napkin and know the key latency numbers from memory. (01)

- [] I can explain **why linearizability is expensive** and when eventual/causal/session consistency is the right call instead. (09)
- [] I can **design a rate limiter that works across N servers** without a single hot counter, and reason about its accuracy/memory trade-off. (16)
- [] I can describe **how a write becomes durable** — WAL, replication, quorum — and what data you lose in each failover mode. (04, 08)
- [] I can explain **why you can't have a distributed transaction** across two services and design a saga with compensations + idempotency keys instead. (11)
- [] I can choose **the right datastore for an access pattern** and justify polyglot persistence wired via CDC. (05, 19)
- [] I can reason about a **retry storm / metastable failure** and name the mechanisms (jitter, budgets, circuit breakers, load shedding) that prevent it. (16)
- [] I can **operate** a service: define an SLO, set a burn-rate alert, and run a canary deploy with a safe schema migration. (17)
- [] I can decide **when NOT to use microservices** and recognize a distributed monolith from its symptoms. (18)
- [] I can walk into a **system design interview**, drive the 7-step framework, and defend the two or three load-bearing decisions of any design. (20)

Welcome in. Start with **Module 01** and work forward.

Foundations & Back-of-the-Envelope Estimation

Source: [readme/research/system-design/01-foundations-and-estimation.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains how engineers estimate whether a system can handle a given number of users — before they build anything. It gives you the key numbers to know (how fast things are, how big data gets, how much traffic a server can take) and a step-by-step method for checking if your design will hold up. Understanding this helps you ask the right questions and make better decisions when building or scaling software.

The main parts explained simply:

- **How fast things are (latency)** — Reading from computer memory is nearly instant. Reading from a hard disk is 20 million times slower. Sending data across the world takes 150 milliseconds, and no amount of code can make it faster — physics sets the limit. Knowing these gaps tells you where to focus: keep hot data in fast memory (called a cache) to avoid slow disk or network reads.
- **How big data gets (data sizes)** — Simple rules help you go from “one user does X” to “1 million users means Y gigabytes.” A photo is roughly 1–5 MB; a year of photos from many users adds up to terabytes fast. Images almost always dwarf text in storage cost.
- **How much traffic a server handles (QPS)** — QPS means “requests per second.” You work it out from how many users you have and what they do per day. Always plan for the busiest moment (peak), not the daily average — a lunchtime spike or a product launch can be 3–10 times the normal rate.
- **The four things that run out (bottlenecks)** — Every server has four limited resources: processing power (CPU), working memory (RAM), disk space and speed, and network speed. Find which one fills up first and fix only that — adding more CPU to a server that’s running out of disk space does nothing.
- **Little’s Law** — A simple equation: if 2,000 requests arrive per second and each takes 50 ms to finish, then at any moment 100 requests are in progress. This tells you how many database connections or worker threads you actually need — too few and things queue up, slow down, and crash.
- **Why averages lie (percentiles)** — An “average response time of 40 ms” can hide the fact that 1 in 100 users waits 4 seconds. Your best, most active customers hit those slow cases most often. Good engineering sets targets using percentiles (e.g. “99% of requests finish under 200 ms”), not averages.
- **The fan-out trap** — When one request splits into calls to many services at once and waits for all of them, even a tiny slowness in any one service shows up most of the time. Twitter solved this by pre-building each user’s feed when a tweet is posted, so reading it is instant instead of gathering from hundreds of sources on every page load.
- **Capacity planning method** — A seven-step checklist: state your assumptions, calculate requests per second (average and peak), calculate storage, calculate bandwidth, find the bottleneck, count servers needed, then sanity-check against known limits.

What to do with this: When sizing any new feature, always compute peak traffic (not average) and check storage after multiplying by replication (usually 3×) and index overhead (2×). Set performance targets in percentiles (p99), not averages — averages always make things look better than they are.

What you'll learn: How to reason quantitatively about systems before you build them — the latency and data-size numbers that anchor every design decision, how to estimate QPS / storage / bandwidth from a user count, and the mental models (Little's Law, percentiles, tail amplification, the four golden resources) that separate a guess from an engineering estimate. By the end you can size a system for tens of millions of users on a whiteboard in ten minutes and defend every number.

Prerequisites: none — start here. This is module 01 of the curriculum.

1. Why estimation is the first skill

A system design interview, or a real capacity-planning meeting, is not won by naming technologies. It's won by being able to say: *"This needs ~12k writes/sec, ~250 GB/day of new data, and a single Postgres box tops out around 5–10k writes/sec — so we need sharding or a write buffer."* That sentence is three estimates and one threshold. Everything downstream (caching, sharding, queues, replication) is justified by numbers like these.

The goal of back-of-the-envelope (BOTE) math is **order-of-magnitude correctness**, not precision. If the real answer is 8,000 QPS and you say 10,000, you're right. If you say 100 or 1,000,000, you'll design the wrong system. Estimation is about staying within one power of ten.

There are exactly three inputs you ever need to memorize:

1. **Latency numbers** (how long operations take).
2. **Powers of two** (how data sizes relate).
3. **A few system thresholds** (what one box can do).

Everything else is multiplication.

2. Latency numbers every engineer should know

These are the canonical figures popularized by Jeff Dean ("Latency Numbers Every Programmer Should Know"). Memorize the *relative* magnitudes; the absolute values drift with hardware but the ratios are stable and that's what matters.

Operation	Latency	Human-scale analogy (×1 billion)
L1 cache reference	0.5 ns	0.5 s — a heartbeat
Branch mispredict	5 ns	5 s
L2 cache reference	7 ns	7 s
Mutex lock/unlock	25 ns	25 s
Main memory (RAM) reference	100 ns	~2 min
Compress 1 KB (Snappy)	3,000 ns (3 μs)	~1 hr
Send 1 KB over 1 Gbps network	10,000 ns (10 μs)	~3 hr
Read 1 MB sequentially from RAM	250,000 ns (250 μs)	~3 days
SSD random read	100,000–150,000 ns (~100 μs)	~1.5 days
Round trip within same datacenter	500,000 ns (0.5 ms)	~6 days
Read 1 MB sequentially from SSD	~1,000,000 ns (1 ms)	~12 days
Rotational disk seek	10,000,000 ns (10 ms)	~4 months
Read 1 MB sequentially from spinning disk	~20,000,000 ns (20 ms)	~8 months
Send a packet CA → Netherlands → CA	150,000,000 ns (150 ms)	~5 years

The “×1 billion” column turns nanoseconds into seconds so your intuition can grip them. It reveals the brutal truth of the **memory hierarchy**:

```

CPU register    ~0 ns    | instant
L1 cache        0.5 ns   | 1×
L2 cache        7 ns     | 14×
RAM             100 ns   | 200×
SSD random      100 μs   | 200,000×
DC round trip   500 μs   | 1,000,000×
Disk seek       10 ms    | 20,000,000×
Cross-continent 150 ms  | 300,000,000×
```

The five rules that fall out of this table

- RAM is ~100× faster than SSD, SSD is ~100× faster than disk seek.** Two orders of magnitude each. This is *the* justification for caching (module `06-caching-and-cdn.md`): keep hot data in RAM and you avoid a 1000× penalty.
- A datacenter round trip (0.5 ms) costs ~5,000 main-memory accesses.** Network calls are expensive. This is why the N+1 query problem (see your own `CLAUDE.md`) destroys performance — each round trip is thousands of in-memory operations.
- Sequential beats random by ~100×** on both SSD and disk. Databases, log-structured storage (Kafka, LSM-trees), and batch jobs are all designed to turn random I/O into sequential I/O.

4. **Cross-continent is ~150 ms, fixed by physics.** Light in fiber travels ~200,000 km/s; a 12,000 km path is ~60 ms one way before any processing. No amount of money makes a single round trip to another continent faster than ~tens of ms. The only fix is *moving the data closer* (CDN, regional replicas).
5. **Compression is cheap relative to network/disk.** 3 μ s to compress 1 KB vs 10 μ s to send it — compress before you transmit when bandwidth is the bottleneck.

3. Powers of two and data-size math

Computers count in powers of two, but humans (and BOTE math) count in powers of ten. The trick is to remember a few anchor points and round.

Power	Exact	Approx (decimal)	Name	Mnemonic
2^{10}	1,024	~1 thousand	Kilo (KB)	“a thousand”
2^{20}	1,048,576	~1 million	Mega (MB)	“a million”
2^{30}	~1.07 billion	~1 billion	Giga (GB)	“a billion”
2^{32}	~4.29 billion	~4 billion	—	max unsigned 32-bit
2^{40}	~1.10 trillion	~1 trillion	Tera (TB)	“a trillion”
2^{50}	~1.13 quadrillion	~1 quadrillion	Peta (PB)	—

The working shortcut: treat $2^{10} \approx 10^3$. Then KB $\rightarrow$ MB $\rightarrow$ GB $\rightarrow$ TB $\rightarrow$ PB are each $\times 1000$. To go from a per-item byte size to a total, just count zeros.

Two more numbers worth burning in:

- **$2^{32} \approx 4.3$ billion** — the IPv4 address space and the moment a 32-bit auto-increment ID overflows. (Real war story: many systems have died at 2.1 billion when a *signed* 32-bit ID hit 2^{31} .)
- **Seconds in useful windows:** 1 day $\approx$ **86,400 s** (round to **$\sim 10^5$**), 1 month $\approx$ **2.6 million s** ($\sim 2.5 \times 10^6$), 1 year $\approx$ **31.5 million s** ($\sim 3 \times 10^7$). These convert “per day” into “per second” instantly.

Per-day $\rightarrow$ per-second: divide by ~100,000. 1 billion events/day $\div 10^5 \approx$ **$\sim 11,500$ /sec average**. Memorize “1 billion/day $\approx$ 12k/sec.”

Common per-object byte sizes for sizing:

- An ASCII char $\approx$ 1 byte; a UTF-8 emoji $\approx$ 4 bytes.
- A UUID $\approx$ 16 bytes binary, 36 bytes as text.
- A tweet/short post $\approx$ 200–300 bytes of text.
- A typical metadata row (a few columns) $\approx$ 100–1,000 bytes.
- A JPEG thumbnail $\approx$ 10–50 KB; a full photo $\approx$ 1–5 MB; a minute of 1080p video $\approx$ ~50–100 MB.

4. QPS, throughput, and read/write ratios

QPS (queries per second) is the heartbeat of capacity planning. The standard derivation:

Daily Active Users (DAU)
× actions per user per day = total daily actions
÷ 86,400 (seconds/day ~10^5) = AVERAGE QPS
× peak factor (2× to 10×) = PEAK QPS ← design to THIS

Always design for peak, not average. Traffic is bursty: lunch-hour spikes, time-zone overlap, Black Friday, a viral post. A common rule of thumb is **peak ≈ 2–3× average** for steady consumer apps, and up to **10×** for spiky/event-driven ones. Some teams instead use the “80/20” framing: 80% of traffic arrives in 20% of the day → peak ≈ (0.8 × daily) ÷ (0.2 × 86,400) ≈ **4× average**.

Read/write ratio determines your entire architecture. Measure or assume it:

System type	Read:Write	Architectural implication
Social feed (Twitter, IG)	100:1 to 1000:1	Heavy caching + read replicas; precompute feeds
E-commerce catalog (your Print-Flow store)	~10:1 to 100:1	CDN + cache product pages; DB handles writes fine
Analytics ingestion	1:100 (write-heavy)	Append-only log, batch, columnar store
Chat / messaging	~1:1	Fan-out on write vs read is a real tradeoff
Ledger / banking	balanced, durability-critical	Strong consistency over raw throughput

A read-heavy system (most consumer apps) is *cacheable* — the primary scaling lever is caching and replication. A write-heavy system can’t be cached away and must scale the write path itself (sharding, batching, append-only logs). Knowing the ratio in the first minute tells you which lever to pull.

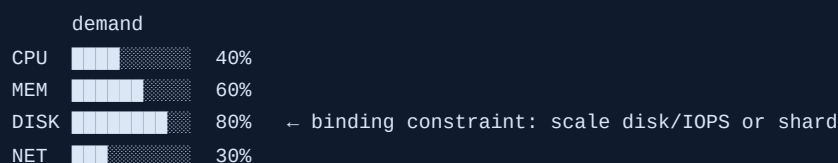
5. The four golden resources — which bottlenecks first?

Every machine has exactly four finite resources. A well-run capacity exercise asks: *which one runs out first?*

Resource	Typical limit (one commodity server)	Saturates first when...	Symptom
CPU	8–64 cores	compute-heavy: encryption, compression, serialization, image/video processing	high load average, requests queue
Memory	32–512 GB	large caches, in-memory datasets, per-connection buffers, JVM heaps	OOM kills, GC thrash, swap
Disk	space (TB) + IOPS (SSD ~10k–500k) + throughput (GB/s)	databases, logging, media storage	I/O wait, slow queries
Network	1–25 Gbps NIC	media serving, replication, large payloads	bandwidth cap, dropped packets

The discipline: estimate demand on all four, then find the binding constraint. A video service is **network/disk-bound** (serving bytes) — CPU is idle. A password-hashing or ML-inference service is **CPU-bound**. An in-memory cache (Redis) is **memory-bound** and often single-threaded so effectively CPU-bound per core. A logging pipeline is **disk-IOPS-bound**.

Designing the wrong dimension is the classic mistake: adding CPU to a network-bound box does nothing. Identify the bottleneck *first*, then scale only that.



6. Little's Law — the one equation to memorize

$$L = \lambda \times W \text{ (items in system) = (arrival rate) } \times \text{ (average time in system)}$$

It holds for *any* stable system regardless of distribution — queues, thread pools, connection pools, the whole service. It's the bridge between throughput and latency.

Worked example — sizing a thread pool / connection pool:

Your API handles $\lambda = 2,000$ requests/sec. Each request spends an average $W = 50 \text{ ms} = 0.05 \text{ s}$ in the system (mostly waiting on the database).

$$L = \lambda \times W = 2,000 \times 0.05 = 100 \text{ concurrent requests in flight}$$

So at any instant ~100 requests are being processed. If each needs a DB connection for its whole duration, you need a **pool of ~100 connections** (plus headroom). Provision 30 connections and 70 requests queue → latency climbs → timeouts → cascade.

Run it backwards to find max throughput from a fixed resource: if you can only afford a DB pool of 50 connections and W stays at 50 ms, then $\lambda_{\text{max}} = L / W = 50 / 0.05 = \mathbf{1,000 \text{ req/s}}$. Beyond that you *must* either add connections, cut W (faster queries, caching), or shed load.

This is why a small latency regression is so dangerous: if W doubles to 100 ms at the same arrival rate, L doubles to 200 — you've silently doubled your concurrency requirement and may exhaust the pool.

Latency and concurrency are coupled. (Connection pooling and backpressure get full treatment in [08-databases-storage-and-replication.md](#) and [10-message-queues-and-async.md](#).)

7. Percentiles — why averages lie

The single most important reliability concept in this module: **never trust the average (mean) latency**. Averages hide the pain.

Consider 10 requests with latencies (ms): `10, 10, 10, 10, 10, 10, 10, 10, 10, 1000`.

- **Mean = 109 ms** — looks acceptable-ish.
- **p50 (median) = 10 ms** — “typical” user is fine.
- **p90 = 10 ms, p100 (max) = 1000 ms** — one user waited a full second.

The mean is dragged around by outliers and tells you about *no actual user*. Percentiles tell you about real experiences:

Percentile	Meaning	Who cares
p50 (median)	half of requests are faster	“typical” experience
p95	1 in 20 requests is slower	noticeable to power users
p99	1 in 100 slower	SLA target for most serious services
p999	1 in 1,000 slower	Amazon/Google-tier; your biggest customers hit it constantly

Why the tail matters more than it seems: your *best* customers — the ones who make the most requests — are statistically the most likely to hit the slow tail. Amazon famously found that **100 ms of extra latency cost ~1% in sales**, and optimizes p999. A user who loads 100 pages will, on average, hit the p99 latency on one of them.

Operational rules:

- **SLOs are stated in percentiles**, e.g. “p99 < 200 ms,” never “average < 200 ms.”
 - **You cannot average percentiles across servers**. p99 of two boxes is not the average of their p99s. Aggregate raw histograms (HDR histograms, t-digest) instead.
 - **Watch the gap** between p50 and p99. A small gap = consistent system. A 10× gap (p50=10 ms, p99=100 ms) signals queueing, GC pauses, lock contention, or noisy neighbors.
-

8. Tail-latency amplification in fan-out

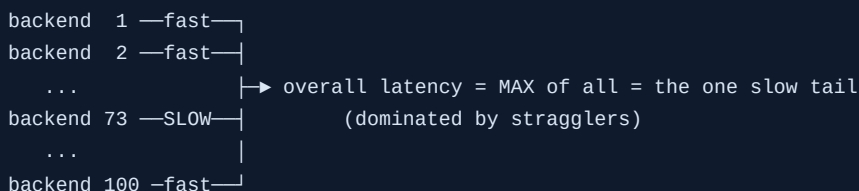
This is the advanced trap that surprises senior engineers. When a request **fans out** to many backends in parallel and must wait for *all* of them, the slow tail of each backend becomes the *common case* of the overall request.

If a single backend has a p99 of 100 ms (1% chance any one call is slow), and your request fans out to **100 backends in parallel and waits for the last one**, the probability that *at least one* is in its slow tail is:

$$P(\text{at least one slow}) = 1 - (0.99)^{100} \approx 1 - 0.366 = 0.634$$

~63% of your user-facing requests now experience the slow path. The system's p99 backend behavior has become the overall p50. This is *tail-latency amplification*, documented in Dean & Barroso's paper "*The Tail at Scale*."

Fan-out request waiting for ALL of 100 backends:



Mitigations used in real systems (Google, etc.):

- **Hedged / backup requests:** send a duplicate to a second replica after a short delay (e.g. p95), take whichever returns first. Cuts the tail dramatically for ~5% extra load.
- **Reduce fan-out width** or fan out to fewer, larger shards.
- **Tied requests / request cancellation** so the loser is cancelled.
- **Make every backend's tail tight** — the only durable fix is reducing per-backend p99.

This is why microservice meshes with deep call graphs can have *terrible* p99 even when every individual service “looks fast” on its own dashboard.

9. A repeatable capacity-planning method

A checklist you can run every time, in order:

1. **State assumptions out loud.** DAU, actions/user/day, peak factor, read:write ratio, average object size, retention period. Write them down — they're the foundation everything rests on, and a reviewer can challenge an assumption but not your arithmetic.
2. **Compute QPS** (average, then peak). Split into read QPS and write QPS using the ratio.
3. **Compute storage/day and storage/year** = (writes/day) × (bytes/object), then × retention. Add replication factor (×3 is standard) and overhead (indexes, metadata: ~1.5–2×).

4. **Compute bandwidth** = QPS × payload size, separately for ingress and egress. Egress is usually the cost and bottleneck (media).
5. **Pick the binding resource** per service (the 4 golden resources).
6. **Divide demand by a single-box threshold** to get server count, add headroom.
7. **Sanity-check** against known thresholds (below) and round to a power of ten.

Single-box thresholds worth memorizing (commodity hardware, rough):

Component	Order-of-magnitude capacity
Web/app server	~1k–10k QPS (depends on work per request)
Postgres/MySQL (single primary)	~5k–10k writes/s, more reads with replicas
Redis (single instance)	~100k+ ops/s (in-memory, single-threaded core)
Kafka (per broker)	~100s of MB/s, ~hundreds of k msgs/s
1 Gbps NIC	~125 MB/s; 10 Gbps ≈ 1.25 GB/s
SSD	~10k–500k IOPS, ~500 MB/s–several GB/s

10. Fully worked example — size a system for 10 million users

Scenario: a Twitter-like feed service. **Assumptions (stated):**

- 10 M DAU.
- Each user posts **2 times/day** (write) and reads their feed **20 times/day** (read) → ~10:1 read:write... but reads are cheap lookups while a post fans out. Let's compute both.
- Average post = **300 bytes** of text + metadata; assume 10% include an image of ~**200 KB**.
- Peak factor = **3×**.
- Retention: keep posts **5 years**. Replication ×3.

Step 1 — Write QPS (posts):

```
10,000,000 users × 2 posts/day = 20,000,000 posts/day
20,000,000 ÷ 86,400 (~10^5)    ≈ 230 writes/sec average
× 3 peak                      ≈ 700 writes/sec peak
```

700 writes/sec is comfortably within a single well-tuned Postgres primary (~5–10k writes/s). **No write sharding needed yet** — a key, money-saving conclusion.

Step 2 — Read QPS (feed loads):

```
10,000,000 × 20 reads/day = 200,000,000 reads/day
÷ 86,400                ≈ 2,300 reads/sec average
× 3 peak                ≈ 7,000 reads/sec peak
```

7k reads/sec exceeds a single primary's comfort zone → **cache (Redis) + read replicas**. With a 90% cache hit rate, only ~700 reads/sec hit the DB. This is the caching lever the read:write ratio predicted.

Step 3 — Storage/year:

```
Text: 20,000,000 posts/day × 300 B = 6 GB/day
Images: 20,000,000 × 10% × 200 KB = 2,000,000 × 200 KB = 400 GB/day
Total raw ≈ 406 GB/day ≈ ~0.4 TB/day

Per year: 0.4 TB × 365 ≈ ~150 TB/year (raw)
× 3 replication                ≈ 450 TB/year
× 5 years retention             ≈ 2.25 PB
```

Text is negligible (~2 TB over 5 years); **images dominate by ~70×**. Conclusion: text/metadata lives in Postgres; **media goes to object storage (S3) + CDN**, not the database. Sizing surfaced the architecture.

Step 4 — Bandwidth (egress, the expensive direction):

```
Feed read returns ~20 posts. Mostly text + thumbnail.
Egress per read ≈ ~50 KB (text + a few thumbnails).
7,000 reads/sec × 50 KB ≈ 350 MB/sec ≈ ~2.8 Gbps peak egress
```

2.8 Gbps overwhelms a single 1 Gbps NIC and is the textbook case for a **CDN** to absorb media egress at the edge ([06-caching-and-cdn.md](#)). Origin then serves only dynamic text.

Step 5 — Server count (app tier): Assume each app server handles ~2,000 QPS of this workload. Total peak QPS ≈ 700 + 7,000 ≈ 7,700.

```
7,700 ÷ 2,000 ≈ 4 servers; ×2 for redundancy/headroom ≈ ~8 app servers
```

Step 6 — Little's Law cross-check (DB pool): at 700 DB-reaching reads/sec (post-cache) + 700 writes/sec ≈ 1,400 req/s to DB, $W \approx 20 \text{ ms} \rightarrow L = 1,400 \times 0.02 = \mathbf{28 \text{ connections}}$ in flight. A pool of ~50 across replicas is ample.

The one-paragraph answer you'd give: “~700 writes/s and ~7k reads/s at peak. Writes fit one Postgres primary; reads need Redis (90% hit) plus read replicas. ~2.25 PB over 5 years dominated by images, so media → S3 + CDN and metadata → Postgres. ~2.8 Gbps egress means a CDN is mandatory. ~8 app servers with headroom. No sharding required at 10 M users — revisit at ~100 M.”

That's a complete, defensible capacity plan from five assumptions and a calculator.

11. Common pitfalls / war stories

- **Designing for average, getting killed by peak.** A team sizes for 230 writes/s average; a product launch in one time zone drives a 15× spike and the DB falls over. Always carry a peak factor and a load-shedding plan.

- **Reporting mean latency on a dashboard.** A service “averages 40 ms” while 1% of requests take 4 s. The biggest customer (millions of requests) complains constantly; the dashboard says everything’s fine. Switch SLOs to p99/p999.
- **Averaging percentiles across hosts.** A monitoring system computes “global p99” by averaging per-host p99s. It’s mathematically meaningless and hides outlier hosts. Aggregate histograms, not percentiles.
- **The 2³¹ integer overflow.** A signed 32-bit auto-increment ID silently caps at 2,147,483,647. High-volume tables hit it and inserts start failing in production with cryptic errors. Use 64-bit (bigint) IDs from day one. (Your [CLAUDE.md](#) mandates UUIDs — that’s the more robust answer.)
- **Ignoring replication and index overhead in storage.** Raw data size × 1 looks fine; reality is ×3 replication × ~2 for indexes/metadata = ~6× your napkin number. Storage estimates that forget this are off by an order of magnitude.
- **Forgetting the network round trip is ~5,000 RAM accesses.** The N+1 query problem (100 round trips instead of 1) turns a 1 ms operation into a 50 ms one. This is the single most common real-world latency bug — and it’s right in your own coding standards.
- **Fan-out p99 surprise.** Each microservice “looks fast” (p99 = 50 ms), but a request that fans out to 50 of them in series/parallel has a p99 in the seconds. Nobody owns the end-to-end number, so nobody sees it until users complain.
- **Treating cross-region latency as tunable.** A team tries to “optimize away” a 150 ms cross-continent hop with code. It’s physics. The only fix is a regional replica or edge cache.

🌱 Case Study: Twitter’s Timeline (fan-out and the BOTE that justified it)

The problem. By the early 2010s Twitter served on the order of **150 million daily active users** posting **~300–400 million tweets/day** (roughly **5,000–6,000 tweets/sec average**, with documented spikes to **~150,000 tweets/sec** during live events like the 2013 “Castle in the Sky” TV broadcast). But the brutal number wasn’t writes — it was reads. The home timeline (“who I follow, newest first”) was requested on the order of **hundreds of thousands of times per second**, a **read:write ratio well into the hundreds-to-one** range. That ratio is the entire story, and it’s exactly the lever from §4: a read-heavy system is won or lost on how cheap a read is.

Step 1 — size it the way §9 says. Run the arithmetic Twitter’s engineers effectively ran:

```
Writes:  ~400M tweets/day ÷ ~105 s/day  ≈ ~4,600 tweets/sec average
          ×3–30 peak ≈ many tens of k/sec
Reads:   timeline loads >> tweets       ≈ ~300k timeline reads/sec
Ratio:   reads/writes ≈ several hundred : 1  ← the design driver
```

A few thousand writes/sec fits comfortably on sharded storage (recall §10: even a single primary handles ~5–10k writes/s). **Reads are the wall.** The naive read — “on every timeline load, query the tweets of everyone I follow, merge, sort by time” — is a *fan-out-on-read*. For a user following 1,000 accounts that’s a

1,000-way scatter-gather per request, hundreds of thousands of times per second. That's the §8 trap in its purest form.

Step 2 — the fan-out cost, made concrete. Twitter's decisive move was **fan-out-on-write**: when you tweet, the system pushes that tweet's ID into a precomputed, per-user "home timeline" cache (held in Redis) for **every one of your followers**, *at write time*. A read then becomes a single sequential lookup of an already-built list — the §10 "precompute feeds" pattern, paid for once at write instead of repeatedly at read.

FAN-OUT-ON-WRITE (the chosen path)

```
you tweet → [delivery fanout] → push tweet-id into
                                follower_1 timeline (Redis)
                                follower_2 timeline (Redis)
                                ... (one write per follower)
                                follower_N timeline (Redis)

follower reads timeline → ONE Redis range read (O(1)-ish, ~ms)
```

The trade is explicit in the math. With an average of a few hundred followers, **one write triggers a few hundred cache writes** — but it converts a hundreds-of-thousands-QPS *read* fan-out into a hundreds-of-thousands-QPS *cheap list read*, while the expensive fan-out happens at the far lower write rate (a few thousand/sec). You move work from the hot path (reads) to the cold path (writes), exactly because the read:write ratio said reads are ~100× more frequent. Storage cost is real: every follower keeps a materialized list of recent tweet IDs in RAM — but IDs are ~8–16 bytes (§3), so a 800-entry timeline is only ~kilobytes, and Redis does **~100k+ ops/sec per instance** (§9 thresholds), which is why an in-memory tier was mandatory rather than hitting the DB.

Step 3 — where fan-out-on-write breaks, and the hybrid. Lady Gaga / Obama-class accounts had **tens of millions of followers**. A single tweet from them would mean tens of millions of cache writes — a write *amplification* that stalls the delivery pipeline and delays everyone's timeline. So Twitter accepted a **hybrid**: ordinary accounts use fan-out-on-write; a small set of celebrity accounts are *excluded* from fan-out, and their tweets are **merged in at read time** for followers who include them. Most timelines are pure cache reads; the handful that follow celebrities pay a small merge. This is the §8 mitigation "reduce fan-out width" applied at the data-model level.

The trade-off they accepted. They gave up **write simplicity and write latency** (and a lot of RAM for duplicated timelines) to buy **read latency and read throughput**. They also accepted **eventual consistency** — your followers don't all see a tweet at the same instant; the fan-out drains over seconds. For a social feed that's fine; for a ledger (§4) it would be unacceptable. And they accepted the operational complexity of two code paths (the hybrid) rather than one clean rule.

Results. The materialized-timeline architecture (the "Timeline" / Redis-backed delivery system Twitter described publicly around 2013) drove home-timeline reads to roughly **a few milliseconds at the median**, served from RAM, while keeping the system stable at **~150k tweets/sec peaks**. Tellingly, when they re-engineered for tail latency, the wins were quoted in **percentiles, not averages** (§7): reported **p99 home-timeline latencies in the low hundreds of milliseconds**, with the explicit goal of shrinking the **p50 → p99**

gap — because a power user refreshing constantly hits the tail repeatedly (§7’s “your best customers live on the tail”).

Lessons

- **The read:write ratio chooses your architecture before anything else does.** Hundreds-to-one reads → precompute on write and serve from RAM; the BOTE in §4/§10 told them this in the first minute.
- **Fan-out is a cost you can move, not erase.** Pay it on the rarer event (writes) when reads dominate — but watch for write-amplification on high-degree nodes and break the rule (hybrid) exactly where the numbers say it hurts.
- **Materializing for reads costs RAM and consistency.** That’s an acceptable trade for a feed, a wrong one for money — always name what you’re giving up.
- **Report and optimize the tail.** “A few ms median” hid nothing only because they also tracked p99 and the p50 → p99 gap; the average would have lied (§7).

12. Test yourself

1. A service receives **500 M requests/day**. What is the average QPS, and a reasonable peak QPS? (*Hint: $\div 10^5$ for average; $\times 2\text{--}3$ for peak.*)
2. You have a thread pool of **200 threads** and each request takes **$W = 100$ ms**. What’s the max sustainable throughput? (*Hint: Little’s Law backwards: $\lambda = L/W$.*)
3. Your single-backend p99 is **99.9% under 50 ms (i.e. 0.1% slow)**. You fan out to **200 backends** and wait for all. Roughly what fraction of requests hit the slow path? (*Hint: $1 - 0.999^{200}$.*)
4. Why can a service with a **40 ms mean** latency still be a serious reliability problem? (*Hint: what does the mean hide? Think p99/p999 and which users hit it.*)
5. You’re storing **100 M new rows/day** at **500 bytes each**. How much raw storage per year, and what’s the realistic figure after $\times 3$ replication and $\times 2$ index overhead? (*Hint: bytes/day → $\times 365$ → $\times 6$.*)
6. A box shows CPU 30%, memory 45%, disk-IOPS 95%, network 20%. You add more CPU cores. What happens to throughput, and what should you have done? (*Hint: bottleneck dimension.*)
7. Why is **sequential** disk access $\sim 100\times$ faster than **random**, and name two real systems designed around this fact. (*Hint: seek time; LSM-trees / Kafka logs.*)
8. Without a calculator, convert **2 billion events/day** to per-second. (*Hint: $\div \sim 10^5$ → then $\times 2$.*)

Answers: (1) $\sim 5,000$ avg, $\sim 10\text{--}15k$ peak. (2) $\lambda = 200/0.1 = 2,000$ req/s. (3) $1 - 0.999^{200} \approx 18\%$. (4) The mean hides the tail; heavy users hit p99/p999 constantly; SLOs must be percentile-based. (5) 50 GB/day → ~ 18 TB/yr raw → ~ 110 TB/yr after $\times 6$. (6) Throughput unchanged — disk IOPS is the binding constraint; scale storage/IOPS or shard. (7) Random access pays seek/rotational latency each time; Kafka and LSM-tree stores (RocksDB, Cassandra) turn writes into sequential appends. (8) $2 \times 10^9 \div 10^5 = 20,000/s$ average.)

13. Further reading

- **Jeff Dean** — “**Latency Numbers Every Programmer Should Know**” (the canonical table; see the interactive “*Latency Numbers*” visualization by Colin Scott that updates them by year).
 - **Martin Kleppmann** — ***Designing Data-Intensive Applications (DDIA)***, Ch. 1 (reliability/scalability/maintainability) and the percentiles/SLO discussion — the single best book for this entire curriculum.
 - **Dean & Barroso** — “**The Tail at Scale**” (Communications of the ACM, 2013) — the definitive paper on tail-latency amplification and hedged requests.
 - **The System Design Primer** (GitHub: donnemartin/system-design-primer) — the “Powers of two” and “Latency numbers” appendices, plus worked back-of-envelope examples.
 - **Brendan Gregg** — ***Systems Performance*** — deep treatment of the four resources, USE method (Utilization/Saturation/Errors), and Little’s Law in practice.
-

Next: [02-](#) onward build on these numbers. Whenever a later module says “this won’t scale past X,” come back here and prove it with the arithmetic. Estimation is the skill you’ll use in every other module — see [07-load-balancing-and-scaling.md](#) for what to do once a single box’s threshold is exceeded.

Networking & Protocols

Source: [readme/research/system-design/02-networking-and-protocols.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains how data travels between a user's browser and a server — and why some websites feel instant while others feel slow. It shows that most of the delay is not about bandwidth ("pipe size") but about the number of back-and-forth messages two computers must exchange before any real data flows. Understanding this helps you make better decisions about speed, real-time features, and infrastructure.

The main parts explained simply:

- **Round-trip tax** — Every time a browser connects to a server, they must exchange several "hello" messages before any useful data moves. Each exchange takes time (a "round-trip"). A cold start over a secure connection can cost 3–5 of these round-trips before a single byte of your page arrives. Cutting round-trips is the single biggest lever for a faster site.
- **TCP vs UDP** — TCP is the reliable delivery method: every byte arrives in order, nothing is lost. UDP is the "fire and forget" method: faster, but packets can be lost or arrive out of order. Most websites use TCP; live video calls and the newer HTTP/3 use UDP because dropping a late packet is better than waiting for it.
- **TLS (the padlock / HTTPS)** — Encryption that protects data in transit. Older TLS needed 2 extra round-trips to set up; the newer version (TLS 1.3) cuts that to 1, and on a return visit can skip the setup entirely (0 round-trips). Never use the "0-RTT" shortcut for actions that move money — it can be replayed by attackers.
- **DNS** — The internet's phone book. Before your browser can connect to a server it must look up the server's address by name. This lookup itself costs time, which is why CDNs and caching matter even before a page starts loading.
- **CDNs and anycast** — A CDN places copies of your content at hundreds of locations around the world. Users are served from the nearest one, turning a 120 ms round-trip into a 5 ms one. Anycast routes traffic to the nearest location automatically at the network level, with no lag from DNS caching.
- **HTTP versions (1.1 / 2 / 3)** — HTTP/1.1 could only handle one request at a time per connection. HTTP/2 handles many at once, but a single lost packet can still freeze all of them. HTTP/3 fixes that by treating each request independently — a lost packet only delays its own request, not everyone else's.
- **Real-time options (WebSocket, SSE, polling)** — For live updates (chat, notifications, order status): SSE is the simple choice when the server just needs to push updates to the browser; WebSocket is needed when the browser also sends data back continuously (chat, collaborative editing). Plain polling (asking "anything new?" every few seconds) wastes resources and should be a last resort.
- **gRPC** — A compact, fast way for servers to talk to each other internally. Not usable directly from a browser without an extra translation layer (gRPC-Web + a proxy like Envoy).

What to do with this: If your site feels slow, the first question is "how many round-trips does a cold page load cost?" — reduce those before anything else (use a CDN, enable TLS 1.3, reuse connections). For any feature that needs live updates (order tracking, notifications), prefer SSE for one-way push and WebSocket only when you need two-way communication.

What you'll learn. How data actually moves between a browser and a server — from the IP packet up through TCP, TLS, and the HTTP family (1.1 / 2 / 3) — and how every layer adds or removes round-trips that you can *count*. By the end you'll be able to reason about why a cold HTTPS request costs ~3–4 RTTs, why HTTP/2 fixed one head-of-line-blocking problem but not the deeper one, and when to reach for WebSockets vs SSE vs gRPC streaming.

Prerequisites. Read `01-foundations-and-estimation.md` first — specifically the latency-numbers table. This module constantly converts protocol behaviour into *round-trips*, and a round-trip is only meaningful if you've internalised that a same-region RTT is ~0.5–1 ms, cross-US ~40 ms, and cross-Atlantic ~80–140 ms. Networking is the dominant tax on user-perceived latency, and almost all of that tax is *round-trips* × *RTT*.

1. The mental model: layers and the “round-trip tax”

The single most useful framing for a senior engineer: **latency is mostly counted in round-trips, and each protocol layer adds a fixed number of them before any useful byte flows.** Bandwidth rarely limits a small request; the speed of light and the number of handshakes do.

A cold request to `https://api.example.com` from a fresh browser:

```
DNS lookup      ~1 RTT (often more; uncached)
TCP handshake   1 RTT  (SYN / SYN-ACK / ACK)
TLS 1.2 handshake 2 RTT  (TLS 1.3: 1 RTT, or 0 with resumption)
HTTP request    1 RTT  (send request, get first byte)
-----
Total before data ~5 RTT cold, ~1 RTT warm
```

At 80 ms cross-Atlantic RTT, that cold path is ~400 ms *before the server even starts working*. This is why connection reuse, TLS resumption, CDNs (shorter RTT), and HTTP/3 (fewer handshakes) all matter so much — they each shave whole RTTs.

OSI vs TCP/IP

OSI is the 7-layer teaching model; the real internet runs the 4-layer TCP/IP model. Know the mapping because interviewers and RFCs use both.

OSI layer	TCP/IP layer	What lives here	Examples
7 Application / 6 Presentation / 5 Session	Application	App protocols, encryption, serialization	HTTP, gRPC, DNS, TLS*
4 Transport	Transport	Ports, reliability, ordering	TCP, UDP, QUIC*
3 Network	Internet	Addressing, routing	IP, ICMP, BGP
2 Data Link / 1 Physical	Link	Frames on the wire	Ethernet, Wi-Fi

*TLS is awkward: it sits between transport and application. QUIC blurs it further — it runs *over UDP* but absorbs TLS and parts of TCP’s job into the application layer (userspace). Keep that in mind; it’s the key to understanding HTTP/3.

Encapsulation: each layer wraps the one above. `[ Ethernet [ IP [ TCP [ TLS [ HTTP ] ] ] ] ]`. Routers read IP headers; switches read Ethernet; the server’s kernel reassembles TCP; userspace decrypts TLS and parses HTTP.

2. Transport layer: TCP vs UDP

TCP — reliable, ordered, connection-oriented

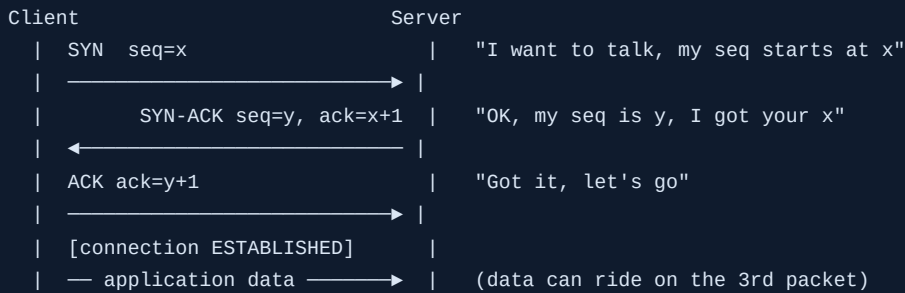
TCP gives you a **reliable, in-order byte stream** over an unreliable packet network. It achieves this with sequence numbers, acknowledgements (ACKs), retransmission of lost segments, and a checksum. It also adds **flow control** (don’t overrun the receiver) and **congestion control** (don’t overrun the network). All of this costs round-trips and head-of-line blocking — the price of reliability.

UDP — fire and forget

UDP is a thin wrapper over IP: just ports and a checksum. No handshake, no ordering, no retransmission, no congestion control. You get datagrams that may arrive out of order, duplicated, or not at all. That’s a *feature* when you’d rather drop a late packet than wait for it (live audio/video, gaming) — and it’s the foundation QUIC builds reliability on top of *in userspace*, so it can iterate faster than kernel TCP.

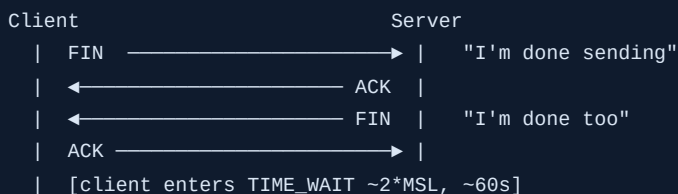
Dimension	TCP	UDP	When to choose
Connection	Yes (handshake)	None	TCP for request/response; UDP for one-shot/streaming
Reliability	Guaranteed delivery + retransmit	None	TCP when every byte matters (web, DB, files)
Ordering	In-order byte stream	No ordering	TCP for streams; UDP when app reorders or doesn’t care
Head-of-line blocking	Yes (one lost segment stalls all)	No	UDP for media where a late frame is useless
Congestion control	Built-in	App’s job	TCP for fairness; UDP must add its own (QUIC does)
Overhead	20-byte header + handshake	8-byte header	UDP for latency-critical small messages
Multiplexing	One stream per connection	Per-datagram	—
Examples	HTTP/1-2, TLS, SSH, DB	DNS, QUIC/HTTP3, VoIP, gaming, video	DNS uses UDP, falls back to TCP for large responses

The TCP 3-way handshake



That's **1 full RTT** before you can send data. The client and server also negotiate the **Maximum Segment Size (MSS)** and options like window scaling and SACK here. *TCP Fast Open (TFO)* can carry data on the SYN to save the RTT on repeat connections, but middlebox breakage limited its adoption.

Connection teardown



The four-way close exists because each side closes its half independently. The gotcha: the side that closes first sits in **TIME_WAIT** (~60 s) to absorb stray late packets and prevent old segments contaminating a new connection on the same 4-tuple. On a busy load balancer or proxy that opens/closes many short connections, you can exhaust ephemeral ports with tens of thousands of sockets stuck in TIME_WAIT — a classic production failure (see war stories).

Flow control vs congestion control

These are different and frequently confused.

- **Flow control** protects the *receiver*. The receiver advertises a **receive window (rwnd)** in every ACK (“I have room for N more bytes”). The sender never sends more unacked data than the window. A slow consumer naturally throttles a fast producer.
- **Congestion control** protects the *network*. The sender maintains a **congestion window (cwnd)** and probes for available bandwidth, treating packet loss (or, in modern algorithms, delay) as the signal to back off. Effective send window = `min(rwnd, cwnd)`.

Slow start: a new connection doesn’t know the available bandwidth, so it starts with a small cwnd (historically ~10 MSS ≈ ~14 KB, per RFC 6928) and **doubles cwnd every RTT** until it hits a threshold (`ssthresh`) or loss. This is why a brand-new connection is slow even on a fat pipe — and why **connection reuse is so valuable**: an established, warmed-up connection already has a large cwnd. After `ssthresh`, it switches to **congestion avoidance** (linear growth). On loss it backs off (multiplicative decrease — “AIMD”).

Algorithms: Reno/CUBIC (loss-based; CUBIC is the Linux default) vs **BBR** (Google; models bottleneck bandwidth + RTT, doesn't wait for loss — much better on lossy/high-BDP links and widely deployed on YouTube/GCP).

Bandwidth-Delay Product (BDP): $BDP = \text{bandwidth} \times RTT$. This is the amount of data “in flight” to keep a pipe full. Example: $100 \text{ Mbps} \times 80 \text{ ms} = 12.5 \text{ MB/s} \times 0.08 \text{ s} \approx 1 \text{ MB}$. If your TCP window (rwnd, capped without **window scaling**) is smaller than the BDP, you *cannot* saturate the link no matter the bandwidth — the sender stalls waiting for ACKs. High-BDP “long fat networks” (satellite, transcontinental) need window scaling enabled (it is, by default, since ~2 decades) and benefit hugely from BBR.

3. TLS: encryption and its handshake cost

TLS sits on top of TCP (for HTTP/1-2) and provides confidentiality, integrity, and authentication (the server proves identity via an X.509 certificate signed by a CA). The handshake is where the RTTs go.

TLS 1.2 — 2 RTTs



Two round-trips *on top of* the TCP handshake. Cold HTTPS = 1 (TCP) + 2 (TLS) = 3 RTTs before the HTTP request.

TLS 1.3 — 1 RTT, or 0-RTT resumption

TLS 1.3 (RFC 8446) is a major redesign: it removed legacy ciphers, made forward secrecy mandatory (ephemeral Diffie-Hellman only), and **cut the handshake to 1 RTT** by having the client *guess* the key-share in its first message.



0-RTT resumption: on a *return* visit, the client can send application data in its very first packet using a pre-shared key (PSK) from the prior session. Zero handshake RTTs. The catch: **0-RTT data is replayable** — an attacker can capture and resend it. So 0-RTT is safe only for idempotent requests (a GET, never a “transfer \$100”). Treat 0-RTT requests as untrusted-for-replay at the application layer.

	TLS 1.2	TLS 1.3
Handshake RTTs	2	1 (0 on resumption)
Forward secrecy	Optional	Mandatory (ephemeral DH)
Cipher suites	Many, some weak	Small, all AEAD
Resumption	Session IDs/tickets	PSK, 0-RTT
Replay risk	—	0-RTT data is replayable

TLS termination & mTLS

TLS termination = where encryption ends. Usually at the load balancer / CDN edge (e.g., AWS ALB, Cloudflare, an Nginx/Envoy proxy), which decrypts and forwards plaintext (or re-encrypts) to backends. Terminating at the edge offloads CPU from app servers and centralises certificate management — but the internal hop is now plaintext unless you re-encrypt. **TLS passthrough** forwards encrypted bytes to the backend (needed when the backend must see the client cert).

mTLS (mutual TLS): *both* sides present certificates. The server verifies the client’s cert too. This is the backbone of zero-trust service-to-service auth in service meshes (Istio/Linkerd via Envoy sidecars) — every internal call is mutually authenticated and encrypted, removing the “trusted network” assumption. Cost: cert issuance/rotation infrastructure (e.g., SPIFFE/SPIRE) and the extra handshake work.

4. DNS: turning names into addresses

Before any TCP handshake, the client must resolve the hostname. The resolution path:

```
Browser cache → OS cache → Recursive resolver (ISP / 1.1.1.1 / 8.8.8.8)
  → Root nameserver (.)      "ask the .com servers"
  → TLD nameserver (.com)    "ask example.com's authoritative NS"
  → Authoritative NS         "example.com = 93.184.216.34"
  → answer cached per TTL, returned to browser
```

A fully cold lookup can be multiple RTTs to different servers; in practice the recursive resolver caches aggressively, so most lookups are ~1 RTT or served from cache. **TTL** controls cache lifetime — low TTLs enable fast failover/rebalancing but increase lookup load and resolver pressure.

DNS as a load balancer / GeoDNS: the authoritative server can return *different answers* based on the resolver’s location (GeoDNS) or in round-robin/weighted fashion. This is how providers like AWS Route 53 (latency-based, geolocation, weighted routing), Cloudflare, and NS1 do global traffic steering. Limitation:

DNS-based LB is coarse — caching and TTL mean you can't react instantly, and you balance *per-resolver*, not *per-user*. That's why anycast (below) is often preferred for the finest-grained, fastest steering.

DoH/DoT (DNS over HTTPS/TLS) encrypt the lookup for privacy. Modern apps and browsers increasingly default to them.

5. CDNs and anycast

A **CDN** (Cloudflare, Akamai, Fastly, AWS CloudFront, Google Cloud CDN) places content close to users at hundreds of edge **PoPs (points of presence)**. The win is twofold: (1) **shorter RTT** — serving from a PoP 5 ms away instead of an origin 120 ms away collapses every handshake RTT and every congestion-control RTT; (2) **offload** — the origin sees only cache misses. Reread the round-trip math in §1: a CDN that cuts RTT from 120 ms to 5 ms makes the *entire* cold path 24× cheaper.

Anycast is the routing trick that makes this work: the *same IP address* is announced from many PoPs via BGP, and the internet's routing naturally sends each client to the topologically nearest PoP. Unlike GeoDNS, anycast steers at the *network* layer with no DNS-cache lag and provides instant failover (withdraw a PoP's BGP announcement and traffic reroutes). Cloudflare and Google's frontends are built on anycast. The historical caveat — TCP connections could in theory flap between PoPs mid-flow — is handled by stable routing and connection-aware edges.

CDNs also do **edge TLS termination** and increasingly **edge compute** (Cloudflare Workers, Lambda@Edge, Fastly Compute) — running logic at the PoP so even dynamic responses skip the origin round-trip.

6. The HTTP family

Keep-alive & connection pooling

HTTP/1.0 opened a new TCP+TLS connection per request — catastrophic given §1's RTT tax. **HTTP/1.1 keep-alive** (`Connection: keep-alive` , default) reuses the connection for many requests, amortising the handshake *and* keeping cwnd warm. **Connection pooling** is the client-side counterpart: browsers keep ~6 connections per origin; backend HTTP clients (and DB drivers — see `05-databases-and-storage.md`) maintain a pool to avoid per-call handshakes. Pool sizing is a real tuning problem: too small → queueing; too large → resource exhaustion and connection storms on restart.

HTTP/1.1's flaw: head-of-line blocking

On a single HTTP/1.1 connection, responses must come back **in request order**. A slow first response blocks every response behind it — **HOL blocking at the HTTP layer**. The 1990s workaround was “pipelining” (rarely worked due to proxies) and “domain sharding” (spread assets over many hostnames to get more parallel connections — an ugly hack that fights congestion control).

HTTP/2: multiplexing over one connection

HTTP/2 (RFC 7540) keeps the HTTP semantics but changes the wire format: it's **binary** and splits everything into **frames** belonging to numbered **streams**. Many streams multiplex over **one TCP connection**, interleaved, so a slow response no longer blocks others *at the HTTP layer*. It adds **header compression (HPACK)** (HTTP headers are hugely repetitive) and **stream prioritisation**. Server push existed but was a flop and is deprecated.

But HTTP/2 still suffers TCP head-of-line blocking. All those streams share *one* TCP byte stream. If a single TCP segment is lost, the kernel holds back *all* subsequent bytes — for *every* multiplexed stream — until the retransmission arrives, because TCP guarantees in-order delivery to the application. On a clean network HTTP/2 is great; on a lossy mobile link, one lost packet stalls all your “parallel” streams. This is the deeper HOL problem HTTP/2 *couldn't* solve, because it lives below HTTP, in TCP.

HTTP/3: QUIC kills TCP head-of-line blocking

HTTP/3 (RFC 9114) runs over **QUIC** (RFC 9000), which runs over **UDP**. QUIC reimplements reliability, ordering, congestion control, and TLS 1.3 *in userspace*, with streams as first-class citizens. Because QUIC understands streams natively, **a lost packet only stalls the stream it belonged to** — other streams keep flowing. That's the fix HTTP/2 couldn't make.

Other QUIC wins:

- **Faster handshake:** transport + TLS 1.3 are combined into **1 RTT** (0-RTT on resumption). No separate TCP-then-TLS staircase.
- **Connection migration:** a QUIC connection is identified by a **Connection ID**, not the IP/port 4-tuple. Switch from Wi-Fi to cellular and the connection survives the IP change — no re-handshake. Huge for mobile.
- Always encrypted; harder for middleboxes to ossify/interfere.

Downsides: UDP is sometimes throttled/blocked by firewalls (so clients fall back to TCP/HTTP2), and userspace congestion control historically cost more CPU than kernel TCP (improving with offload).

	HTTP/1.1	HTTP/2	HTTP/3
Transport	TCP	TCP	QUIC over UDP
Wire format	Text	Binary frames	Binary frames
Concurrency	1 req/response per conn (~6 conns)	Multiplexed streams, 1 conn	Multiplexed streams, 1 conn
HOL blocking	HTTP-layer (per conn)	TCP-layer only	None (per-stream loss isolation)
Header compression	None	HPACK	QPACK
Handshake RTTs (cold)	TCP + TLS (3)	TCP + TLS (3)	1 (combined), 0 on resume
Connection migration	No	No	Yes (Connection ID)
When to use	Legacy/simple, debuggable	Default for most HTTPS sites	Lossy/mobile networks, latency-critical CDNs

Practical note: you usually don't *choose* — you enable all three at the edge (CDN/LB) and clients negotiate the best via ALPN (in TLS) and the `Alt-Svc` header (advertises HTTP/3). Cloudflare, Google, Meta serve HTTP/3 broadly today.

7. Real-time: WebSocket vs SSE vs polling vs gRPC

HTTP is request/response. For server → client push you need something else. (See `07-real-time-and-messaging.md` for the messaging side.)

- **Short polling:** client asks “anything new?” every N seconds. Simple, but wasteful (most responses are empty) and latency = polling interval.
- **Long polling:** client request *hangs open* until the server has data (or times out), then immediately re-requests. Lower latency, fewer empty responses, but a held connection per client and re-request overhead.
- **Server-Sent Events (SSE):** a single long-lived HTTP response that the server keeps writing to (`text/event-stream`). **One-directional** (server → client only), text-only, but dead simple, rides ordinary HTTP/2, and the browser `EventSource` auto-reconnects with `Last-Event-ID`. Great for notifications, live feeds, dashboards, LLM token streaming.
- **WebSocket:** a full **bidirectional** binary/text channel. Starts as an HTTP request with `Upgrade: websocket`, then the connection is “upgraded” off HTTP into a raw framed duplex socket. Best for chat, multiplayer, collaborative editing, trading. Cost: it's not plain HTTP, so proxies/LBs need WS support, and you manage your own reconnect/heartbeat/backpressure.

```

WebSocket upgrade:
Client → GET /ws Upgrade: websocket, Sec-WebSocket-Key: ...
Server → 101 Switching Protocols, Sec-WebSocket-Accept: ...
      [now a bidirectional frame stream, no more HTTP]

```

	Short poll	Long poll	SSE	WebSocket
Direction	C → S, pull	C → S, pull (held)	S → C only	Bidirectional
Transport	HTTP	HTTP	HTTP (1 conn)	TCP via HTTP upgrade
Latency	= interval	low	low	lowest
Server cost	high (empty reqs)	held conns	held conns	held conns
Auto-reconnect	n/a	manual	built-in (EventSource)	manual
Binary	no	no	no (text)	yes
Proxy/LB friendliness	trivial	easy	easy	needs WS support
When to use	trivial/legacy	fallback	feeds, notifications, LLM streaming	chat, games, collab, trading

gRPC over HTTP/2

gRPC is a binary RPC framework using **Protocol Buffers** (compact, schema-typed) over **HTTP/2**. It leans on HTTP/2 multiplexing and supports four call types: **unary** (request/response), **server streaming**, **client streaming**, and **bidirectional streaming** — all over a single connection's streams. It's the default for internal service-to-service comms (low overhead, strong contracts, code-gen, deadlines/cancellation propagation). Caveats: HTTP/2 is required, so it doesn't run natively in browsers — you need **gRPC-Web** (a proxy like Envoy translates) for browser clients. Compared to JSON/REST, gRPC trades human-readability for size, speed, and schema enforcement (see [08-apis-and-communication.md](#)).

8. Common pitfalls / war stories

- **TIME_WAIT port exhaustion.** A proxy or a service mesh sidecar opening a fresh short-lived connection per request piles up tens of thousands of sockets in TIME_WAIT and exhausts ephemeral ports → new connections fail with `EADDRNOTAVAIL`. Fix: **reuse connections (keep-alive / pooling)**, raise the ephemeral port range, enable `tcp_tw_reuse`. The real fix is almost always “stop opening so many connections.”
- **HTTP/2 made things slower on flaky mobile.** Teams enable HTTP/2, multiplex 100 streams over one TCP connection, then a single packet loss on a 4G link stalls *all* of them via TCP HOL blocking —

sometimes worse than HTTP/1.1's 6 independent connections (loss on one doesn't block the others). The fix is HTTP/3/QUIC, not more HTTP/2 tuning.

- **DNS TTL too high during a migration/failover.** You change the IP but resolvers (and stubborn clients/JVMs that cache DNS forever) keep hitting the dead origin for hours. Set low TTLs *before* a planned cutover; for instant failover prefer anycast or a health-checked LB behind a stable IP.
- **0-RTT replay.** A team enabled TLS 1.3 0-RTT for "speed" and an attacker replayed a captured early-data request. 0-RTT is replayable by design — never put non-idempotent operations on the 0-RTT path.
- **TLS terminated at the LB, plaintext inside.** "We're HTTPS!" — but the LB → app hop was plaintext over a shared VPC, and an internal foothold could sniff it. Re-encrypt internal hops or adopt mTLS for service-to-service.
- **Slow start tax on serverless / per-request connections.** Functions that open a fresh DB or HTTP connection per invocation pay the full handshake + cold cwnd every time. Persistent connection pools (or proxies like RDS Proxy / PgBouncer) keep windows warm.
- **WebSocket through a "smart" proxy.** A load balancer that doesn't honour `Upgrade` silently downgrades or kills WS connections; idle-timeout proxies drop long-lived WS/SSE without heartbeats. Add app-level pings and configure LB idle timeouts above your heartbeat interval.

Case Study: Cloudflare's HTTP/3 + QUIC + anycast edge

The problem. Cloudflare runs a reverse-proxy CDN sitting in front of millions of websites, serving on the order of tens of millions of HTTP requests *per second* across 300+ PoPs in 100+ countries. A huge slice of that traffic is mobile — phones on lossy 4G/5G links, behind carrier NAT, hopping between Wi-Fi and cellular. For those users the dominant cost is exactly the **round-trip tax from §1**: a cold HTTPS page load was paying DNS + TCP(1) + TLS(2) = ~3 RTTs of handshake *before the first byte*, and on a 2–5% packet-loss mobile link, HTTP/2's "parallel" streams were stalling on **TCP head-of-line blocking (§6)** — sometimes performing *worse* than HTTP/1.1's six independent connections. The challenge: cut the handshake RTTs, kill the HOL stall on lossy links, and survive the IP changes mobile clients constantly suffer — at planet scale, without asking site owners to do anything.

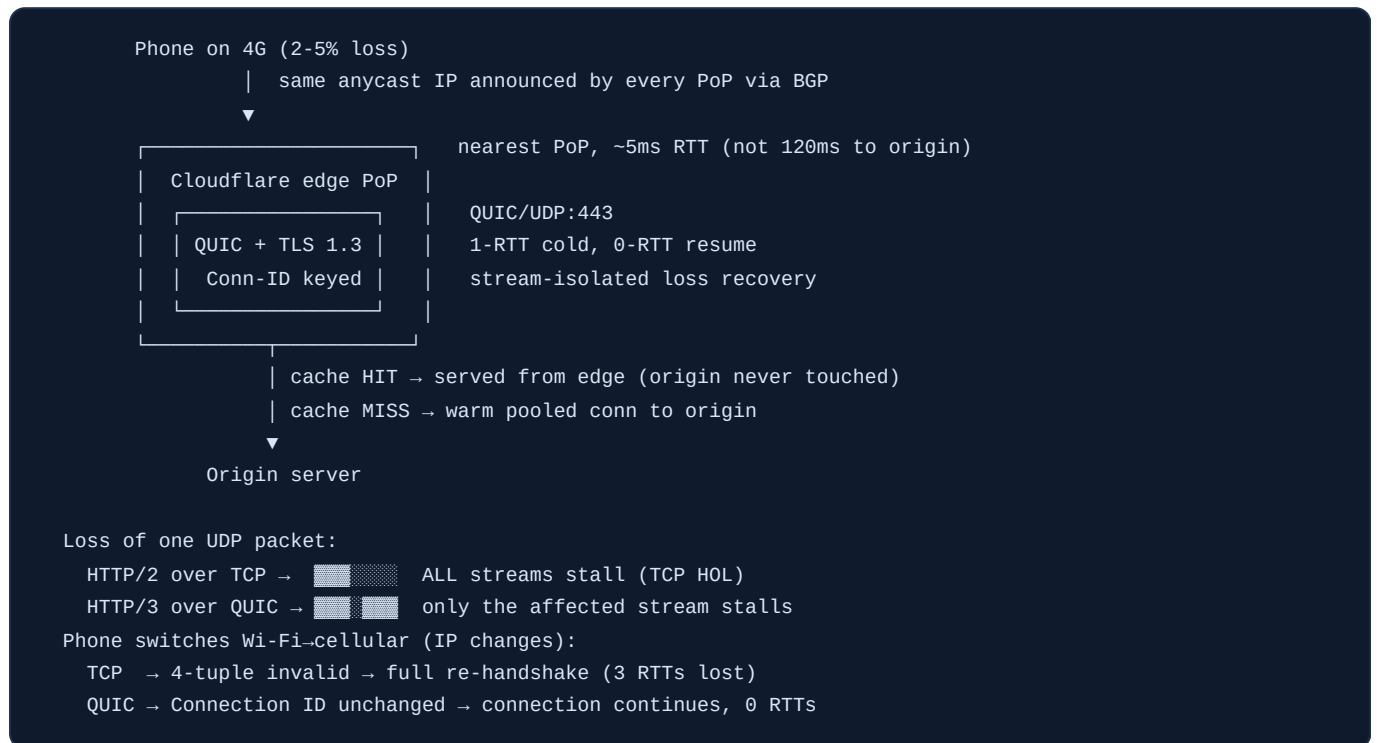
Applying the module's concepts.

Anycast routing (§5) gets the client to a near PoP with no DNS lag. Cloudflare announces the *same* IP from every PoP via BGP, so a client's packets reach the topologically nearest edge automatically. This is the **anycast pattern from the CDN section**: steering happens at the network layer, instantly, with no TTL/cache delay — so the RTT that *every* subsequent handshake is multiplied by is already as small as the network allows (often single-digit ms instead of 100+ ms to origin).

HTTP/3 / QUIC (§6) collapses the handshake and removes TCP HOL blocking. Once the packet lands at a near PoP, QUIC's combined transport+TLS-1.3 handshake replaces the TCP-then-TLS staircase, turning ~3 cold RTTs into **1 RTT — and 0-RTT on resumption** for returning visitors (exactly the **0-RTT resumption** described in §3 and the QUIC wins in §6). Because QUIC streams are first-class at the

transport layer, **a lost packet only stalls its own stream** — the precise fix HTTP/2 *couldn't* make because it lived above TCP. On a lossy mobile link, the other in-flight responses keep flowing.

Connection reuse + migration (§2, §6) keep the warmed-up path alive. A QUIC connection is keyed by a **Connection ID**, not the IP/port 4-tuple, so when a phone switches Wi-Fi → cellular the connection — and its warmed-up congestion window (the **slow-start tax** from §2 already paid) — *survives the IP change* with no re-handshake. This is connection reuse taken to its logical extreme: not just amortising one handshake, but preserving it across network changes.



The trade-off they accepted. QUIC runs in **userspace over UDP**, which costs noticeably more CPU than kernel TCP — early on, several times the per-byte send cost — and UDP is throttled or blocked by some firewalls and carrier middleboxes. Cloudflare accepted both: they invested heavily in optimising userspace congestion control and UDP send paths (GSO/segmentation offload, eBPF), and they **fail back gracefully** — clients that can't get UDP/443 through fall straight back to HTTP/2 over TCP. The bet was that the latency win for the majority on clean-enough networks outweighs the CPU cost and the fallback minority. They also accepted 0-RTT's **replay risk** (§3) by gating early-data to idempotent requests at the edge.

Results. Cloudflare reported HTTP/3 reaching first-byte and page-load improvements most pronounced exactly where the theory predicts — high-latency, high-loss connections, where eliminating TCP HOL blocking and saving handshake RTTs compound. Independent and Cloudflare measurements have shown meaningful tail-latency (p95/p99) reductions on lossy/mobile networks, with the largest wins on the slowest connections; connection migration eliminated reconnection stalls for users moving between networks. The handshake itself dropped from ~3 cold RTTs to 1 (0 on resume) — at 80 ms cross-region RTT that alone is ~160 ms shaved off a cold visit before any caching benefit.

Lessons

- **Latency wins come from removing whole round-trips, not shaving milliseconds.** Anycast (near PoP), QUIC's 1-RTT/0-RTT handshake, and connection migration each eliminate *entire RTTs* — the

highest-leverage optimisation per §1's round-trip math.

- **Solve a problem at the layer it actually lives in.** HTTP/2's HOL stall was a *TCP* problem; no amount of HTTP tuning fixed it. Moving reliability into userspace QUIC was the only place the fix could go.
- **A faster default protocol still needs a fallback.** UDP isn't universally allowed, so HTTP/3 is offered (via `Alt-Svc` /ALPN) with automatic TCP fallback — never a hard cutover that strands users.
- **Identity beats addressing for mobile.** Keying a connection by a stable Connection ID instead of the IP 4-tuple is what makes seamless network switching possible — a generalisable idea wherever endpoints roam.

9. Test yourself

1. **Count the RTTs** for a cold `https://` GET over TLS 1.2/HTTP1.1 vs a warm connection. (*Hint: DNS + TCP(1) + TLS1.2(2) + request(1); warm reuses TCP+TLS so ≈ 1 .*)
 2. Explain why HTTP/2 fixes HTTP-layer head-of-line blocking but **not** TCP-layer HOL blocking, and how HTTP/3 fixes the latter. (*Hint: streams share one ordered TCP byte stream; QUIC makes streams independent at the transport layer.*)
 3. What is the bandwidth-delay product of a 1 Gbps link with 50 ms RTT, and why does it matter for your TCP window? (*Hint: $125 \text{ MB/s} \times 0.05 \text{ s} \approx 6.25 \text{ MB}$ in flight; a smaller window can't fill the pipe.*)
 4. Why is TLS 1.3 0-RTT data unsafe for a “purchase” request but fine for fetching a product image? (*Hint: early data is replayable; idempotency.*)
 5. Contrast GeoDNS and anycast for global load balancing. Which reacts faster to a PoP failure and why? (*Hint: DNS caching/TTL lag vs BGP-level steering.*)
 6. When would you pick SSE over WebSocket, and vice versa? (*Hint: one-way push + auto-reconnect simplicity vs bidirectional/binary.*)
 7. Why does opening a new connection per request hurt even on a fast network with no DNS cost? (*Hint: handshake RTTs + cold congestion window / slow start.*)
 8. Your gRPC service works server-to-server but won't work from a browser. Why, and what's the fix? (*Hint: browsers can't speak raw HTTP/2 framing/trailers; gRPC-Web + Envoy proxy.*)
-

10. Further reading

- **Ilya Grigorik, *High Performance Browser Networking*** (O'Reilly, free online) — the definitive practical text on TCP, TLS, HTTP/1-2, WebSocket, and the RTT-counting mindset used throughout this module.
- **RFCs:** 9293 (TCP), 8446 (TLS 1.3), 9000 (QUIC), 9114 (HTTP/3), 7540 (HTTP/2), 6298 (TCP retransmission), 6928 (initial window), 6455 (WebSocket).
- **DDIA (Kleppmann), *Designing Data-Intensive Applications*** — networking assumptions behind distributed systems (unreliable networks, timeouts) in Ch. 8.
- **Cloudflare & Google blogs** on QUIC/HTTP/3, BBR, and anycast — excellent real-world deployment writeups.

- Siblings: `01-foundations-and-estimation.md` (latency numbers), `05-databases-and-storage.md` (connection pooling), `07-real-time-and-messaging.md` , `08-apis-and-communication.md` .

API Design & Service Communication

Source: [readme/research/system-design/03-api-design-and-communication.md](#)

IN SHORT — KEY TAKEAWAYS

This document is about how different parts of a software system talk to each other — and how to design those conversations so they are reliable, easy to change, and hard to break. If you have ever wondered why a website sometimes charges you twice after a slow connection, or why changing one thing in a backend can silently break something in an app, this document explains the rules engineers follow to prevent that.

The main parts explained simply:

- **Wait-for-reply vs fire-and-forget** — When your app asks for something, it can either wait for the answer before doing anything else (like a phone call), or send the request and move on, picking up the reply later (like a text message). Waiting is fine for small fast tasks like loading a page. Fire-and-forget is better for slow jobs like generating a PDF or sending an email — it keeps the app feeling fast.
- **REST, GraphQL, gRPC, tRPC — four ways to ask** — These are four different “languages” software services use to talk. REST is the most common and works like a simple menu of web addresses. GraphQL lets the asking side pick exactly which pieces of data it wants, cutting waste. gRPC is a fast, compact option used between internal services. tRPC works only when both sides are written in the same language (TypeScript). For most projects, REST is the right default.
- **Clean web addresses and correct reply codes** — A well-designed API uses neat address patterns (like `/orders/123`) and sends back the right status number: `200` means success, `404` means not found, `422` means the data you sent was wrong, `500` means the server itself broke. Using the right code matters because it tells the app whether to retry or not.
- **Versioning — how to improve an API without breaking old users** — When you change an API, old apps that rely on it can break silently (they just show wrong data with no error). The safe rule: only add new fields; never rename or remove existing ones. When you must break something, add the new field alongside the old one, update every user of the old field, then remove it later.
- **Pagination — avoiding the “show all” trap** — Loading thousands of records at once crashes databases. Offset-based paging (page 1, page 2...) slows down badly on large tables and can show duplicate or skipped rows when new data arrives. Cursor-based paging solves both problems: it always costs the same and stays stable even as new data is added.
- **Idempotency keys — preventing double charges** — If a network request times out, the app does not know whether it succeeded. For dangerous actions like charging a card, sending the same “key” (a unique ID) with every try lets the server recognize a repeat and return the original result instead of running the action again. This is how Stripe prevents double charging.
- **Rate limits and error messages** — APIs protect themselves by capping how many requests one caller can send per minute. When a limit is hit, the server returns `429` and tells the caller how long to wait before trying again. Error messages should always be structured and consistent so apps can handle them automatically — never just a raw number or vague phrase.
- **API gateway and BFF (Backend-for-Frontend)** — A gateway is a single entry door that handles security, routing, and logging before requests reach the real services. A BFF is a thin custom layer

built specifically for one type of user (the admin panel gets one, the customer storefront gets another), so each gets exactly the data shape it needs without wasted effort.

- **Webhooks — letting the server call you back** — Instead of asking “is it done yet?” every few seconds (polling), webhooks let the server send a message to your app the moment something happens. The receiver must verify the message is genuine (using a secret signature), handle duplicates gracefully (the same event can arrive more than once), and reply quickly.
- **Long-running jobs — accept now, finish later** — For slow tasks (PDF export, bulk import), the API should accept the request instantly with a **202 Accepted** reply and give back a link the caller can check for progress. Never hold the connection open while slow work finishes.

What to do with this: Always add new API fields without removing old ones until every consumer is updated — silent breakage from a renamed field is one of the most common bugs. For any action that charges money or has real consequences, use idempotency keys so retries never cause double side-effects.

What you’ll learn: How to choose between REST, GraphQL, gRPC, and tRPC, and when each genuinely wins; how synchronous request/response and asynchronous event-driven styles trade off; and the senior-level mechanics that keep an API healthy at scale — proper resource modeling, versioning without breaking consumers, cursor pagination, idempotency keys for safe retries, structured error contracts, BFFs and gateways, contract testing, webhooks, and long-running async operations.

Prerequisites: Read [01-fundamentals-and-scalability.md](#) (latency, throughput, scaling axes) and [02-...](#) (HTTP, TCP, TLS, DNS). This module assumes you know what an HTTP request/response is and how connections work. It builds directly on those primitives.

1. The two big axes

Before picking a protocol, separate two independent decisions. People conflate them and pick badly.

1. **Communication style** — *synchronous* (caller blocks waiting for a reply) vs *asynchronous* (caller fires and continues; reply arrives later or never).
2. **Interaction shape** — *request/response* (RPC-like: “do X, give me Y”) vs *event-driven* (“X happened” broadcast to whoever cares).

	request/response	event-driven
sync	REST GET, gRPC unary, GraphQL query	(rare — blocking on an event is an anti-pattern)
async	202 + status URL, webhooks (callback)	message queues, Kafka, pub/sub, domain events

Synchronous coupling is the silent killer of distributed systems: if service A *blocks* on B, then B’s latency and downtime become A’s. Async (queues, events) decouples availability — A enqueues and returns even

if B is down. See [10-message-queues-and-streaming.md](#) and [15-microservices-and-decomposition.md](#).

Rule of thumb: Use sync request/response when the caller *needs the answer to proceed* (loading a product page). Use async/events when the work can happen out-of-band (send email, generate a PDF, rebuild a search index).

2. REST vs GraphQL vs gRPC vs tRPC

All four move structured data between processes; they differ in *who controls the shape*, the *wire format*, and the *coupling*.

Dimension	REST	GraphQL	gRPC	tRPC
Transport	HTTP/1.1 or 2, JSON	HTTP, JSON (single endpoint)	HTTP/2, Protobuf binary	HTTP, JSON
Schema/contract	OpenAPI (optional)	Strongly typed schema (SDL)	<code>.proto</code> (strict)	TS types (compile-time only)
Who picks fields	Server	Client (per-query)	Server	Server
Streaming	SSE / chunked (awkward)	Subscriptions	Native bidi streaming	Limited
Over-/under-fetching	Common	Solved by design	Minimal (binary)	Minimal
Caching	HTTP caching just works	Hard (POST, one URL)	App-level only	App-level
Cross-language	Excellent	Good	Excellent (codegen)	TS-only
Browser-friendly	Yes	Yes	No (needs grpc-web proxy)	Yes
When it wins	Public APIs, CRUD, broad reach, HTTP caching	Many clients with divergent needs (mobile vs web), aggregating many sources	Internal service-to-service, low latency, polyglot, streaming	Full-stack TS monorepo, no codegen overhead

REST is the default for public and CRUD-shaped APIs. It rides HTTP semantics (verbs, status codes, caching, conditional requests) so intermediaries — CDNs, proxies, browsers — understand it for free. Stripe, GitHub, and Twilio are REST(ish).

GraphQL shines when one backend serves many clients that each want different field subsets, and when a single screen aggregates several backend sources. The client sends a query; the server returns *exactly* those fields, eliminating the over-fetch (download fields you ignore) and under-fetch (N requests to assemble one screen) problems. Cost: caching is hard (everything is `POST /graphql`), and a naive resolver layer reintroduces N+1 queries (mitigated by DataLoader batching). GitHub's v4 API is GraphQL.

gRPC is the standard for *internal* east-west traffic. Protobuf is compact and fast; HTTP/2 gives multiplexing and bidirectional streaming; codegen produces typed stubs in every language. It's poor for browsers (needs a grpc-web proxy) and unfriendly to ad-hoc curl debugging.

tRPC removes the schema-duplication tax in a TypeScript monorepo: the client imports the server's *types* directly, so there's no codegen and no drift — but it only works when both ends are TS. For your Nuxt+Laravel stack, tRPC isn't applicable (Laravel isn't TS); REST is the right backbone.

3. REST done properly: resources, verbs, status codes

REST models **resources** (nouns) addressed by URLs; verbs express the operation. The classic mistake is putting verbs in the path (`POST /createOrder`) — that's RPC wearing a REST costume.

Verb	Meaning	Safe?	Idempotent?
GET	Read	✓	✓
POST	Create / non-idempotent action	✗	✗
PUT	Replace whole resource	✗	✓
PATCH	Partial update	✗	✗ (usually)
DELETE	Remove	✗	✓

Safe = no side effects. *Idempotent* = running it N times == running it once. These properties drive what's safe to retry (§6) and cache.

Resource modeling. Use plural collection nouns and nest sub-resources:

```
GET    /orders?status=open&page=2    # filtered collection
POST   /orders                      # create
GET    /orders/9f3a-uuid           # one resource (use uuid, not int id)
PATCH /orders/9f3a-uuid           # partial update
DELETE /orders/9f3a-uuid
GET    /orders/9f3a-uuid/line-items # sub-collection
POST   /orders/9f3a-uuid/refunds  # "action" modeled as a sub-resource
```

That last line is the trick for actions that don't fit CRUD: model the *outcome* as a resource (`POST /refunds`) rather than `POST /order/refund` .

Status codes — use the precise one. The category matters as much as the exact number:

- **2xx** success — `200 OK` , `201 Created` (+ `Location` header), `202 Accepted` (async, §9), `204 No Content` .
- **3xx** redirect/conditional — `304 Not Modified` (ETag hit).
- **4xx** *client's fault, don't retry as-is* — `400` (malformed), `401` (unauthenticated), `403` (authenticated but forbidden), `404` , `409` (conflict), `422` (validation failed), `429` (rate limited).

- `5xx` server's fault, retry may help — `500` , `503` (unavailable, often with `Retry-After`), `504` (gateway timeout).
- The 4xx/5xx split is a *contract with retry logic*: clients retry 5xx and 429, not 400/422. Returning `500` for a validation error makes clients hammer you pointlessly.

4. Versioning & schema evolution without breaking consumers

The cardinal rule (from [`CLAUDE.md` 's backward-compat section]): **never remove, rename, or retype a field that a live consumer reads**. Additive changes are safe; subtractive ones break.

Strategies:

Strategy	Example	Pros	Cons
URI versioning	<code>/v1/orders</code> , <code>/v2/orders</code>	Obvious, cacheable, easy to route	Version sprawl; URLs lie about “the” resource
Header versioning	<code>Accept: application/vnd.acme.v2+json</code>	Clean URLs	Invisible, harder to test in a browser
Content negotiation	<code>Accept</code> media type	RESTful purity	Same invisibility issue
No versioning + additive evolution	Add fields, never remove	No version explosion	Requires discipline + deprecation tooling

Stripe famously uses **date-based versions pinned per account** (`Stripe-Version: 2024-06-20`) and runs request/response *transformers* that translate between the account's pinned version and the current internal model — so old integrations never break while the core schema evolves. That's the gold standard: one internal model, compatibility shims at the edge.

Safe evolution checklist:

-  Add an optional field, add a new endpoint, add a new enum value *only if clients tolerate unknowns*.
-  Remove/rename a field, tighten validation, change a type, change default behavior of a `200` .
- For breaking changes: add the new field *alongside* the old, migrate every consumer (admin, storefront, designer, webhooks), then remove the old field in a later release.

Protobuf schema evolution (for gRPC/Kafka payloads): each field has a numeric *tag*. Rules — never reuse or change a tag number; new fields must be `optional` /have defaults; you can rename a field freely (the tag, not the name, is on the wire); to remove a field, `reserve` its tag so it's never reused. Readers ignore unknown tags (forward compatibility) and supply defaults for missing ones (backward compatibility). This is exactly the encoding-evolution material in DDIA Ch.4.

Client		Server (+ idempotency store)
POST /charges		
Idempotency-Key: abc-123	→	key abc-123 seen before?
		↳ no → run charge, store(abc-123, result), 201
← 201 Created		↳ yes → return stored result (no re-charge)
(timeout! client unsure)		
POST /charges (RETRY)		
Idempotency-Key: abc-123	→	key abc-123 seen → REPLAY stored 201
← 201 Created		(same body, no second charge)

Mechanics that matter at staff level:

- **Scope the key** to the operation + account, with a TTL (Stripe keeps keys ~24h).
- **Handle the concurrent retry** (client fires the retry before the first finishes): take a per-key lock or rely on a unique DB constraint on the key; return `409` or block until the first completes. Never run two bodies for the same key.
- **Fingerprint the body** — if the same key arrives with a *different* payload, reject (`422`), because that's a client bug, not a retry.
- The store *is* the idempotency guarantee; if it's flaky, your guarantee is flaky.

Stripe's `Idempotency-Key` header is the canonical reference. PayPal, Adyen, and Square do the same.

Deep dive: `11-distributed-transactions-and-idempotency.md`.

7. Rate-limit headers & error-contract design

Rate limiting protects you from abuse and noisy neighbors (deep dive: `16-rate-limiting-and-resiliency.md`). The *API-design* part is the response contract. On a limited request return `429 Too Many Requests` plus headers so a well-behaved client can self-throttle:

```
HTTP/1.1 429 Too Many Requests
RateLimit-Limit: 100
RateLimit-Remaining: 0
RateLimit-Reset: 30      # seconds until the window resets
Retry-After: 30
```

(GitHub uses `X-RateLimit-*`; the IETF is standardizing the un-prefixed `RateLimit-*` fields.) Always send `Retry-After` on `429` and `503` so clients back off deterministically instead of hammering.

Error contracts. Don't return bare strings or HTML. Standardize on `application/problem+json` (RFC 9457), a single machine-parseable shape across every endpoint:


```

HTTP/1.1 422 Unprocessable Entity
Content-Type: application/problem+json
{
  "type": "https://api.acme.com/errors/validation",
  "title": "Validation failed",
  "status": 422,
  "detail": "The 'email' field must be a valid email address.",
  "instance": "/orders/9f3a-uuid",
  "errors": { "email": ["must be a valid email"] }
}

```

A stable error contract lets clients branch on `type` (a URI, not a translated string), show `detail` to users, and field-map `errors`. Per `CLAUDE.md`'s UX rules, the *frontend* must turn this into plain language — never surface `"422"` or raw `type` URIs to a non-technical store owner.

8. BFF, API gateways, and contract testing

API gateway — a single front door for many backend services. It centralizes cross-cutting concerns so each service doesn't reimplement them: TLS termination, authn, rate limiting, routing, request logging, and response shaping. Kong, AWS API Gateway, and Envoy are typical. The risk is the gateway becoming a “god object” with business logic in it — keep it dumb (routing + policy), keep logic in services.

Backend-for-Frontend (BFF) — a *per-client* API layer. Your web app and mobile app have different needs (mobile wants fewer round-trips and smaller payloads; web can do more chatty calls). Instead of one bloated API serving both poorly, each frontend gets a thin BFF that aggregates and tailors. Your **storefront** (`frontstore/`) and **admin** (`nuxt/`) are effectively two BFF-shaped consumers of the Laravel API — and Nuxt's server routes (Nitro) can act as a true BFF, hiding internal endpoints and composing calls server-side.

```

mobile → Mobile BFF ─┐
                       └─▶ [order svc] [catalog svc] [pricing svc]
web      → Web BFF ───┘

```

Contract testing — the antidote to integration drift in microservices. Instead of slow end-to-end tests, the *consumer* declares the requests it makes and the responses it expects (a “pact”); the *provider* runs those expectations against itself in CI. If the provider removes a field the consumer relies on, the provider's build fails *before* deploy. Pact is the well-known tool. This catches the §10 silent-breaking-change class mechanically. See `14-testing-distributed-systems.md`.

9. Webhooks vs polling & long-running operations

Polling vs webhooks — two ways to learn “did X happen yet?”

	Polling	Webhooks
Direction	Client asks repeatedly	Server calls client (callback)
Latency	Up to poll interval	Near-real-time
Wasted requests	Many empty polls	None
Complexity on receiver	Trivial	Must host an endpoint, verify, dedupe
When	Simple, low-volume, no public URL	Event-driven, many consumers, low latency

Webhook flow and its hard parts:

```

Provider (Stripe)                                Your server
| event: payment.succeeded                          |
| POST /webhooks/stripe → | 1. verify HMAC signature (Stripe-Signature)
| Stripe-Signature: t=.,v1=<hmac> | 2. check event id seen? (dedupe – at-least-once!)
|                               | 3. enqueue job, return 200 FAST
|                               | 4. process async; non-200 ⇒ provider retries
| ← 200 OK (within seconds) ← |

```

Webhook delivery is **at-least-once**: providers retry until they get a `2xx`, so the *same* event can arrive twice — your handler must be **idempotent** (dedupe by event id). Always **verify the signature** (HMAC of the raw body with a shared secret) or anyone can POST fake events. Return `2xx` immediately and do work asynchronously, or slow processing triggers retries and duplicates. This is exactly the event-driven async style from §1.

Long-running operations (202 + status URL). When work can't finish within a request (PDF generation, bulk import, image processing — your `pdf-service`), don't hold the connection. Accept and hand back a status resource:

```

POST /exports
→ 202 Accepted
  Location: /exports/job-42
  { "id":"job-42", "status":"pending" }

GET /exports/job-42
→ 200 { "status":"processing", "progress":40 }
...
GET /exports/job-42
→ 200 { "status":"done", "result_url":"https://.../export.pdf" }

```

The client polls the status URL (or you fire a webhook on completion — best to offer both). `202` means “accepted, not done”; never return `200` with a fake “success” for work that's still queued.

10. Common pitfalls / war stories

- **The silent breaking change.** A backend renames `total` → `grand_total` in a `200` response. No error fires anywhere; the storefront just shows `$0` because `order.total` is now `undefined`. A

frontend showing wrong data is worse than one showing an error. This is the exact class `CLAUDE.md` calls out — and contract tests (§8) catch it mechanically. Always: add new field, migrate consumers, remove old later.

- **The silent dropped field.** A form sends `attention_to`; the backend's `FormRequest` `rules()` doesn't list it, so it's discarded. User sees "Saved", data is gone. Every input must round-trip `validate` → `save` → `read` → `render`, proven by a test. (Same `CLAUDE.md` "silent-lie" class.)
- **500 for validation.** Returning `5xx` on bad input makes clients retry forever and pollutes your error budget. Validation = `422`.
- **Offset pagination on a growing feed.** Users report "I keep seeing the same order twice." Cause: inserts shifting offsets mid-scroll. Switch to keyset.
- **Non-idempotent webhook handler.** Stripe retries `payment.succeeded`; your handler ships the order twice. Dedupe by event id.
- **GraphQL N+1.** Resolver fetches the author for each of 100 posts → 101 queries. Fix with `DataLoader` batching, not by abandoning GraphQL.
- **Verb-in-URL "REST".** `POST /getUser`, `POST /order/cancel` — you've built RPC over HTTP and lost caching, idempotency semantics, and intermediary understanding.

Case Study: Stripe API

Stripe's whole business is one promise: developers integrate the API *once* and it keeps working — through retries, network failures, and years of schema changes — without ever double-charging a customer. At Stripe's scale that promise is non-trivial. The API handles **billions of requests per day**, processes **hundreds of billions of dollars** in annual payment volume, and serves **millions of businesses** whose integrations were written across more than a decade. A single one of those integrations might have been deployed in 2015 and never touched since. It must still work today, and a charge that runs twice is real money lost. The problem statement is therefore: *every concept in this module, applied at once, with money on the line.*

Idempotency: the POST-retry problem, in production

The motivating failure is exactly §6's ambiguous timeout. A merchant's server sends `POST /v1/charges`, the connection drops *after* Stripe charged the card but *before* the `201` came back. The client has no idea whether it worked. Retry blindly and you double-charge; don't retry and you might lose a successful payment from your records.

Stripe's answer is the canonical **idempotency key** pattern from this module. The client generates a UUID and sends `Idempotency-Key: <uuid>`. Stripe stores `(key → first response)` and **replays** the stored response for any retry carrying the same key — N requests, one charge.

Merchant server		Stripe API + idempotency store
POST /v1/charges		
Idempotency-Key: a1b2	→	seen a1b2?
		↳ no → charge card, store(a1b2, 201+body)
← 201 {id: ch_x}	←	
x TCP reset – unsure!		
POST /v1/charges (retry)		
Idempotency-Key: a1b2	→	seen a1b2 → REPLAY stored 201 (no 2nd charge)
← 201 {id: ch_x}	←	

Stripe implements every staff-level mechanic §6 lists: keys are **scoped per account** with a **~24h TTL**; concurrent retries (the retry fires before the first finishes) are serialized by a **per-key lock** that returns a **409** rather than running two bodies; and Stripe **fingerprints the request body** — reuse a key with a *different* payload and you get an error, because that's a client bug, not a retry. The idempotency store is the guarantee, so it's backed by their durable datastore, not an in-memory cache.

Dated versioning: backward-compat without version sprawl

This is §4's gold-standard example made concrete. Stripe does **not** use `/v1/`, `/v2/` URL sprawl. The path stays `/v1/` forever; the real version is a **date pinned per account** — `Stripe-Version: 2024-06-20`. When an account first calls the API its version is frozen to that day's schema. Internally Stripe keeps **one current model** and a chain of **request/response transformers**: an integration pinned to 2018 hits the modern code, and the response is run *backwards* through every transformer between today and 2018 before it ships. A 2015 integration literally receives 2015-shaped JSON from 2026 code. This is precisely the “one internal model, compatibility shims at the edge” pattern §4 describes — additive-only core, edge translation for olds clients — and it's why Stripe almost never forces a migration.

Cursor pagination & the error contract

Listing objects (`GET /v1/charges`) uses **cursor pagination** (§5), not offset — `?starting_after=ch_abc&limit=100`. With merchants holding millions of charges, `OFFSET 2000000` would scan-and-discard millions of rows per page; the cursor is an opaque object id that anchors a keyset seek, so deep pages cost the same as shallow ones. Errors follow a **stable, typed contract** (§7): every error is a JSON object with a machine-branchable `type` (`card_error`, `invalid_request_error`, `rate_limit_error`), a `code`, and a human `message`, plus **429** responses that carry backoff guidance — exactly the “branch on a stable field, never parse the prose” discipline from the error-contract section.

The trade-off they accepted

Dated versioning is **expensive to operate**. Every breaking schema change requires writing — and *keeping forever* — a transformer that downgrades the new shape to every prior version. Years in, that's a long, permanently-maintained, well-tested chain of shims that every response passes through, adding latency and engineering toil. Stripe deliberately took on **enormous, perpetual internal complexity** to buy **zero forced migrations for customers**. For most APIs that trade is wrong (just cut a `/v2/`); for a payments API whose moat is “integrate once, trust forever,” reducing customer-facing breakage to near-zero is worth any amount of internal plumbing. Same logic on idempotency: they accept the cost of a

durable, locked, body-fingerprinted key store on the hot path of every mutating request to guarantee no double-charge ever.

Lessons

- **Make the unsafe operation safe to retry, don't tell clients "don't retry."** Networks are ambiguous by nature; the idempotency-key + server-side replay pattern turns a dangerous `POST` into something a client can retry without thinking — which is the only retry policy clients actually follow.
- **Pay the versioning cost at the edge, once, so consumers never pay it.** One internal model plus per-version transformers beats URL version sprawl when your value proposition is long-lived integrations — but it's a real, permanent operational tax, so only spend it where backward-compat is the product.
- **A typed error/version/cursor contract is a promise to machines, not humans.** Clients branch on `type`, pin a `Stripe-Version`, and pass opaque cursors back verbatim — every one of those is a stable surface you can evolve behind without breaking the consumer.

11. Test yourself

1. A `POST /charges` times out. Is it safe to retry blindly? What mechanism makes it safe? (*Hint: ambiguous outcome; idempotency key + server-side replay.*)
 2. Why does `LIMIT 20 OFFSET 1000000` get slow, and what fixes it? (*Hint: DB scans+discards a million rows; keyset on an indexed sort key.*)
 3. You must rename a JSON field consumed by three frontends. What's the safe sequence? (*Hint: add new alongside old → migrate all consumers → remove old later; never rename in place.*)
 4. When does GraphQL beat REST, and what does it cost you? (*Hint: many clients with divergent field needs / aggregation; cost = HTTP caching, resolver N+1.*)
 5. Webhooks are delivered at-least-once. What two properties must your receiver have? (*Hint: idempotent dedupe by event id; signature verification.*)
 6. Which status code for: bad input, not logged in, logged in but not allowed, too many requests, async accepted-but-not-done? (*Hint: 422, 401, 403, 429, 202.*)
 7. In a Protobuf schema, why must you never reuse a field's tag number, and how do you safely remove a field? (*Hint: tag is the wire identity; `reserve` the tag.*)
 8. Why is synchronous service-to-service coupling dangerous, and what reduces it? (*Hint: B's downtime/latency becomes A's; async queues/events decouple availability.*)
-

12. Further reading

- **Stripe API docs** — idempotency keys, date-based versioning, cursor pagination, error objects. The canonical real-world reference.
- **Roy Fielding's dissertation, Ch. 5** — the original definition of REST and its constraints (read it once to see how much "REST" in the wild isn't).

- **RFC 9457** — Problem Details for HTTP APIs ([application/problem+json](#)).
- **gRPC docs + Protocol Buffers language guide** — service definitions, streaming, schema evolution rules.
- **DDIA (Kleppmann), Ch. 4 — Encoding and Evolution** — backward/forward compatibility, Protobuf/Avro/Thrift, the definitive treatment of schema evolution.
- **GitHub & Slack API docs** — production cursor pagination and rate-limit header conventions.
- **Pact docs** — consumer-driven contract testing.

Next: [04-...](#) — how the resources you just modeled are actually stored, indexed, and queried.

Database Internals: Storage Engines, Indexes, Transactions

Source: [readme/research/system-design/04-databases-internals.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains how databases physically store data, why some databases are better for reading versus writing, how indexes speed up (and sometimes slow down) queries, and how databases keep data safe and consistent when many users work at the same time.

Understanding this helps you pick the right tool, spot why a query gets slow, and avoid silent data bugs.

The main parts explained simply:

- **Pages (how data lives on disk)** — Databases read and write data in fixed-size chunks called pages (usually 8–16 KB), not one row at a time. Reading in order is far faster than jumping around the disk, and this single fact shapes every design decision below.
- **B-tree vs LSM-tree (two storage engine families)** — B-trees (PostgreSQL, MySQL) are great for mixed reads and writes; they update data in place. LSM-trees (Cassandra, RocksDB) suit constant heavy writes; they only append, making writes very fast but reads slightly slower.
- **Indexes** — Like a book index, they let the database find rows without scanning everything. But every index also slows down inserts and updates, since the database must update each index too. Add them where they speed up reads; remove ones nobody uses.
- **Write-ahead log (WAL)** — Before changing any data, the database writes a note about the change first. If power fails, it replays that note on restart to finish or undo the change safely. This is how committed data survives crashes.
- **Buffer pool (page cache)** — Databases keep hot data in RAM. If the data you use most fits in RAM, queries are fast. If not, every miss reads from disk, which is much slower.
- **ACID** — Four guarantees every reliable database makes: changes are all-or-nothing, they respect declared rules, multiple users don't corrupt each other's work, and committed data survives a crash.
- **Isolation levels and concurrency bugs** — When two users change data at the same time, subtle bugs can occur (one reads outdated data, two overwrite each other silently). Higher isolation levels block more of these bugs but cost more speed. Serializable isolation is safest for money and inventory.
- **MVCC (Multi-Version Concurrency Control)** — The database keeps multiple versions of a row so readers never block writers. Old versions pile up and need regular cleanup (called VACUUM in PostgreSQL); skipping it causes bloat and gradual slowdowns.
- **Query planner and EXPLAIN** — The database picks a plan for how to run each query. The EXPLAIN command shows you that plan. A “Seq Scan” on a large table usually means a missing index.
- **Discord case study** — Discord grew from millions to trillions of messages and migrated from MongoDB to Cassandra, then to ScyllaDB. Their write-heavy, append-mostly workload was a perfect fit for LSM-trees. Key lesson: group data by how you read it, keep those groups bounded in size, and solve extreme hot-spot problems above the database layer.

What to do with this: When a query gets slow as data grows, run EXPLAIN and look for a missing index first. For high-write features like logs, events, or chat, consider an LSM-based database over a

traditional SQL one.

What you'll learn. How databases physically store rows on disk, the deep trade-off between the two dominant storage-engine families (B-trees vs LSM-trees), how indexes actually accelerate (and sometimes sabotage) queries, and how transactions stay correct under concurrency via ACID, isolation levels, MVCC, and the write-ahead log. By the end you should be able to read an **EXPLAIN**, predict why a query is slow, reason about a concurrency anomaly, and pick the right engine for a workload.

Prerequisites. Read `01-fundamentals.md` first — it establishes latency numbers (memory vs SSD vs disk vs network), throughput-vs-latency thinking, and the back-of-envelope habits this module assumes. Familiarity with SQL helps but isn't required.

1. How a database stores data on disk

Strip away SQL and a database is a thing that turns logical records into bytes on a block device and back, *durably* and *fast*. Two physical facts dominate every design decision:

1. **Disks are block devices.** You cannot read or write one byte; you read/write a *page* (typically 4 KB, 8 KB, or 16 KB). Postgres pages are 8 KB; InnoDB pages are 16 KB. The page is the atomic unit of I/O and caching.
2. **Random I/O is far slower than sequential I/O.** On spinning disks the gap is $\sim 100\times$; even on NVMe SSDs, sequential writes are friendlier (less write amplification, better wear). This single fact is *why* LSM-trees exist.

The naive design — a flat append-only file of records — gives $O(1)$ writes but $O(n)$ lookups. To get fast lookups you need an **index**: a separate data structure that maps a key to a location. The structure you pick *is* the storage engine's identity.

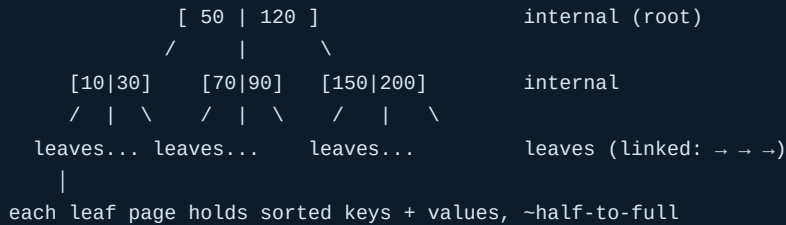
```
Logical row: (id=42, name='Acme', plan='pro')
              |  serialize
              ▼
Page (8KB):  [ header | ...row42... | ...row17... | free → |
              |
              | one unit of read / write / cache
```

A **heap** (Postgres) stores rows in no particular order — insertion order with reuse of freed space. A **clustered index** (InnoDB) stores the rows *inside* the index leaves, physically ordered by primary key. That difference cascades through everything below.

2. B-tree vs LSM-tree: the two great families

2.1 B-trees (the default: Postgres, InnoDB, SQLite, Oracle)

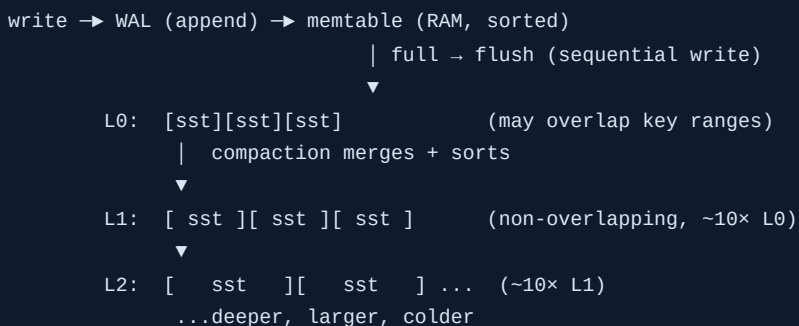
A **B+tree** is a balanced, high-fan-out search tree. Internal nodes hold only keys + child pointers; leaves hold the data (or pointers to it) and are linked for range scans. Fan-out is huge (hundreds of keys per page), so depth is tiny: a few billion rows fit in 3–4 levels.



- **Lookup:** $O(\log n)$ — descend from root, ~3–4 page reads (mostly cached).
- **Range scan:** find start leaf, walk the linked list. Excellent.
- **Write:** find the leaf, write *in place*. If the leaf is full, **split** it (and possibly propagate up). Writes are **random** (the target page can be anywhere) and **mutate existing pages**.
- **Durability hazard:** a page split can touch multiple pages; a crash mid-split corrupts the tree. Fixed by the WAL (§4) and, in Postgres, *full-page writes* to survive torn pages.

2.2 LSM-trees (RocksDB, LevelDB, Cassandra, ScyllaDB, HBase, modern MyRocks)

A **Log-Structured Merge tree** never updates in place. Writes go to an in-memory sorted structure (the **memtable**, often a skip list) plus an append-only WAL. When the memtable fills, it's flushed *sequentially* to an immutable on-disk file called an **SSTable** (Sorted String Table). Background **compaction** merges SSTables, dropping superseded/deleted keys.



- **Write:** append to WAL + insert into memtable. **Sequential, fast, no in-place mutation.** This is the headline win.
- **Read:** check memtable, then L0 SSTables, then deeper levels — a key may live anywhere. Mitigated by per-SSTable **Bloom filters** (skip files that definitely lack the key) and block caches. Worst case touches many files.
- **Delete:** write a **tombstone** marker; the real removal happens at compaction. Range deletes + slow compaction = “tombstone hell” (reads scan dead keys).

- **Space:** old versions linger until compacted.

2.3 The three amplifications (memorize these)

Amplification	Means	B-tree	LSM-tree
Write	bytes written to disk ÷ bytes of logical data	Lowish but page-granular; ~2–3× (WAL + page)	High: each key rewritten on every compaction level; 10–30× typical
Read	I/Os per logical read	Low & predictable (~tree depth)	Higher & variable (memtable + N levels), Bloom-filtered down
Space	disk used ÷ logical data	Internal fragmentation; ~1.3–2× (page half-empty after splits)	Better with leveled compaction (~1.1×); worse with size-tiered

2.4 When each wins

Choose B-tree when...	Choose LSM when...
Read-heavy or balanced workloads	Write-heavy / high ingest (logs, metrics, events, time series)
Point lookups + range scans both matter	Sequential write throughput is the bottleneck
You want predictable, low read latency	You can trade read tail-latency for write throughput
In-place updates of small fields	Append-mostly, few updates, key-value access
Examples: Postgres, MySQL/InnoDB, SQLite	Examples: Cassandra, RocksDB, ScyllaDB, HBase, MyRocks

Real engine choices. Postgres = heap + B-tree secondary indexes + MVCC. MySQL InnoDB = clustered B+tree (rows live in the PK index). Cassandra/Scylla = LSM (write-optimized, eventually-consistent ring — see [08-replication-and-partitioning.md](#)). RocksDB is the embeddable LSM that powers Kafka Streams state stores, CockroachDB, TiKV, and MyRocks.

3. How indexes actually work

An index is just a *second* data structure (usually a B-tree) ordered by some key, whose leaves point back to the row.

3.1 Clustered/primary vs secondary

- **Clustered (InnoDB):** the primary key B-tree *is* the table — leaves contain the full row. There is exactly one (the table's physical order). A secondary index leaf stores the **indexed columns + the primary key value**, not a physical pointer. So a secondary-index lookup that needs other columns does *two* B-tree descents: secondary → PK → row. (This is why a huge/random PK like a UUIDv4 hurts InnoDB: it bloats every secondary index and randomizes inserts. Prefer sequential keys or UUIDv7.)

- **Heap + secondary (Postgres):** rows live in the heap; every index (including the PK) is secondary and stores a `ctid` (page, offset) pointer into the heap. No “clustered” table unless you run `CLUSTER` (a one-shot reorder, not maintained).

3.2 Covering index & index-only scans

If an index contains *every column a query needs*, the engine never touches the table — an **index-only scan** (Postgres) / **covering index** (InnoDB). Hugely faster for hot read paths.

```
-- Query: SELECT plan FROM stores WHERE tenant_id = ?;
CREATE INDEX ix_stores_tenant_plan ON stores (tenant_id) INCLUDE (plan);
-- 'plan' rides in the leaf → index-only scan, no heap fetch.
```

(In Postgres, index-only scans also require the page’s visibility map bit to be set — i.e. recently `VACUUM` ed — or it still visits the heap to check tuple visibility. §6.)

3.3 Composite indexes and the leftmost-prefix rule

A composite index on `(a, b, c)` is sorted by `a`, then `b`, then `c` — like a phone book by (last, first). It can serve:

- `WHERE a = ?` ✓
- `WHERE a = ? AND b = ?` ✓
- `WHERE a = ? AND b = ? AND c = ?` ✓ (and an `ORDER BY` matching the order)
- `WHERE a = ? AND c = ?` ⚠ uses only the `a` prefix, then filters `c`
- `WHERE b = ?` ✗ **cannot use the index** — no leftmost prefix.

A range on a column *stops* prefix usage for everything after it: `WHERE a = ? AND b > ? AND c = ?` uses `a, b` for seeking but `c` cannot be used for the seek. Order columns: **equality first, then the range/sort column last**. See `05-data-modeling-sql-nosql.md` for designing these around access patterns.

3.4 Why an index can hurt

Every index is a copy that must be kept in sync. An `INSERT` / `UPDATE` / `DELETE` must update **every** affected index — extra random writes, more WAL, more page splits, more space. Ten indexes on a hot write table can make writes several times slower. Indexes are a **read tax paid on writes**: add them deliberately, drop unused ones (`pg_stat_user_indexes.idx_scan = 0` ⇒ suspect).

4. The write-ahead log (WAL) and durability

Rule: log the intent before mutating the data. Before any change touches a data page, the engine appends a record describing the change to a sequential **WAL** (Postgres WAL / InnoDB redo log) and `fsync` s it. Only then can the change be applied to in-memory pages and lazily flushed.

Why it works:

- The WAL write is **sequential** (fast) and small; the data-page writes can be **deferred and batched**.
- On crash, **replay** the WAL from the last checkpoint: redo committed changes, undo uncommitted ones. The database is restored to a consistent state.
- A **commit** is durable the instant its WAL record is **fsync** ed — *not* when data pages hit disk. This decouples durability from random data-page I/O. (`synchronous_commit = off` trades a few ms of possible loss for throughput — a knob, not a default.)

```
T writes row → [WAL: "set page P, off 0, val V"] fsync ✓ → commit returns
                        | later, async
                        ▼
                    dirty page P flushed at checkpoint
crash → reread WAL from last checkpoint → redo/undo → consistent
```

The WAL is also the backbone of **replication** (ship WAL to replicas) and **point-in-time recovery** — see [08-replication-and-partitioning.md](#).

5. The page cache / buffer pool

Databases keep hot pages in RAM: InnoDB's **buffer pool**, Postgres's **shared_buffers** (plus the OS page cache underneath). Reads check the pool first; misses fault a page in from disk. Dirty pages accumulate and are flushed at **checkpoints**. Sizing this is the single biggest perf lever for a B-tree DB: if the working set fits in RAM, lookups are CPU-bound tree walks; if not, you pay SSD/disk latency on misses (recall the numbers from [01](#)). LSM engines lean on **block caches** + Bloom filters instead, since their files are immutable.

6. ACID, precisely

- **Atomicity** — a transaction is all-or-nothing. Implemented via undo info (Postgres: old tuple versions; InnoDB: undo log) + WAL.
- **Consistency** — it moves the DB from one valid state to another *per declared constraints* (FKs, CHECKs, uniqueness). This is partly the app's job; the DB enforces declared rules.
- **Isolation** — concurrent transactions don't corrupt each other's view. The slippery one; §7–8.
- **Durability** — once committed, survives a crash. The WAL (§4).

"Consistency" here is **not** the "C" in CAP — that's a different word for replica agreement. Don't conflate them; see [09-cap-pacelc-consistency-models.md](#).

7. Isolation levels and concurrency anomalies

The SQL standard defines four isolation levels by which anomalies they forbid. The anomalies:

- **Dirty read** — T2 reads a row T1 wrote but hasn't committed; T1 rolls back ⇒ T2 read a value that never existed.
- **Non-repeatable read** — T1 reads a row, T2 commits an update to it, T1 reads again and gets a *different value* within the same transaction.
- **Phantom read** — T1 runs `SELECT ... WHERE status='open'` (5 rows), T2 inserts a matching row + commits, T1 reruns and now sees 6 — a *new row* appears in a range.
- **Lost update** — T1 and T2 both read `count=10`, both write `11`; one update is silently lost. Classic with read-modify-write.
- **Write skew** — T1 and T2 read an overlapping set, each checks an invariant that still holds, each writes a *different* row; together they break the invariant. Example: two doctors each check “≥1 doctor on call” (true), each books themselves off — now zero on call. Snapshot isolation does **not** prevent this; only serializable does.

Isolation level	Dirty read	Non-repeatable	Phantom	Lost update	Write skew	When to use
Read Uncommitted	✓ allowed	✓	✓	✓	✓	Almost never (Postgres treats it as Read Committed anyway)
Read Committed	✗ prevented	✓ allowed	✓ allowed	✓ allowed	✓ allowed	Default in Postgres/Oracle; OLTP where each statement sees fresh committed data
Repeatable Read	✗	✗	✗ in PG (snapshot)*	✗ in PG	✓ allowed	Reports/multi-read consistency; PG calls snapshot isolation this
Serializable	✗	✗	✗	✗	✗ prevented	Money, inventory, invariants across rows; correctness over throughput

* The standard *allows* phantoms at Repeatable Read; **Postgres's Repeatable Read is full snapshot isolation and blocks phantoms** but still permits write skew. MySQL InnoDB's Repeatable Read uses next-key (gap) locks to block phantoms too. **Know your engine — “Repeatable Read” means different things.**

Postgres **Serializable Snapshot Isolation (SSI)** runs at snapshot-isolation speed but *monitors* for dangerous read-write dependency cycles and **aborts** one transaction with a serialization failure (`40001`) — so your app must **retry** on `40001`. This is optimistic serializability, not heavy locking.

8. MVCC vs locking; optimistic vs pessimistic

8.1 MVCC (Multi-Version Concurrency Control) — “readers don’t block writers”

Instead of locking rows for reads, the DB keeps **multiple versions** of each row and shows each transaction a consistent *snapshot*. In Postgres, every row (tuple) carries hidden system columns `xmin` (the txid that created it) and `xmax` (the txid that deleted/superseded it). An `UPDATE` does **not** overwrite — it writes a **new tuple** and marks the old one’s `xmax`.

```
tuple history for store id=42:
```

```
v1 xmin=100 xmax=105  plan='free'  ← visible to snapshots < 105
v2 xmin=105 xmax=0     plan='pro'   ← current, visible to snapshots ≥ 105
```

▲

```
each transaction's snapshot decides which version it sees,
by comparing xmin/xmax against the set of committed txids
```

Consequences:

- Reads never block writes and vice versa — huge concurrency win.
- **Dead tuples accumulate.** Old versions become invisible to all live transactions but still occupy pages = **bloat**. `VACUUM` reclaims them (autovacuum runs continuously); `VACUUM FULL` rewrites the table (takes a lock). Without vacuuming, tables and indexes bloat and slow down.
- **Transaction ID wraparound:** txids are 32-bit; Postgres must “freeze” old tuples before the counter wraps. A neglected DB can hit emergency anti-wraparound vacuum (or, historically, refuse writes). Monitor `age(datfrozenxid)`.

InnoDB also does MVCC, but differently: it **updates the row in place** in the clustered index and keeps old versions in the **undo log**; readers reconstruct an old snapshot by walking undo records. So InnoDB has undo bloat, not heap dead-tuple bloat, and “purge” threads clean undo. (See [05-data-modeling-sql-nosql.md](#) for how this shapes schema choices.)

8.2 Pessimistic vs optimistic concurrency control

- **Pessimistic:** take locks up front and hold them (`SELECT ... FOR UPDATE` , row/gap locks). Safe under contention; risks **deadlocks** (the DB detects a cycle and aborts a victim) and reduced concurrency. Use when conflicts are likely (decrementing the same inventory row repeatedly).
- **Optimistic:** don’t lock; assume no conflict, validate at commit (or via a `version` column / `WHERE updated_at = ?`), and **retry** on conflict. Cheap when conflicts are rare; wasteful (rollbacks) when they’re common. Postgres SSI is optimistic; app-level “version column then `UPDATE ... WHERE version = N` ” is the portable pattern. *This is the correct fix for the lost-update bug* — a bare read-modify-write in app code is not.

9. The query planner / optimizer and reading EXPLAIN

SQL is *declarative*: you state *what*, the **planner** decides *how*. It enumerates execution plans (which index, join order, join algorithm) and picks the **cheapest** by a cost model fed by **statistics** (`ANALYZE` gathers row counts, distinct values, histograms, most-common-values).

Key plan nodes:

- **Seq Scan** — read the whole table. Fine for small/most-of-table queries; a red flag on a large table with a selective filter (⇒ missing index).
- **Index Scan** — descend an index, then fetch matching heap rows.
- **Index Only Scan** — index covers all needed columns; no heap fetch (§3.2).
- **Bitmap Heap Scan** — gather many matches via index into a bitmap, then read heap pages in physical order (good for medium selectivity).
- **Nested Loop / Hash Join / Merge Join** — three join strategies; the planner picks by sizes and available indexes.

```
EXPLAIN ANALYZE
SELECT * FROM orders WHERE tenant_id = 7 AND status = 'paid';

Index Scan using ix_orders_tenant_status on orders
  (cost=0.43..812.5 rows=120 width=240)
  (actual time=0.05..1.2 rows=118 loops=1)
    Index Cond: (tenant_id = 7 AND status = 'paid')
Planning Time: 0.2 ms
Execution Time: 1.4 ms
```

How to read it: compare **estimated rows** vs **actual rows** — a big gap means stale statistics (run `ANALYZE`) and probably a bad plan. Watch for **Seq Scan** on big tables, huge **loops=** on a Nested Loop (an N+1 inside the DB), and **Rows Removed by Filter** (the index brought too much; tighten it). `EXPLAIN (ANALYZE, BUFFERS)` adds page-hit/read counts so you see cache behavior.

10. Common pitfalls / war stories

- **The missing index.** A `WHERE email = ?` with no index = Seq Scan; fine at 10k rows, a meltdown at 10M. Symptom: a query that “got slow as we grew.” Fix: index the predicate; verify with `EXPLAIN`.
- **The long-running transaction that bloats MVCC.** An idle-in-transaction connection (a forgotten `BEGIN`, a slow report, an ORM that left a txn open) holds back the MVCC horizon: autovacuum *cannot* remove dead tuples newer than the oldest live snapshot. Tables bloat, queries crawl, disk fills. Watch `pg_stat_activity` for `idle in transaction`; set `idle_in_transaction_session_timeout`.
- **SELECT N+1.** ORM loads 100 parents, then fires 100 child queries in a loop. 101 round-trips where 1–2 would do. (This codebase's `CLAUDE.md` calls it out: eager-load with `with(...)`.) The DB is fine each time; the *network round-trips* kill you. Fix: eager-load / join / batch.

- ## 🧩 Case Study: Discord's message store (MongoDB → Cassandra → ScyllaDB)

By 2015 Discord stored messages in a single MongoDB replica set and crossed **100 million messages**. The working set no longer fit in RAM, so reads fell off the cliff §5 describes: every lookup that missed the buffer pool paid SSD latency, and p99s became unpredictable. By 2017 they were at **billions of messages**; by 2022, **trillions** — peaking past **a million messages written per second** at busy hours. This is the textbook signal from §2.4: a write-heavy, append-mostly, time-ordered workload where sequential write throughput, not in-place updates, is the bottleneck. A B-tree's random in-place writes (§2.1) were exactly the wrong tool. They moved to **Cassandra** — an LSM-tree engine (§2.2).

Designing the partition + clustering key. Discord's dominant access pattern is “give me the most recent messages in a channel, scrollable backwards.” In Cassandra/Scylla the primary key is `(partition_key, clustering_key...)` : the partition key picks which node owns the data, and the clustering key sets the sorted order *within* that partition on disk — exactly the leftmost-prefix, equality-then-range idea from §3.3, but applied to an LSM SSTable layout. A naive `PARTITION KEY = channel_id` would put an entire channel's lifetime in one partition. Megachannels would grow unbounded — an LSM partition that never stops accumulating SSTables, so reads have to merge across more and more files (§2.2 “a key may live anywhere”). Their fix was a **composite, bucketed partition key**:

```
PRIMARY KEY ((channel_id, bucket), message_id)
            |____ partition ____|   | clustering |
```

bucket = fixed time window (≈ 10 days) derived from the message's Snowflake ID
message_id = Snowflake $\rightarrow$ time-sortable, so rows arrive in clustering order
(sequential appends to the memtable, no re-sort)

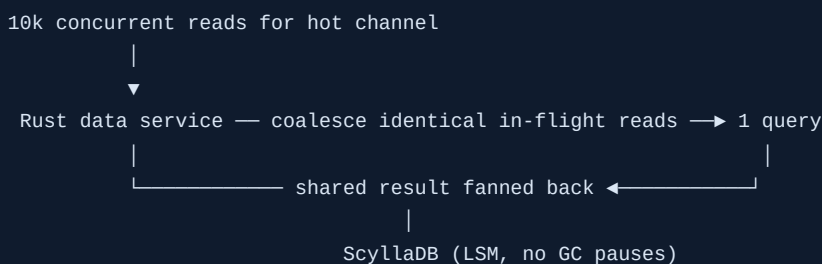
Read "latest in channel":
pick newest bucket $\rightarrow$ one partition $\rightarrow$ SSTables already sorted by message_id
 $\rightarrow$ take tail rows (range scan §3.3)

Bucketing bounds each partition's size, so compaction stays cheap and a channel read touches a handful of SSTables, Bloom-filtered (§2.2) down to the few that matter.

The hot-partition pain. Even bucketed, one partition lives on one set of replicas. When a celebrity or `@everyone` announcement made a single channel red-hot, all reads and writes for that `(channel_id, bucket)` slammed the same nodes — the **hot-partition** problem flagged in §2.2/§10. Cassandra made it worse for two reasons rooted in this module:

1. **Read latency from LSM read amplification (§2.3).** Fan-out reads on a hot partition had to check the memtable plus many SSTable levels; Bloom filters help, but the tail still touched several files, and Discord saw p99s blow out into **hundreds of milliseconds**, occasionally seconds.
2. **Tombstones and compaction storms (§2.2, §10).** Deletes write tombstones, not real removals. Heavy edit/delete traffic plus Cassandra's JVM **garbage-collection** pauses meant compaction couldn't keep up; reads scanned dead keys, and GC stalls added latency spikes on top.

The ScyllaDB migration. ScyllaDB is wire-compatible with Cassandra and keeps the *same* LSM model, partition/clustering keys, and data layout — so none of the §3.3 key design changed. What changed is the runtime: C++ instead of the JVM (no GC pauses), a shard-per-core architecture, and a tighter compaction scheduler. Discord also put a **Rust “data services” layer** in front of the store that does **request coalescing**: if 10,000 users request the same hot channel page simultaneously, it issues **one** database query and fans the result back out — directly attacking the hot-partition read storm above the storage engine.



The trade-off they accepted. They stayed on an LSM engine and *kept paying read amplification and eventual-consistency semantics* (no cross-partition transactions, no serializable invariants §7) in exchange for write throughput and horizontal scale. They explicitly chose **denormalization + bounded partitions over a normalized, transactional B-tree store** — the §2.4 trade in its purest form. They also accepted operational complexity (a bespoke Rust coalescing tier) rather than asking the database to absorb hot-key load it fundamentally can't.

Results. After the ScyllaDB move, Discord reported the cluster shrinking from **177 Cassandra nodes to 72 ScyllaDB nodes** while holding more data, p99 read latencies dropping from the hundreds-of-milliseconds range to roughly **15 ms (reads) / 5 ms (writes)**, and the GC-pause tail latency disappearing entirely. The migration backfilled **trillions of rows** in about nine days using a fleet of Rust migrator workers.

Lessons

- **The partition/clustering key is the whole design in an LSM store.** Pick it from your dominant access pattern (§3.3) and *bound partition size* — an unbounded partition is a latent hot-partition and compaction time bomb. Bucketing a time-ordered key is the canonical fix.
- **LSM buys write throughput and pays in read amplification and tombstones (§2.3).** Know which side of the §2.4 trade your workload is on before you migrate; Discord's write-heavy, append-mostly traffic was textbook LSM.
- **Some load can't be fixed in the storage layer.** A single hot partition lives on one replica set; solve it *above* the database with caching and request coalescing.
- **Same data model, different runtime can be the win.** Cassandra → ScyllaDB kept the LSM design intact and removed JVM GC pauses — proof that tail latency is often an engine/runtime problem, not a data-model one.

11. Test yourself

1. Why do LSM-trees achieve higher *write* throughput than B-trees, and what do they pay for it on reads? (*Hint: sequential appends + immutable SSTables vs in-place random page writes; reads must check memtable + multiple levels, mitigated by Bloom filters.*)
 2. Given an index on `(tenant_id, status, created_at)`, which of these can seek the index: `WHERE status='paid'` ? `WHERE tenant_id=7 AND created_at > ?` ? `WHERE tenant_id=7 AND status='paid'` ? (*Hint: leftmost-prefix; the second uses only the `tenant_id` prefix for seeking.*)
 3. A `COMMIT` returned successfully but the modified data pages were not yet on disk when the server lost power. Is the data safe? Why? (*Hint: WAL fsync at commit + crash recovery redo.*)
 4. Postgres Repeatable Read prevents phantoms but allows write skew. Explain the difference with the on-call-doctors example. (*Hint: snapshot consistency vs cross-row invariant; only Serializable/SSI catches the rw-dependency cycle.*)
 5. Why does an idle-in-transaction connection make *unrelated* tables slow? (*Hint: it pins the oldest snapshot; autovacuum can't reclaim dead tuples ⇒ bloat.*)
 6. When would you prefer optimistic concurrency over `SELECT ... FOR UPDATE` ? (*Hint: low conflict rate; cost of occasional retry < cost of holding locks; portability.*)
 7. `EXPLAIN ANALYZE` shows `rows=5` estimated but `actual rows=50000` on a Nested Loop. What's wrong and what do you do? (*Hint: stale stats ⇒ `ANALYZE` ; the underestimate likely picked a bad join; maybe needs a hash join / better index.*)
 8. Why does adding a covering/ `INCLUDE` index speed reads but slow writes? (*Hint: index-only scan avoids heap fetch; every write must also maintain the larger index.*)
-

12. Further reading

- **Designing Data-Intensive Applications** (Kleppmann) — **Ch. 3** (Storage and Retrieval: B-trees vs LSM, indexes) and **Ch. 7** (Transactions: isolation levels, anomalies, SSI). The single best companion to this module.
- **Database Internals** (Alex Petrov) — Part I (storage engines, B-trees, LSM, page layout, WAL) in depth.
- **PostgreSQL documentation** — *Transaction Isolation, MVCC, Routine Vacuuming, Using EXPLAIN, Index Types*. Authoritative and precise.
- **MySQL Reference Manual** — *InnoDB Locking and Transaction Model* (clustered index, gap/next-key locks, undo, MVCC).
- **RocksDB wiki** — compaction styles (leveled vs universal), Bloom filters, write/space amplification trade-offs.
- Then continue: [05-data-modeling-sql-nosql.md](#) , [08-replication-and-partitioning.md](#) , [09-cap-pace1c-consistency-models.md](#) .

05 — Data Modeling: SQL vs NoSQL & Polyglot Persistence

Source: [readme/research/system-design/05-data-modeling-sql-nosql.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains how to decide where and how to store data in a software system — and when a traditional database is the right choice versus a modern “NoSQL” database. Getting this wrong early leads to slow apps, expensive fixes, and systems that break under heavy traffic. Getting it right means your app stays fast no matter how many customers show up.

The main parts explained simply:

- **Traditional (SQL) databases — the meticulous library** — Store each piece of information in exactly one place and look it up by joining different tables together. Great for most apps. A well-run Postgres database handles millions of records without any problem. Only look elsewhere when you’ve truly outgrown it.
- **Normalization — no repeated facts** — A design rule that says each piece of data (like a customer’s name) lives in exactly one place. When the customer updates their name, only one record changes. The flip side is you sometimes need to combine multiple tables to answer a question.
- **Why big databases can’t always join tables** — When data is split across many machines (to handle huge traffic), combining tables requires moving data across the network, which is very slow. This is the core reason companies move away from joins at massive scale.
- **Four types of NoSQL databases** — Each is built for a different job: key-value stores (fast single-item lookup, like a dictionary), document stores (store a whole related group — like an order plus all its items — as one record), wide-column stores (fast for time-series or sensor data), and graph databases (mapping connections between things, like a social network).
- **Design around your questions, not your data** — NoSQL requires you to decide upfront exactly what questions you’ll ask the database. If you know “I’ll always look up orders by customer,” build the storage around that. If you don’t know your questions yet, a traditional database is safer.
- **Partition keys and sort keys — the most important NoSQL choice** — The “partition key” decides which machine stores your data; pick one with many distinct values (like user ID, not order status) so load spreads evenly and no single machine gets overwhelmed.
- **Single-table design** — A DynamoDB technique where many different types of records (customers, orders, addresses) share one table, organized by smart key choices so one database call returns everything you need, with no joins.
- **Pre-computing results to make reads faster** — Instead of calculating answers at read time, you can store pre-built answers at write time. Fast to read, more work to write. Twitter does this for most users but skips it for celebrities with 100 million followers because writing to 100 million places per tweet is too expensive.
- **Using multiple databases together (polyglot persistence)** — Large systems often use Postgres for reliable storage, Elasticsearch for search, Redis for speed, and others — each doing what it does best. The hard part: keeping them in sync safely using a “change stream” rather than writing to both at once (which breaks if the app crashes mid-write).
- **Common mistakes** — Switching to NoSQL too early (most apps never need it), picking a bad partition key that sends all traffic to one machine, or treating NoSQL like SQL and ending up with slow app-side joins.

What to do with this: Start with a standard Postgres database — it handles far more than most people think. Only consider NoSQL when you can clearly name every query your app will run and traditional joins are provably too slow. When you do use NoSQL, the partition key choice (spread load evenly, use a high-variety field like user ID) is the single most important decision you'll make.

What you'll learn: How to model data for systems that must survive scale, starting from relational normalization and ending at access-pattern-first NoSQL design (single-table DynamoDB, partition/sort key selection, hot-partition avoidance). You'll learn *why* joins die at scale, how each NoSQL family stores and retrieves data, and how to reason about polyglot persistence like a staff engineer in a design review or a 3 a.m. incident.

Prerequisites: Read [01-foundations-and-estimation.md](#) (back-of-envelope math, latency numbers), [03-partitioning-sharding.md](#) (hash vs range partitioning), and [04-replication-consistency.md](#) (replica lag, read-your-writes). For the consistency trade-offs referenced here, see [09-cap-pacelc.md](#). Caching interactions are covered in [07-caching.md](#).

1. Intuition first: the library vs the warehouse

A **relational database** is a meticulous library. Every fact lives in exactly one place — one shelf for authors, one for books, one for loans. Ask any question (“which books did members born in 1990 borrow?”) and the librarian *joins* shelves on the fly. Flexible, no duplication, but every question requires walking between shelves.

A **NoSQL store** is an Amazon fulfillment warehouse. Instead of one canonical shelf per fact, items are pre-packed into the exact box a customer will request. There's duplication (the same SKU sits in many boxes), and you can't easily answer a question you didn't pre-pack for — but the question you *did* design for is a single grab off a single shelf, at any scale.

The fundamental tension: **relational optimizes for storage flexibility and ad-hoc queries; NoSQL optimizes for a known set of queries at extreme scale by trading storage and write-amplification for read locality.**

2. Normalization: the relational baseline

Normalization removes redundancy so each fact is stored once. The practical targets:

- **1NF** — atomic columns, no repeating groups (no comma-separated lists in a column).
- **2NF** — no partial dependency on part of a composite key.
- **3NF / BCNF** — no transitive dependencies; every non-key column depends on *the key, the whole key, and nothing but the key*.

```
NORMALIZED (3NF)
orders(id, customer_id, created_at)
customers(id, name, email)
order_items(order_id, product_id, qty)
products(id, name, price)

"Show order #42 with customer + items + product names"
→ JOIN orders ⋈ customers ⋈ order_items ⋈ products
```

Why normalize: a customer renames themselves once, in one row. No update anomalies, no contradictory copies, minimal storage. This is correct, and for OLTP systems that fit on one (replicated) primary, it is usually the *right default*. Most apps never outgrow a well-indexed Postgres instance — see the war stories in §11.

Denormalization deliberately reintroduces redundancy to avoid a join: copy `customer_name` onto the order, or store a JSON blob of items on the order row. You trade write complexity and consistency risk for read speed.

	Normalized	Denormalized
Storage	Minimal	Larger (duplicated facts)
Write path	Single write, clean	Must update N copies (write amplification)
Read path	Joins at query time	Pre-joined, single fetch
Consistency	Strong by construction	Risk of stale/divergent copies
Ad-hoc queries	Excellent	Poor (only pre-modeled shapes)
When to use	OLTP, evolving queries, moderate scale	Read-heavy hot paths, known queries, scale-out

3. Why joins die at scale

A join on a single machine is cheap: hash-join or merge-join over local pages, $O(n+m)$. The problem is **distributed joins**. Once `orders` lives on shard A (keyed by `order_id`) and `customers` lives on shard B (keyed by `customer_id`), joining them requires shipping rows across the network — a *shuffle*.

Back-of-envelope: a hash join of two 10 M-row tables co-located in RAM is ~milliseconds. The same join across a 20-node cluster must repartition both tables over the network. At 100 bytes/row that's ~1 GB shuffled each side; at 10 Gbps that's ~1.6 s *just for data movement*, plus tail-latency from the slowest node (see `08-latency-tail-percentiles.md`). The join cost grows with fan-out and the slowest shard, not the average. This is why analytics engines (Spark, Presto/Trino) treat the shuffle as the dominant cost and why OLTP systems avoid cross-shard joins entirely.

The scale-out resolution: don't join at read time. Either (a) pick a partition key so related rows co-locate (all of a customer's orders on one shard), or (b) pre-join at write time (denormalize). NoSQL chooses (b) aggressively.

4. The NoSQL families

“NoSQL” is four different things with different physics. Pick the family by *access pattern*, not by hype.

KEY-VALUE	DOCUMENT	WIDE-COLUMN	GRAPH
key → blob	key → JSON tree	row → sparse columns	nodes + edges
[k]:[.....]	{id, nested[...]}	rowkey cf:col=val ...	(A)-[REL]->(B)
O(1) by key	O(1) by id +	range scan by rowkey,	traverse adjacency
no query inside	secondary indexes	columns grouped on disk	index-free hops

Family	Data model	Reads it's great at	Reads it's bad at	Real systems
Key-value	Opaque value by key	Point get/put by key	Anything not by key	Redis, DynamoDB (core), Riak, memcached
Document	Nested JSON per id	Fetch whole aggregate by id; some secondary-index queries	Cross-document joins, multi-doc transactions (historically)	MongoDB, Couchbase, Firestore
Wide-column	Rows of sparse, grouped columns; range scans within a partition	Time-series, “give me row X cols A–F”, range scans	Ad-hoc filters on non-key columns	Cassandra, ScyllaDB, HBase, Bigtable
Graph	Nodes + edges, index-free adjacency	Multi-hop traversals (friends-of-friends, paths)	Bulk aggregation, full scans	Neo4j, Neptune, JanusGraph

Key insight per family:

- **Key-value** is the substrate everything else is built on. DynamoDB and Redis are key-value at heart; richer models are layers on top.
- **Document** stores reward the *aggregate* pattern (Domain-Driven Design): store the whole consistency boundary (order + its items + shipping) as one document so the common read is one fetch and the common write is one atomic update.
- **Wide-column** is built on the **LSM-tree** (log-structured merge-tree): writes append to a memtable + commit log, flush to immutable SSTables, and compact in the background. This gives huge write throughput (sequential I/O) at the cost of read amplification and compaction load. Cassandra's data model is literally “a distributed, sorted, sparse map indexed by (partition key, clustering columns) .” See the Bigtable paper.
- **Graph** stores precompute adjacency so a hop is a pointer-follow, not an index lookup — “index-free adjacency.” A 5-hop traversal that would be five self-joins in SQL is five pointer hops in Neo4j.

Note on the LSM-tree vs B-tree split: relational engines (Postgres, InnoDB) default to **B-trees** — read-optimized, in-place updates, lower write throughput. LSM engines (Cassandra, RocksDB, modern DynamoDB internals) are **write-optimized**. This is the deepest read-vs-write trade-off in storage and is covered further in DDIA Ch. 3.

5. The access-pattern-first method (the core skill)

Relational modeling starts with *entities* (“what things exist?”). NoSQL modeling starts with *queries* (“what will the application ask, how often, and how fast?”). Reverse the process:

1. Enumerate EVERY access pattern (read + write) with frequency + latency target.
e.g.
 - getOrderById (10k/s, p99 < 10ms)
 - listOrdersByCustomer (2k/s, newest first, paginated)
 - listOrdersByStatus (admin, 50/s)
2. For each, decide the key (partition + sort) or index that answers it in ONE request.
3. Collapse overlapping patterns onto shared keys → minimize number of tables/indexes.
4. Only THEN draw the physical model.

If you can't name the access pattern, you can't model for NoSQL. This is why DynamoDB design reviews open with “list your queries,” not “draw your ER diagram.”

6. Choosing a partition key and sort key

This is the single highest-leverage decision in a NoSQL system. It determines both *correctness of access* and *distribution of load*.

Partition key (PK / hash key): hashed to pick a physical partition. It defines the unit of co-location and the unit of contention. Rules:

- **High cardinality** — many distinct values so load spreads (`user_id` , not `country`).
- **Even access distribution** — values should be hit roughly uniformly over time (avoid `status` where 90% of traffic is `status=PENDING`).
- **Matches your most frequent point/range query's grouping** — everything you fetch together must share a PK.

Sort key (SK / range key): orders items *within* a partition and enables range queries, pagination, and “latest N.” Composite sort keys encode hierarchy:

```
PK = CUSTOMER#123
SK = ORDER#2026-06-16#A1    → range query SK begins_with "ORDER#" gives all orders, sorted by date
SK = PROFILE#               → same partition holds the profile row too
SK = ADDR#home / ADDR#work  → and the addresses
```

This is **item collocation**: one query against `PK=CUSTOMER#123` with a `begins_with` on SK returns the whole customer aggregate in a single round trip — the NoSQL replacement for a join.

Decision	Good choice	Bad choice	Failure if wrong
Partition key	<code>userId</code> (high cardinality, even)	<code>tenantId</code> when one tenant is 90% of traffic	Hot partition, throttling
Sort key	<code>createdAt</code> (enables “latest”, pagination)	random UUID	Can’t range-scan, no ordering
Aggregate boundary	per-customer colocated items	one giant partition for all orders	Unbounded partition growth

7. Single-table design (DynamoDB)

DynamoDB’s most counterintuitive practice: put *multiple entity types in one table* and overload generic keys (`PK` , `SK` , `GSI1PK` ...). The goal is to satisfy many access patterns with the fewest requests, because DynamoDB has no joins and charges per request/RCU.

```

PK          | SK          | type   | attrs...
-----|-----|-----|-----
CUSTOMER#123 | PROFILE#    | customer | name, email
CUSTOMER#123 | ORDER#2026-06-16#A1 | order   | total, status
CUSTOMER#123 | ORDER#2026-06-10#A0 | order   | total, status
ORDER#A1      | ITEM#prod-9  | item    | qty, price
ORDER#A1      | ITEM#prod-3  | item    | qty, price

getCustomerWithOrders: Query PK=CUSTOMER#123    → profile + all orders, 1 request
getOrderWithItems:     Query PK=ORDER#A1         → all line items, 1 request

```

Why this is worth the cognitive cost: the relational version of “customer dashboard” is 3–4 joins; here it’s one `Query` returning a pre-sorted, colocated item collection in single-digit milliseconds at any scale. The cost is rigidity — a genuinely new access pattern may need a new index or a backfill.

When single-table is *wrong*: highly relational, ad-hoc analytical workloads; small apps where the operational simplicity of Postgres wins; or teams without DynamoDB depth (the model is unforgiving). Don’t cargo-cult it. (Alex DeBrie’s *DynamoDB Book* is the canonical treatment.)

8. Secondary indexes in NoSQL

A secondary index lets you query by an attribute that isn’t the table’s primary key — by maintaining a *second copy* of the data keyed differently.

- **Local Secondary Index (LSI):** same partition key, *different sort key*. Shares the partition, so it’s strongly consistent — but bounded by the 10 GB per-partition limit and must be created at table creation.
- **Global Secondary Index (GSI):** *entirely different* partition + sort key. It’s an asynchronously-replicated denormalized copy. **Eventually consistent** (typically <1 s lag, but unbounded under load), with its own

provisioned capacity. The dominant tool for “query by status,” “query by email,” etc.

Base table keyed by (CUSTOMER#, ORDER#) GSI1 keyed by (STATUS, createdAt)
...needs "all PENDING orders, newest first" → write also sets GSI1PK=STATUS#PENDING
DynamoDB async-replicates matching items into GSI1; query GSI1 to get them sorted.

	LSI	GSI	Cassandra secondary index	Cassandra materialized view
Different partition key	No	Yes	No (queries the same node set)	Yes
Consistency	Strong	Eventual	Local/strong-ish	Eventual
Cost model	Shares table capacity	Own capacity	Cheap but scatter-gather	Extra writes
Pitfall	10 GB partition cap	Hot GSI partitions, throttle propagates back	Low-cardinality index = full-cluster scatter	Can silently diverge

The deep failure mode: a GSI with a low-cardinality partition key (e.g. `GSI1PK = status`) recreates the hot-partition problem on the index, and because the index shares the write path, **a throttled GSI throttles the base-table write**. Index choice can break the table it indexes.

9. Materialized views & write-vs-read optimization

You always pay the read-vs-write cost *somewhere*. The lever is **where** you do the work.

WRITE-TIME work (fan-out-on-write) write → compute → store N denormalized copies / push to followers' feeds fast reads, expensive writes good when read:write >> 1	READ-TIME work (fan-out-on-read) write once → read → join/aggregate live cheap writes, expensive reads good when write:read high or fan-out huge

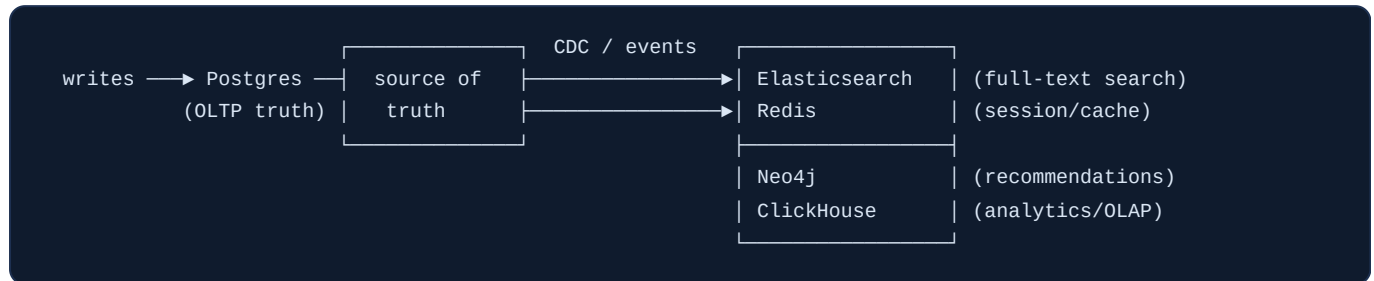
This is exactly Twitter’s timeline problem: fan-out-on-write (push each tweet into all followers’ precomputed timelines) gives O(1) reads but explodes for celebrities with 100 M followers (one tweet = 100 M writes). Twitter uses a **hybrid**: fan-out-on-write for normal users, fan-out-on-read (merge at query time) for celebrities. (DDIA Ch. 1 walks this exact case.)

Materialized views are write-time precomputation managed by the datastore: a continuously-maintained, denormalized query result. Cassandra materialized views, Postgres `MATERIALIZED VIEW` (manual `REFRESH` — *not* automatic), and stream-processing views built from a Kafka changelog (see `12-stream-processing.md`) all implement this. The query that was a 4-table join becomes a single keyed read against the view.

CQRS (Command Query Responsibility Segregation) generalizes this: writes go to a normalized model; a stream of change events projects into many read-optimized denormalized views, each tuned to one query. Polyglot persistence (next) is CQRS taken to its logical end — different *databases* per read shape.

10. Polyglot persistence

Stop forcing one database to be good at everything. Use the right store per access pattern, kept in sync via a change stream (CDC / event log).



Need	Store	Why
Transactions, money, source of truth	Postgres / MySQL	ACID, mature, joins
Full-text / faceted search	Elasticsearch / OpenSearch	inverted index, relevance
Session, cache, rate-limit, leaderboard	Redis	in-memory, sub-ms, sorted sets
Massive write-heavy time-series	Cassandra / ScyllaDB	LSM writes, range scans
Analytics / aggregations	ClickHouse / BigQuery / Snowflake	columnar OLAP
Relationships / paths	Neo4j / Neptune	index-free adjacency

The cost nobody budgets for: every extra store is a copy that can drift, a separate failure/backup/security domain, and a dual-write hazard. **Never dual-write from the app** (write Postgres then Elasticsearch in the same request) — a crash between them leaves them inconsistent forever. Use **Change Data Capture** (Debezium reading the WAL) or the **transactional outbox** pattern so one atomic DB write is the single source of changes that propagate downstream. This is the bridge to [12-stream-processing.md](#).

11. Common pitfalls / war stories

- **Reaching for NoSQL too early.** “We might be the next Google” — then you have 50k rows, no joins, and hand-rolled referential integrity in app code. Postgres handles millions of rows and tens of thousands of TPS on one box. Most “scale” problems are missing-index problems. Outgrow the single node *first*.
- **The hot partition / celebrity key.** A `tenant_id` partition key where one tenant (or `status=ACTIVE`, or the Justin Bieber row) takes 90% of traffic. The partition’s node saturates while the cluster idles. Fixes: **write sharding** (suffix the key with `#0..#N` and scatter-gather reads), composite keys, or isolating whales onto dedicated partitions. This is the most common NoSQL production incident.

- **Modeling NoSQL like SQL.** Normalizing into many DynamoDB tables and then trying to “join” in application code = N+1 queries over the network, each adding a round trip. The whole point was collocation.
- **Unbounded partition growth.** `PK = ORDERS` (all orders in one partition) hits the 10 GB Dynamo partition limit / Cassandra’s wide-row problems. Partitions must be bounded by design (per-customer, per-day buckets).
- **GSI/index throttling the base table.** §8 — a low-cardinality or under-provisioned GSI back-pressures writes to the main table.
- **Assuming GSI / replica reads are strongly consistent.** They lag. A user creates an order then immediately filters “my orders by status” and doesn’t see it (read-your-writes violation; see `04-replication-consistency.md`).
- **Dual writes without an outbox.** App writes DB, then queues an event, then crashes — the event never sends and the search index is permanently stale. Outbox or CDC, always.
- **`SELECT *` of a normalized aggregate over the wire repeatedly** — the relational sibling of N+1; eager-load with `with([...])` (the codebase even mandates this — see `CLAUDE.md` “prevent n+1 queries”).
- **Schema-on-read complacency.** Document stores let you skip migrations, so five years of inconsistent shapes accumulate and every reader needs defensive `?? default` everywhere. Schemaless ≠ structure-free; version your documents.

🌱 Case Study: Amazon DynamoDB & the Retail Shopping Cart

The problem. In the late 2000s Amazon’s retail catalog and shopping-cart services ran on a sharded Oracle relational tier. Every holiday, traffic spiked: Prime Day and Cyber Monday push the platform to tens of millions of requests per second across services, and the shopping cart alone must stay available at the worst possible moment — the seconds before a customer checks out. The relational tier had two fatal properties at that scale: cross-shard **joins** got slower as the cluster grew (§3), and the system favored consistency over availability, so during a partition or a hot node, carts would error out. A cart that throws a 500 during Prime Day is lost revenue measured in millions per minute. Amazon’s internal post-mortems found that ~70% of their operations were simple key-value access (get cart, get item, put item) — the join machinery was paying for flexibility nobody used.

Applying access-pattern-first modeling. This is the exact reversal from §5. The cart team did not draw an ER diagram; they enumerated queries and their SLOs:

<code>getCart(customerId)</code>	~tens of millions/s peak, p99 < 10ms, MUST be available
<code>addItem / updateQty / remove</code>	high write rate, single-item, idempotent
<code>getCustomerOrderHistory</code>	lower QPS, newest-first, paginated

Every one of these is a point or range operation rooted at `customerId`. None of them needs a join. That observation is what makes the workload a NoSQL fit rather than a relational one — and it is the §5 method in its purest form: *if you can name the access pattern, you can pick a key that answers it in one request.*

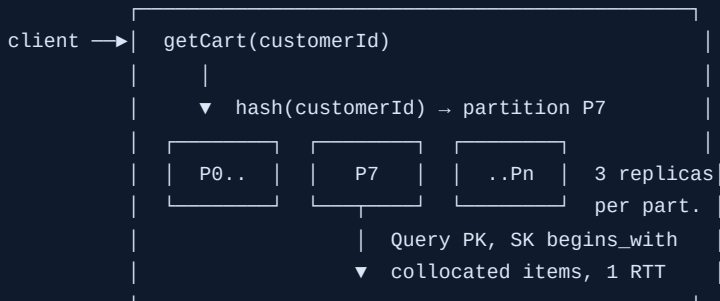
Partition key and sort key choice (§6). `customerId` is the partition key: extremely high cardinality (hundreds of millions of values), hit roughly uniformly, and it is the grouping shared by every frequent query. Within a customer's partition, a composite sort key collocates the cart, profile, and order history — the §6 item-collocation pattern — so the cart read is a single `Query`, not a four-table join.

```
PK = CUSTOMER#88231 | SK = CART#           | item, qty, price (live cart)
PK = CUSTOMER#88231 | SK = CART#           | item, qty, price
PK = CUSTOMER#88231 | SK = ORDER#2026-06-16#A1 | total, status (history)
PK = CUSTOMER#88231 | SK = PROFILE#        | name, prefs

getCart: Query PK=CUSTOMER#88231, SK begins_with "CART#" → whole cart, 1 hop
```

Avoiding hot partitions (§6, §11). Choosing `customerId` instead of something low-cardinality like `region` or `cart_status` is the deliberate hot-partition avoidance from §6: load spreads across millions of distinct partition keys, so no single physical node carries a disproportionate share. Internally DynamoDB also adaptively splits a partition that heats up and uses **consistent hashing** (the Dynamo paper lineage) to redistribute keys when nodes are added — the same mechanism your `03-partitioning-sharding.md` covers, applied so the cart team never has to manually rebalance.

Denormalization over joins (§2, §3). The cart stores a *copy* of item name and price at add-time rather than joining to a live catalog table on every read. That is textbook denormalization: they accept write-time duplication and the risk of a stale price snapshot to get a single-fetch read that never shuffles data across shards. Price re-validation happens once, at checkout, instead of on every cart render — paying the consistency cost exactly where it matters and nowhere else.



The trade-off they accepted. DynamoDB defaults to **eventual consistency** and gives up the relational tier's ad-hoc query power and cross-entity transactions (this is the AP-leaning corner of CAP/PACELC — see `09-cap-pace1c.md`). The cart team chose availability and bounded latency over strong consistency: a read that occasionally reflects state a few hundred milliseconds stale is acceptable for a cart, while an unavailable cart is not. They also accepted the §7 rigidity tax — a genuinely new query shape needs a new GSI or a backfill, not just a fresh `WHERE` clause.

Results. After migrating its consumer-facing services off Oracle (completed around 2018–2019), Amazon reported single-digit-millisecond p99 reads at any scale, and DynamoDB has sustained peaks in the range of **tens of millions of requests per second** during Prime Day with high availability. The retail Oracle exit removed an entire tier of sharding-and-join operational pain; the cart stopped being the thing that fell over under peak load.

Lessons

- **Let the queries pick the database.** ~70% key-value access meant the join engine was pure overhead. Enumerate access patterns *before* choosing relational vs NoSQL — the §5 method is the decision, not just the modeling step.
- **The partition key is a load-balancing decision, not just an addressing one.** `customerId` 's high cardinality is what prevents hot partitions at Prime-Day scale; a “natural” low-cardinality key (`region` , `status`) would have recreated the relational hot-node failure.
- **Denormalize at the boundary that matters.** Snapshotting price into the cart trades a small staleness window for join-free reads — then re-validate once at checkout. Pay the consistency cost where correctness is required, not on every read.
- **Eventual consistency is a feature you buy availability with.** They gave up strong reads and ad-hoc queries on purpose; that purchase is what kept carts answering during the highest-revenue minutes of the year.

Test yourself

1. You must serve “get a user’s 20 most recent orders, paginated” at 5k req/s, p99 < 15 ms. What partition key and sort key do you pick, and why does a random-UUID sort key fail? *Hint: PK groups by user; SK must be time-ordered for “recent” + pagination.*
 2. Why can a single badly-chosen GSI partition key throttle writes to the *base* table? *Hint: GSI shares the write path and has its own (exhaustible) capacity; back-pressure propagates.*
 3. Twitter timelines: when is fan-out-on-write the wrong choice, and what’s the hybrid fix? *Hint: celebrity fan-out cost; merge-on-read for high-follower accounts.*
 4. Give a concrete reason a distributed join across two shards is orders of magnitude slower than the same join co-located on one node. *Hint: network shuffle + tail latency of the slowest shard.*
 5. Your team wants Postgres + Elasticsearch + Redis kept in sync. Why is dual-write from the app a latent data-corruption bug, and what’s the correct pattern? *Hint: no atomicity across two systems; outbox / CDC.*
 6. When is normalized 3NF Postgres the *right* answer over DynamoDB single-table, despite “scale”? *Hint: ad-hoc/evolving queries, transactions, sub-10 M rows, operational simplicity.*
 7. Define a hot partition and list three mitigations. *Hint: skewed key; write-sharding suffix, composite key, whale isolation.*
 8. Why does an LSM-tree (Cassandra) sustain higher write throughput than a B-tree (InnoDB), and what does it pay for it? *Hint: sequential appends + background compaction; read amplification + compaction cost.*
-

13. Further reading

- Kleppmann, *Designing Data-Intensive Applications (DDIA)* — Ch. 2 (data models & query languages), Ch. 3 (storage engines: B-trees vs LSM-trees). The canonical text.

- **Alex DeBrie**, *The DynamoDB Book* — single-table design, key selection, GSIs; plus AWS's "NoSQL design for DynamoDB" docs.
- **Chang et al.**, *Bigtable: A Distributed Storage System for Structured Data* (Google, 2006) — origin of the wide-column model.
- **DeCandia et al.**, *Dynamo: Amazon's Highly Available Key-value Store* (2007) — consistent hashing, eventual consistency, the key-value lineage.
- **O'Neil et al.**, *The Log-Structured Merge-Tree (LSM-Tree)* (1996) — the write-optimized storage paper.
- **Martin Fowler** — "PolyglotPersistence" and "CQRS" (martinfowler.com) — the architectural patterns.
- **Debezium documentation** — Change Data Capture and the transactional outbox pattern.
- Cross-links: [03-partitioning-sharding.md](#) , [04-replication-consistency.md](#) , [07-caching.md](#) , [09-cap-pacelc.md](#) , [12-stream-processing.md](#) .

06 — Caching Deep Dive

Source: [readme/research/system-design/06-caching-deep.md](#)

IN SHORT — KEY TAKEAWAYS

This document is about **caching** — a technique where your software saves a copy of information in a very fast place so it does not have to look it up from the slow database every single time. When done well, caching makes websites feel instant and protects the database from being overwhelmed. When done badly, it can secretly cause bigger outages than having no cache at all.

The main parts explained simply:

- **Why caching works** — Most websites have a small number of pages or products that almost everyone looks at, and a large number that almost nobody touches. Caching that popular 5–20% of information means you can serve 90%+ of visitors without ever touching the database. The math is striking: improving your cache hit rate from 95% to 99% cuts database work by five times.
- **The layers of caching** — Caching is not one single box. It is a stack: the visitor's browser saves some things, a global content delivery network (CDN) like Cloudflare saves more, your app server holds a tiny local copy, a shared fast store like Redis holds the main cache, and even the database quietly keeps hot data in its own memory. Each layer is faster and cheaper than the one below it.
- **How you read and write with a cache (the patterns)** — The most common approach ("cache-aside") checks the cache first; on a miss it loads from the database and stores the result. On a write, it deletes the cached copy rather than updating it — this prevents a tricky bug where two writes race and leave stale data behind. Other patterns (write-through, write-behind) trade off write speed against the risk of losing data if the cache crashes.
- **Eviction policies — what gets removed when the cache is full** — The cache has limited space. LRU ("least recently used") removes whatever was not touched for the longest time and is the most common choice. LFU ("least frequently used") keeps popular items regardless of when they were last read. Modern systems use a smarter mix (W-TinyLFU) that handles both. Redis uses a fast approximation, not a perfect list, to keep overhead low.
- **Cache invalidation — keeping the copy fresh** — This is notoriously hard. The simplest approach is giving every cached item an expiry time (TTL) so it dies automatically. More precise approaches delete the cached copy the moment the real data changes. For pages assembled from many pieces (a category page containing dozens of products), you need tag-based or generation-key invalidation so one product change wipes all the pages that show it.
- **The four failure modes that turn a cache into a liability** — (1) Stampede: one popular item expires and thousands of visitors all hit the database at the same moment. Fix: let only one request reload it while the rest wait. (2) Hot key: one item is so popular it overloads the single server holding it. Fix: keep a short-lived copy on each app server so the shared cache barely sees it. (3) Penetration: attackers flood you with requests for items that do not exist, bypassing the cache every time. Fix: use a Bloom filter — a tiny memory structure that answers "definitely not in the database" in one step. (4) Avalanche: all your cached items expire at the same second (e.g. after a full restart), and the database collapses under the flood. Fix: always add a small random amount to every expiry time so they spread out.
- **Consistent hashing — splitting the cache across many servers** — When one cache server is not enough, you spread keys across many. A naive split (key number divided by server count) means

adding or removing one server re-shuffles almost everything and empties the cache instantly. Consistent hashing puts servers and keys on a ring so adding one server only moves a small fraction of keys.

- **Redis vs Memcached** — Both are fast in-memory stores. Memcached is simpler and uses multiple CPU cores well — good for pure speed. Redis does far more: leaderboards, rate limiters, queues, atomic counters, and it can save data to disk so a restart does not lose everything. Most teams use Redis by default.
- **Facebook's real-world example** — Facebook used plain cache-aside with delete-on-write at billions of requests per second. They added “leases” (only one server reloads a missed item, others wait) to stop stampedes, tailed the database commit log to spread cache deletions across regions automatically, and accepted that data could be a few seconds out of date — a deliberate trade-off to keep the system fast.

What to do with this: Always add a small random amount to every cache expiry time (jitter) — it is a one-line change that prevents mass simultaneous expiry. And when something is not in the database, cache that “not found” answer too, or use a Bloom filter — otherwise attackers or bad requests can flood your database by bypassing the cache entirely.

What you'll learn. How caching actually buys you latency and throughput (locality, the 80/20 rule, and the latency math from [01-foundations-and-estimation.md](#)), the full stack of cache layers from browser to DB buffer pool, every major caching pattern and eviction policy with real implementation detail, and the four classic distributed-cache failure modes (stampede, hot keys, penetration, avalanche) with the mitigations real systems ship. By the end you should be able to design a cache tier, reason about its consistency, and survive an interview question on “why did your cache make the outage worse.”

Prerequisites. Read [01-foundations-and-estimation.md](#) first — you need the latency numbers and back-of-envelope math. [08-replication-and-partitioning.md](#) helps for the consistent-hashing section (sharding the cache is the same problem as sharding a database). [09-cap-pacelc-consistency-models.md](#) deepens the cache ↔ DB consistency discussion.

1. Why caching works: locality, 80/20, and latency math

A cache is a small, fast store that holds a copy of data whose authoritative home (the “origin” or “backing store”) is large and slow. It pays off only because real workloads are **non-uniform**:

- **Temporal locality** — data accessed now is likely accessed again soon (a trending product, a logged-in user's profile).
- **Spatial locality** — data near accessed data is likely accessed too (the next rows of a result set, adjacent pixels of a tile).
- **The 80/20 / Zipfian rule** — in most web workloads a small fraction of keys serve the large majority of requests. Page views, product reads, and search queries follow a Zipf distribution: rank-1 is roughly

twice as popular as rank-2, three times rank-3, etc. This is *why* a cache holding 5–20% of your keys can serve 90%+ of reads.

The latency math (tie-back to 01)

Recall the orders-of-magnitude ladder from 01 :

```
L1 cache reference ..... ~1 ns
Main memory reference ..... ~100 ns
Redis GET over LAN (RTT + processing) . ~0.2-1 ms
SSD random read ..... ~100 µs
Postgres indexed point query ..... ~1-10 ms
Cross-region round trip ..... ~50-150 ms
```

A cache replaces a 1–10 ms DB query (often several, plus app-server CPU and connection-pool wait) with a sub-millisecond memory lookup. The **effective average latency** is governed by the hit ratio:

$$T_{avg} = H \cdot T_{hit} + (1 - H) \cdot (T_{miss} + T_{fill})$$

where **H** is hit ratio, **T_hit** ~ 0.3 ms (Redis), **T_miss** ~ 5 ms (DB), **T_fill** ~ negligible. Plug in numbers — notice how the **tail** is dominated by misses, and how the *last few percent* of hit ratio matter enormously:

Hit ratio H	T_avg (T_hit=0.3ms, T_miss=5ms)	DB load vs no cache
0%	5.0 ms	100%
50%	2.65 ms	50%
90%	0.77 ms	10%
95%	0.535 ms	5%
99%	0.347 ms	1%
99.9%	0.305 ms	0.1%

The killer insight: going from 95% → 99% halves average latency *and* cuts DB load 5×. Going from 99% → 99.9% barely moves latency but cuts DB load another 10×. **The point of a high hit ratio is often DB-load protection, not user latency.** This is also why a cache that *drops* from 99% to 90% during an incident can cause a **10× origin load spike** — the origin was provisioned for 1% of traffic. That is the seed of cache avalanche (§7).

2. The cache layers

Caching is not one box; it's a stack, each layer absorbing what the layer above missed. Numbers are typical.

1. CLIENT / BROWSER	HTTP cache, Service Worker Cache-Control, ETag, localStorage	~0 ms, free scope: 1 user
2. CDN / EDGE	Cloudflare, Fastly, Akamai, Varnish, CloudFront – caches HTML/JSON/assets near user	~10-30 ms scope: a POP region
3. APPLICATION / IN-PROCESS	per-instance LRU map, Caffeine (JVM), array/APCu (PHP), in-memory dict	~0.001 ms (ns-µs) scope: 1 app instance
4. DISTRIBUTED CACHE	Redis, Memcached cluster shared across all app instances	~0.2-1 ms scope: whole fleet
5. DATABASE BUFFER POOL	Postgres shared_buffers, InnoDB buffer pool, OS page cache	~0.01-0.1 ms (RAM) hidden, automatic
6. ORIGIN / DISK	actual table on SSD	~0.1-10 ms

Each layer trades **coherence for speed and scope**. The browser cache is fastest but only one user and easily stale; the distributed cache is shared and authoritative-ish but a network hop away; the DB buffer pool is invisible but is the reason an “uncached” query is sometimes still fast (the rows are already in RAM). A common interview trap: someone “adds Redis” in front of a query that was already 100% served from the Postgres buffer pool — they added a network hop and won nothing. Always measure where the time actually goes.

Designing the stack: push caching as far up and out as the consistency budget allows. Static assets → CDN with year-long TTL + content-hash filenames. Personalized HTML → edge with short TTL or `private`. Hot read-mostly objects → distributed cache. Per-request-repeated lookups → in-process. In Print-Flow-360 terms, this maps directly to `readme/STOREFRONT_CACHE_GUIDE.md` (CacheService TTLs) and the Nitro route-rule cache in `frontstore/nuxt.config.ts` — note the rule there that private routes (`/cart`, `/checkout`) must stay `no-store`, which is exactly layer-1/2 correctness.

3. Caching patterns (read & write)

The pattern decides *who* fills the cache, *who* writes the DB, and *what consistency* you get. This is the most-asked design topic.

3.1 Cache-aside (lazy loading) — the default

The application owns the logic. The cache is a dumb store.

```

READ:
  app —GET key→ cache
      miss
  app —SELECT→ DB —row→ app
  app —SET key, TTL→ cache      (populate on miss)
  return row

WRITE:
  app —UPDATE→ DB
  app —DEL key→ cache          (invalidate, don't update)

```

- **Pros:** simple; cache holds only what's actually read; cache failure ≠ data loss (DB is truth); works with any cache.
- **Cons:** first read of any key is a miss (cold cache); you own invalidation; a race window exists (see §8).
- **Used by:** almost everyone. Facebook's memcache (the *Scaling Memcache at Facebook* paper) is cache-aside with `delete` on write.

Why DEL not SET on write? Updating the cache on write seems efficient but creates a nasty stale-write race (two concurrent writers can interleave SET-cache and UPDATE-db so the cache ends with the *older* value). Deleting is idempotent and lets the next reader repopulate from the source of truth. "Invalidate, don't update" is the safe default.

3.2 Read-through

Same read shape as cache-aside, but the cache *library/provider* does the DB load behind a loader function, so the app just calls `cache.get(key)`. Encapsulates the miss logic (good for consistency across call sites; e.g. Caffeine `LoadingCache`, AWS DAX for DynamoDB).

3.3 Write-through

Every write goes to the cache, which synchronously writes the DB before returning.

```

WRITE:  app —SET→ cache —UPDATE (sync)→ DB —ok→ cache —ok→ app

```

- **Pros:** cache always consistent with DB; reads after write are guaranteed warm.
- **Cons:** every write pays cache + DB latency; caches data that may never be read (write-heavy, read-rare data wastes cache memory). Often paired with read-through.

3.4 Write-behind / write-back

Write to cache, acknowledge immediately, flush to DB asynchronously (batched/coalesced).

```

WRITE:  app —SET→ cache —ack→ app
          | (async, batched)
          ▼
          DB

```

- **Pros:** lowest write latency, absorbs write bursts, coalesces many writes to one key into one DB write (great for counters, view-counts, leaderboards).
- **Cons:** **data-loss window** if the cache node dies before flush; complex ordering/consistency; the DB temporarily lags. Use only where some loss is tolerable or the cache is durable (Redis AOF, or a WAL in front).

3.5 Refresh-ahead

Proactively refresh popular keys *before* they expire, so reads never hit a cold key. Predicts hot keys (or refreshes the top-N by access). Reduces tail latency for predictable hot data; wasteful and wrong for low-locality data. Closely related to probabilistic early expiration (§7).

Comparison

Pattern	Read path	Write path	Consistency	Write latency	Best for	Risk
Cache-aside	app loads on miss	DB then DEL cache	eventual, small race window	DB only	general read-mostly (the default)	cold-start misses; you own invalidation
Read-through	provider loads on miss	(usually + write-through)	eventual	—	uniform encapsulated access	provider lock-in
Write-through	warm after write	cache → DB sync	strong (cache=DB)	cache + DB	read-after-write critical	slow writes; caches cold data
Write-behind	warm	cache → DB async	eventual, lag	cache only (fast)	counters, bursty writes, leaderboards	data loss on crash
Refresh-ahead	never cold for hot keys	(orthogonal)	eventual	—	predictable hot keys	wasted refresh on cold keys

4. Eviction policies

A cache has bounded memory; when full, something must go. The policy decides what.

- **FIFO** — evict oldest-inserted. Ignores access frequency; cheap; usually bad hit ratio. Suffers Bélády's anomaly (more cache can mean fewer hits).
- **LRU (Least Recently Used)** — evict the item not touched for longest. Excellent for temporal locality; the workhorse. Weakness: a one-time scan of many cold keys (a batch job, a crawler) evicts your hot set — “cache pollution” / scan resistance failure.
- **LFU (Least Frequently Used)** — evict the least-accessed. Great for stable popularity; weakness: a key that was hot yesterday stays resident forever (“cache staleness”), and new items struggle to build a

count. Needs aging/decay to be practical.

- **TTL** — not really an eviction policy but an expiry: each key dies after a fixed time regardless of memory. Combined with LRU/LFU in practice (Redis: TTL expiry + maxmemory eviction).
- **ARC (Adaptive Replacement Cache)** — keeps two lists (recently used + frequently used) plus “ghost” entries (recently evicted keys with no data) and self-tunes the balance between recency and frequency. Scan-resistant. Patented by IBM (used in ZFS, historically).
- **W-TinyLFU** — modern state of the art (Caffeine’s default). A tiny LRU “window” admits new items; a frequency sketch (a Count-Min Sketch with aging — see [15-probabilistic-structures-and-algorithms.md](#)) decides whether a new candidate is more valuable than the eviction victim. Near-optimal hit ratios with $O(1)$ overhead and tiny metadata. This is what you reach for today if you control the cache library.

How LRU is actually implemented

Naively, LRU needs $O(1)$ “move to front on access” and $O(1)$ “evict from back.” The classic structure is a **hash map + doubly-linked list**:

```
hashmap: key → node
list (MRU ... LRU): [A] ⇄ [B] ⇄ [C] ⇄ [D]
                    head   tail
GET(B): map lookup  $O(1)$  → unlink B → splice B to head → return value
PUT(X) when full: evict tail (D) → remove from map → insert X at head
```

Every operation is $O(1)$. The doubly-linked list gives $O(1)$ unlink; the map gives $O(1)$ find. This is the canonical “LRU cache” coding-interview problem.

Real caches cheat for speed. Redis does **not** keep a perfect global LRU list — that costs memory and a list mutation per access. Instead Redis uses **approximate LRU**: each object stores a 24-bit access timestamp (`lru` field); on eviction Redis samples N random keys (`maxmemory-samples` , default 5) and evicts the oldest of the sample. Higher sample size → closer to true LRU, more CPU. Redis 4.0+ also offers **approximate LFU** (a logarithmic counter with time decay) via `maxmemory-policy allkeys-lfu` . Memcached uses a segmented LRU with per-slab-class lists. The lesson: at scale, *approximate* eviction with bounded sampling beats exact eviction with global bookkeeping.

Policy	Captures	Weakness	When to use
FIFO	nothing useful	poor hit ratio	almost never
LRU	recency	scan pollution	general default, recency-driven workloads
LFU	frequency	stale hot keys; cold-start	stable popularity (CDN-like)
TTL	freshness	not memory-aware	always, combined with above
ARC	recency + frequency, adaptive	complexity, patent	filesystem/page caches
W-TinyLFU	recency + frequency, scan-resistant	needs library support	modern app caches (Caffeine)

5. Cache invalidation — one of the two hard problems

“There are only two hard things in Computer Science: cache invalidation and naming things.” — Phil Karlton

Invalidation is hard because the cached copy and the source of truth drift, and you must decide *when* and *how* to reconcile, across many readers/writers/layers. Strategies, weakest to strongest:

1. **TTL expiry (passive).** Let entries die after N seconds; accept up to N seconds of staleness. Dead simple, no coordination, the backbone of CDN/HTTP caching. Tune TTL to your staleness budget. The whole art is picking N — and **jittering** it (§7).
2. **Explicit invalidation on write (active).** Cache-aside `DEL key` after the DB write. Precise, but you must enumerate every key derived from the changed data — including *fan-out* keys (a product change invalidates the product page, the category listing, the search index, the homepage rail...). Missing one = a silent stale read.
3. **Versioned / immutable keys.** Embed a version or content hash in the key: `product:42:v17` or `app.a3f9.css`. A write bumps the version; old keys are simply never read again and age out naturally. **No invalidation race at all** — this is the strongest, cheapest technique when you can do it (CDN asset hashing lives here). Cost: you need to propagate the new version to readers.
4. **Write-through / read-through** keep cache and DB in lock-step (§3) — invalidation by construction.
5. **Change-data-capture (CDC) invalidation.** Tail the DB’s replication log (Postgres logical decoding, Debezium) and invalidate/refresh cache entries from the authoritative change stream. Decouples invalidation from app code, catches every write including out-of-band ones. See `12-messaging-and-event-driven.md` and `14-stream-processing-realtime.md`.

The fan-out problem is the real killer. A single product edit might require invalidating dozens of derived caches. Two tactics: (a) **tag/group invalidation** — tag every derived key with `product:42` and flush the

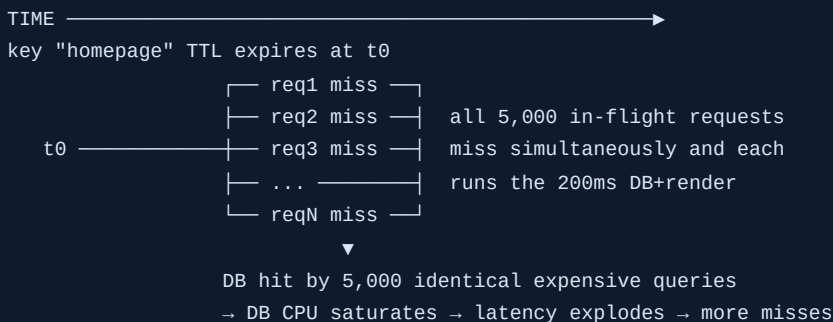
tag (Varnish “ban”, Cache Tags in many frameworks); (b) **generation keys** — store `gen:category:5 = 88` and build dependent keys as `category:5:g88:...`; bump the generation to invalidate the whole family in $O(1)$ without enumerating keys.

6. Failure modes that make caches a liability

A cache is a performance optimization that quietly becomes a **load-bearing structural component**: once the DB is provisioned for 1% of traffic (§1), the cache *must not fail*. These four failure modes are the canonical interview material.

6.1 Cache stampede / thundering herd

A hot key expires (or a cold cache starts). Many concurrent requests all miss, all hit the DB at once, all recompute the same expensive value, and may all write it back — a self-inflicted load spike that can topple the origin.



Mitigations:

- **Request coalescing / single-flight.** Only the *first* misser recomputes; everyone else waits on the same in-flight promise/future. In Go this is golang.org/x/sync/singleflight; in a single process it's a per-key lock/promise map; across processes it's a distributed lock (see below). This is the single most important fix.
- **Distributed lock on recompute.** First misser takes a short-lived lock (`SET key:lock val NX EX 10`); others either wait-and-retry or serve stale. Beware lock correctness (see Redlock debate in [10-consensus-and-coordination.md](#)); a *best-effort* lock that occasionally lets two recompute through is fine here — the goal is “few” not “exactly one.”
- **Probabilistic early expiration (XFetch).** Don't wait for hard expiry. Each reader recomputes with a small probability that grows as the entry nears expiry, so *one* lucky reader refreshes the key *before* it expires while it's still being served warm. The classic formula (from Vattani et al., “Optimal Probabilistic Cache Stampede Prevention”):

```
recompute if: now - delta · beta · ln(rand()) ≥ expiry
```

where `delta` is the measured recompute cost and `beta ≥ 1` tunes eagerness. Elegant because it needs **no locks and no coordination** — it staggers refreshes statistically.

- **Serve-stale-while-revalidate.** Keep the expired value, return it immediately, and refresh in the background (HTTP `stale-while-revalidate`, Nuxt/Nitro SWR, Varnish grace mode). Users never wait on a miss.

6.2 Hot keys (the celebrity problem)

One key is so popular that the single cache shard holding it saturates its CPU/network even though the rest of the cluster is idle (a viral product, a global config blob, a celebrity's profile). Sharding (§9) doesn't help — every read hashes to the *same* node.

Mitigations:

- **Local/in-process cache (L1) in front of the distributed cache (L2).** Each app instance caches the hot key locally with a short TTL (1–5 s), so the shard sees `#instances` requests/sec, not `#users`. This is the most common fix and exactly the layered stack in §2. Accept tiny extra staleness.
- **Key splitting / replication.** Store N copies `cfg:0..cfg:9`, readers pick a random suffix → load spreads across N shards. Writes must update all N.
- **Read replicas of the cache** (Redis replicas) for read fan-out.

6.3 Cache penetration

Requests for keys that **don't exist anywhere** (random/garbage IDs, a malicious scraper hitting `user:99999999`). Every such request misses the cache *and* misses the DB, so the cache provides zero protection and the DB takes the full hit.

Mitigations:

- **Negative caching.** Cache the “not found” result too (`user:99999999 = NULL`, short TTL). Now repeat lookups for the same bad key are absorbed. Watch memory if the bad keys are unique each time.
- **Bloom filter.** Keep a Bloom filter (see `15-probabilistic-structures-and-algorithms.md`) of all valid keys in memory; before hitting cache/DB, test the filter — if it says “definitely not present,” reject in O(1) without touching the DB. No false negatives means you never wrongly reject a real key; the small false-positive rate just means a few bad keys slip through to the DB. This is the standard defense against ID-enumeration attacks.

6.4 Cache avalanche

Many keys expire *at the same instant* (e.g. you warmed the whole cache at deploy time with identical TTLs, or the entire Redis cluster restarts), so a huge swath of traffic falls through to the DB simultaneously → origin collapse. The macro version of a stampede.

Mitigations:

- **Jittered TTLs.** Never use a fixed TTL; add randomness: `ttl = base + rand(0, base·0.1)`. Spreads expiry over a window so misses don't synchronize. (This is the cheap fix you should *always* apply, by default.)

- **Multi-level caching + serve-stale** so a Redis outage degrades to L1/stale, not to the DB.
- **Circuit breaker + request shedding at the origin** (see [16-rate-limiting-and-resiliency.md](#)) — when the DB is overloaded, shed/queue rather than pile on.
- **Cache warming / staggered restart** so a cold cluster repopulates gradually behind a rate-limited backfill.

Failure	Trigger	Primary fix	Secondary
Stampede	one hot key expires, many concurrent misses	single-flight / lock	XFetch, stale-while-revalidate
Hot key	one key far hotter than others	L1 local cache	key splitting, replicas
Penetration	requests for non-existent keys	bloom filter	negative caching
Avalanche	mass simultaneous expiry / cache outage	jittered TTL	multi-level, circuit breaker, warming

7. Putting it together: a robust read path

Production read paths combine several of the above. A battle-tested cache-aside read for a hot, expensive object looks like:

```

get(key):
    v = L1.get(key)                # in-process, ns
    if v: return v
    v = redis.get(key)             # distributed, ~0.5ms
    if v:
        L1.set(key, v, ttl=2s)    # absorb hot keys (§6.2)
        if should_early_refresh(v): # XFetch (§6.1)
            background_refresh(key)
        return v
    if not bloom.maybe_contains(key): # penetration guard (§6.3)
        return NOT_FOUND
    if redis.set(lock, 1, NX, EX=5): # single-flight (§6.1)
        v = db.load(key)           # ~5ms
        redis.set(key, v or NULL_SENTINEL, # negative caching (§6.3)
                    ttl = base + jitter()) # avalanche guard (§6.4)
        redis.del(lock)
        return v
    else:
        sleep(20ms); return get(key) # wait for the leader's fill

```

Every line maps to a §6 mitigation. That density is the point — a “cache” in a senior design is this whole assembly, not `Cache::remember()`.

8. Write consistency between cache and DB (the dual-write problem)

You are writing to *two* stores (cache + DB) without a transaction spanning them. Any interleaving can leave the cache stale. Classic race with cache-aside “DEL on write”:

```
T1 (read, key cold)      T2 (write)
miss
SELECT → v_old

                        UPDATE → v_new
                        DEL key      ← deletes nothing (key absent)
SET key = v_old          ← writes STALE value, no TTL bound until expiry
```

Now the cache serves `v_old` until TTL. Mitigations, roughly increasing strength:

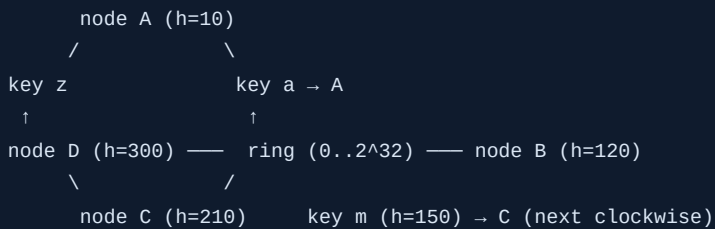
- **Always set a TTL** so any stale write self-heals in bounded time (the pragmatic 80% fix).
- **Delete after the DB commit, not before**, and consider **delayed double-delete**: `DEL key` now, then `DEL key` again after a short delay (e.g. 500 ms) to clobber any racing stale repopulation.
- **Version/CAS** the cached value (store `(value, version)` ; only overwrite if your version $\geq$ stored) so a late stale SET can't win.
- **Write-through** to remove the dual-write entirely (one write path), at write-latency cost.
- **CDC-driven invalidation** (§5.5): the cache is invalidated *from the commit log*, so it can't drift from committed state and out-of-band writes are covered. This is the strongest decoupled answer and ties to `11-distributed-transactions-and-idempotency.md` (the broader dual-write problem) and the outbox pattern.

Note: you cannot get *strong* read-your-writes through a remote cache without either write-through or read-from-primary-on-recent-write. See `09-cap-pacelc-consistency-models.md` — a cache is an explicitly eventually-consistent replica.

9. Distributed caching and consistent hashing

One cache node isn't enough RAM or throughput, so you shard keys across N nodes. The naive map is `node = hash(key) % N`. The catastrophe: **change N (add/remove a node) and almost every key remaps**, so nearly the entire cache misses at once → instant avalanche (§6.4).

Consistent hashing fixes this. Map both nodes and keys onto a hash ring `[0, 232)`. A key belongs to the first node clockwise from its hash. Adding/removing a node only remaps the keys in *one arc* — on average `K/N` keys move, not all of them.



Add node E at h=140 → only keys in (120,140] move from C to E.
Everything else stays put.

Real systems use **virtual nodes** (each physical node placed at many ring positions, e.g. 100–200 vnodes) to smooth out uneven arc sizes — otherwise three random points on a ring give wildly unequal load. Consistent hashing powers Memcached client libraries (ketama), DynamoDB, Cassandra, and CDN request routing. The detailed treatment (and alternatives like rendezvous/HRW hashing, and bounded-load consistent hashing) is in [08-replication-and-partitioning.md](#). Memcached deliberately keeps sharding **client-side** (nodes don't know about each other — “no central coordination”); Redis Cluster shards server-side via 16,384 hash slots assigned to nodes.

10. Redis vs Memcached

Both are in-memory key-value stores reached over the network. They diverge sharply beyond **GET** / **SET**.

Dimension	Memcached	Redis
Data model	opaque string/blob values only	rich types: strings, hashes, lists, sets, sorted sets, streams, bitmaps, HyperLogLog, geo, bitfields
Threading	multi-threaded (scales on many cores per node)	mostly single-threaded command exec (I/O threads in 6+); simpler, no data races, predictable
Persistence	none (pure cache)	optional RDB snapshots + AOF log → survives restart
Replication / HA	none built-in (client-side sharding)	replicas, Sentinel failover, Redis Cluster
Eviction	slab LRU	LRU/LFU/random/TTL, approximate (§4), per-policy
Memory efficiency	slab allocator, very low per-item overhead; great for many small uniform items	more per-key overhead, but specialized encodings (ziplist/listpack, intset) for small collections
Atomic ops	CAS, incr/decr	huge atomic command set + Lua scripts + transactions (MULTI/EXEC) + functions
Pub/Sub, queues, locks	no	yes (Pub/Sub, Streams, sorted-set queues, Redlock)
Max value size	1 MB default	512 MB
Best when	pure, simple, multi-core, throughput-max look-aside cache; huge fleet of tiny items (Facebook)	you need data structures, persistence, HA, atomic server-side logic, or cache + lightweight datastore in one

Rule of thumb: if all you need is a fast look-aside string cache and you want to saturate many cores per node, Memcached is leaner and simpler. If you need anything richer — rate limiters, leaderboards, sessions that survive restart, queues, locks, atomic counters — Redis. Most teams default to Redis today for the Swiss-army-knife value, accepting its single-threaded ceiling (which you scale past via Cluster or sharding).

11. Redis advanced data structures & tricks

Why Redis wins designs: it moves logic *to the data*, atomically, server-side.

- **Sorted sets (ZSET)** → leaderboards (`ZADD` / `ZREVRANGE`), priority queues, time-windowed rate limiters (`ZADD ts` , `ZREMRANGEBYSCORE` to drop old, `ZCARD` to count — the sliding-window limiter in `16-rate-limiting-and-resiliency.md`), and “schedule by score = run-at-timestamp.”
- **Hashes** → store an object’s fields without serializing the whole blob; `HINCRBY` one field atomically. Small hashes use a compact listpack encoding (very memory-efficient).
- **HyperLogLog** (`PFADD` / `PFCOUNT`) → count unique visitors in ~12 KB with ~0.81% error regardless of cardinality (see `15-probabilistic-structures-and-algorithms.md`).
- **Bitmaps / SETBIT** → daily active users, feature flags per user-id, retention grids — 1 bit per user.

- **INCR / DECR** → atomic counters; the building block of fixed-window rate limiting and write-behind view counters.
 - **Lua scripts / Functions** → execute multi-step read-modify-write atomically server-side (e.g. check-and-decrement inventory, token-bucket refill) with no round-trip races. The server runs it as one unit (single-threaded = effectively serializable).
 - **SET key val NX EX 10** → the one-liner distributed lock / single-flight primitive (mind correctness caveats; [10-consensus-and-coordination.md](#)).
 - **Streams (XADD /consumer groups)** → a lightweight Kafka-like log with at-least-once delivery and acks ([12-messaging-and-event-driven.md](#)).
 - **Pipelining** → batch many commands in one round trip; turns $N \times \text{RTT}$ into $1 \times \text{RTT}$ (huge for the LAN-latency-bound case). Combine with **MGET / MSET** for multi-key reads.
 - **EXPIRE / PEXPIRE + lazy + active expiry** → Redis expires keys both lazily (on access) and via a background sampler; this is why you set TTLs liberally.
-

12. Common pitfalls / war stories

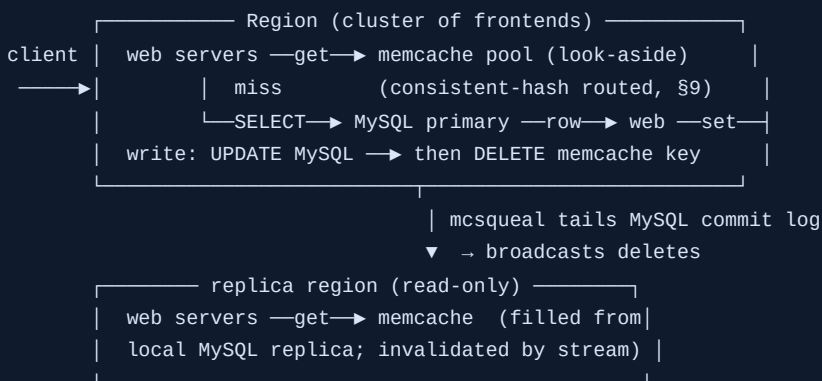
- **The cache that became the database.** Team set Redis **maxmemory-policy noeviction** and stored the *only* copy of session data with no persistence. A node restart logged out every user mid-checkout. Lesson: a cache is disposable by definition; if losing it hurts, it's a datastore — give it persistence/replication and treat it as such.
- **The TTL avalanche after deploy.** A nightly job warmed the whole product catalog into Redis at 02:00 with a uniform 6-hour TTL. At 08:00 — peak traffic — every key expired in the same minute and the DB fell over. Fix was one line: jitter the TTL. (§6.4)
- **The Postgres-buffer-pool illusion.** Engineers “added Redis” to speed a query and saw *zero* improvement, because the working set already lived in **shared_buffers** — the network hop to Redis was as slow as the in-RAM index scan. Always profile *where* the latency is before adding a layer (§2).
- **The fan-out miss.** A price edit invalidated **product:42** but not the cached category listing and homepage rail that embedded the old price — customers saw stale prices for 24 h. Caching derived/aggregate views needs tag or generation invalidation, not per-object DEL (§5). (This is also the “silent-lie” failure class the project [CLAUDE.md](#) warns about, surfacing through the cache layer.)
- **The hot config key.** A global feature-flag blob in Redis read on every request pinned one shard's CPU at 100% while the cluster idled. A 2-second in-process L1 cache dropped that shard's traffic 1000× (§6.2).
- **Update-on-write stale race.** A write path did **SET cache=newval** instead of **DEL cache** ; under concurrency the cache occasionally pinned the *old* value indefinitely (§3.1, §8). Switched to delete-with-TTL.
- **KEYS * in production.** Someone ran **KEYS prefix:*** to enumerate keys; it's $O(N)$ and blocks Redis's single thread, stalling the whole fleet. Use **SCAN** (cursor-based, non-blocking) always.

- **Caching personalized/private data at a shared layer.** A logged-in user's HTML got cached at the CDN/edge and served to other users. Private routes must be `no-store / private` (the exact `frontstore/nuxt.config.ts` rule for `/cart` , `/checkout`).

🌱 Case Study: Scaling Memcache at Facebook

The problem. By the early 2010s Facebook served well over a billion users from a read-dominated social graph — your news feed, friends, profiles. A single page load could fan out to **hundreds or thousands** of distinct data items, and the workload was overwhelmingly read-heavy (reads outnumbered writes by roughly two to three orders of magnitude). At that volume the system fielded on the order of **billions of memcache requests per second** across the fleet, with individual leaf nodes handling on the order of a million ops/sec. No relational database tier could survive that read load directly. The mandate: absorb nearly all reads in RAM, keep latency low enough that a single page assembling thousands of items stayed snappy, and never let the cache tier turn into a load amplifier that toppled MySQL.

Pattern: cache-aside at scale (§3.1). Facebook ran the textbook **cache-aside / look-aside** pattern, exactly as described above. On a read, the web server checks memcache; on a miss it loads from MySQL and populates memcache. On a write it updates MySQL and then **deletes** the key — the “invalidate, don't update” rule from §3.1. Deletes are idempotent and avoid the stale-write race; the next reader repopulates from the source of truth. Memcache itself stayed a dumb, fast store; all the logic lived in the application, with **client-side consistent hashing** (§9) routing each key to its leaf node (no central coordinator — the Memcached design philosophy from §10).



Pattern: leases for thundering herd + stale reads (§6.1, §8). A hot key expiring under this fan-out is the classic stampede of §6.1 — thousands of in-flight requests miss at once and dogpile MySQL. Facebook's elegant fix was the **lease**: on a miss, memcache hands exactly one client a 64-bit *lease token* and makes the others wait briefly and retry. Only the lease holder is allowed to recompute and `set` the value — this is **request coalescing / single-flight (§6.1)** implemented inside the cache server itself, no external distributed lock needed. The same lease mechanism *also* solves the dual-write stale-read race from §8: if a `delete` invalidates the key while a recompute is in flight, the outstanding lease token is voided, so the racing `set` of a now-stale value is **rejected**. One primitive kills both the stampede and the stale-set

interleaving. They paired this with **stale-while-revalidate**-style behavior (serving a slightly old value while one client refreshes) to keep tail latency flat during refreshes.

Pattern: regional invalidation (§5). Facebook ran multiple geographic regions, each a full memcache + MySQL-replica stack fronting a single primary region for writes. A write in the primary had to invalidate caches *everywhere*, or replica regions would serve stale data indefinitely. Doing this from application code is the **fan-out invalidation problem of §5** at planetary scale, and it's fragile (miss one path, get a silent stale read). Their answer is the strongest technique in §5: **CDC-driven invalidation**. A daemon called `mcsqueal` tails the MySQL **commit log** and broadcasts the resulting cache deletes out to the memcache tiers — invalidation derived from the authoritative change stream, not from hopeful app code, so even out-of-band writes are covered. Within a region they also used **gutter pools** (a small standby pool that absorbs traffic for keys whose normal leaf node has failed) to stop a single dead node from raining its share of misses onto MySQL — a direct mitigation against the avalanche dynamic of §6.4.

The trade-off they accepted. Facebook deliberately chose **best-effort eventual consistency over strong consistency**. The cache ↔ DB relationship is the explicitly eventually-consistent replica of §8: a delete can be lost, a cross-region replication lag can briefly expose old data, and they accepted a small, bounded staleness window in exchange for being able to serve the read volume at all. As the paper puts it, they optimize to **reduce the probability and duration** of stale reads rather than eliminate them. They gave up linearizability to keep the read tier in RAM and the latency low — a textbook PACELC “choose latency” call (`09-cap-pacelc-consistency-models.md`).

Results. Cache hit ratios sat in the high-90s, so MySQL saw only a tiny fraction of read traffic — the §1 insight that the last few percent of hit ratio is *DB-load protection*, not user latency, made literal at fleet scale. Leases cut the database load from stampedes by orders of magnitude on hot keys, and individual memcache nodes sustained ~1M requests/sec at sub-millisecond latency. The system scaled cache-aside from one cluster to many regions without changing the core pattern — proof that the §3.1 default, hardened with §6 mitigations, goes a very long way.

Lessons

- **The default pattern scales — if you harden it.** Plain cache-aside with `delete` -on-write got Facebook to a billion users; what made it survivable was layering §6 mitigations (leases = single-flight, gutter pools = avalanche guard) on top, not replacing the pattern.
- **One well-chosen primitive can solve two problems.** The lease handled *both* thundering herd (§6.1) and the dual-write stale-read race (§8) at once. Look for invariants that collapse multiple failure modes.
- **Invalidate from the source of truth, not from app code.** At fan-out scale, CDC-style invalidation off the commit log (`mcsqueal`) is more reliable than enumerating derived keys in every write path — exactly the §5 “strongest, decoupled” answer.
- **Pick your consistency target explicitly.** They didn't accidentally end up eventually consistent; they *chose* to minimize staleness probability/duration rather than guarantee freshness, because the alternative couldn't carry the load.

13. Test yourself

1. Your cache hit ratio drops from 99% to 90% during an incident. By roughly what factor does origin DB load increase, and why is this often more dangerous than the latency change? (Hint: §1 table — miss rate 1% → 10%.)
 2. Why does cache-aside delete the key on write instead of updating it with the new value? Give the concrete race that updating creates. (Hint: §3.1 / §8 interleaving.)
 3. A single product becomes viral and one Redis shard saturates while the cluster is idle. Why doesn't adding more shards help, and what's the standard fix? (Hint: §6.2 — same hash, L1 local cache.)
 4. Explain probabilistic early expiration (XFetch) and what advantage it has over a distributed lock for stampede prevention. (Hint: §6.1 — no coordination, staggered refresh before expiry.)
 5. You shard a Memcached cluster with `hash(key) % N`. What happens to your hit ratio the moment you add one node, and what hashing scheme avoids it? (Hint: §9 — mass remap vs consistent hashing / vnodes.)
 6. When would you choose write-behind over write-through, and what's the failure you're accepting? (Hint: §3.4 — bursty counters; data-loss window on crash.)
 7. How does a Bloom filter defend against cache penetration, and why is its false-positive (not false-negative) property the safe one here? (Hint: §6.3 / 15 — “definitely not present” never rejects a real key.)
 8. Redis is single-threaded for command execution. Name two consequences — one benefit, one risk — and how `KEYS *` interacts with it. (Hint: §10/§12 — serializable atomicity vs one slow $O(N)$ command blocking everyone.)
-

14. Further reading

- Nishtala et al., “**Scaling Memcache at Facebook**” (NSDI 2013) — the canonical real-world cache-aside paper: leases (stampede prevention), regional pools, cold-cluster warmup, and the `delete`-on-write consistency model. Read this twice.
- Vattani, Chierichetti, Lowenstein, “**Optimal Probabilistic Cache Stampede Prevention**” (VLDB 2015) — the XFetch derivation.
- **Redis documentation** — data types, eviction (`maxmemory-policy` , approximate LRU/LFU), persistence (RDB/AOF), Cluster, and `SCAN` vs `KEYS` .
- **Memcached wiki** — slab allocation, LRU, and the rationale for client-side sharding.
- Kleppmann, **Designing Data-Intensive Applications** — Ch. 1 (latency/percentiles), Ch. 5 (replication & consistency, underpinning cache ↔ DB), Ch. 6 (partitioning / consistent hashing).
- Karger et al., “**Consistent Hashing and Random Trees**” (1997) — the original; pairs with `08-replication-and-partitioning.md` .
- Caffeine (Java) design notes on **W-TinyLFU** — the modern eviction state of the art.

- Project-local: `readme/STOREFRONT_CACHE_GUIDE.md` , `readme/PERFORMANCE_RND.md` , and the Nitro route-cache warning in `frontstore/nuxt.config.ts` .
-

Next: `07-load-balancing-and-scaling.md` (where caching meets request distribution) and `16-rate-limiting-and-resiliency.md` (the circuit breakers and limiters that protect the origin when the cache fails).

07 · Load Balancing, Scaling & Stateless Design

Source: [readme/research/system-design/07-load-balancing-and-scaling.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains how a website that runs on one computer grows into a fleet of many computers that can handle big crowds without breaking. It covers how traffic gets shared across those computers, how the system detects a broken server and stops sending it customers, and what you must do to your app before any of this works. Every serious web service relies on these ideas.

The main parts explained simply:

- **Scaling up vs scaling out** — You can either buy a bigger, faster single machine (scaling up) or add more machines and split the work between them (scaling out). Scaling out is the long-term answer because it has no ceiling and keeps running even if one machine dies.
- **Load balancer** — A traffic cop that sits in front of all your machines and decides which machine handles each incoming request. Without it, every request hits the same one machine.
- **L4 vs L7 load balancer** — An L4 balancer just sees an incoming connection and sends it somewhere. An L7 balancer can actually read the web address and headers, so it can send “/checkout” to one group of servers and “/images” to another. L7 is smarter but costs more to run.
- **Balancing algorithms** — The rules the traffic cop uses to pick a machine. Round-robin takes turns evenly. “Power of two choices” picks two random machines and sends to the less-busy one — a small trick that gives nearly perfect balance with no extra coordination.
- **Health checks** — The load balancer pings each machine every few seconds. If a machine stops responding, the balancer stops sending it customers. The trap: if the check also tests the database and the database hiccups, the balancer can eject every machine at once, turning a small problem into a full outage.
- **Stateless design** — The keystone idea. If a machine holds nothing about the user in its own memory (sessions stored in Redis instead), any machine can serve any request. This is what makes adding or removing machines safe and painless. Sticky sessions (storing the session on the machine itself) fights every other scaling tool.
- **Autoscaling** — Automatically adding machines when traffic rises and removing them when it drops. The catch: new machines take time to boot, so you must trigger scale-out before you are already overloaded, not after.
- **Connection draining** — When you remove a machine (for a deploy or to shrink the fleet), you first stop sending it new requests, wait for it to finish the requests already in progress, then shut it down. Skip this and active users get errors mid-action.
- **Database scaling** — App servers are easy to scale out. Databases are harder because they hold the actual data. Read replicas let many machines answer read queries; sharding splits the data across multiple independent databases to handle more writes. Both have trade-offs.
- **Load shedding** — When every machine is already full, it is better to immediately say “sorry, try again” to 10% of requests than to slowly process 100% until all of them time out. Dropping on purpose, fast, protects the users you can still serve.

What to do with this: Make sure your app stores sessions in Redis or a database — not on the local disk — so any server can handle any user. When you need to handle more traffic, add more servers rather than endlessly upgrading one big one, and set your autoscaler to react to queue depth or request count, not just CPU, so new capacity arrives before the brownout starts.

What you'll learn: How a single web app turns into a fleet that survives traffic spikes and node death — the layers of load balancing (L4 vs L7), the algorithms that distribute requests, why “stateless” is the keystone trick that makes horizontal scaling possible, how databases scale differently from app servers, and how systems shed load *on purpose* instead of falling over. You'll leave able to reason about an autoscaling design under interview pressure or during a 3 a.m. incident.

Prerequisites: Read [01-foundations-and-estimation.md](#) (latency/throughput math, back-of-envelope), [05-caching-and-cdns.md](#) (externalized state, edge caching), and ideally [09-cap-pacelc-and-consistency.md](#) (why DB scaling is hard). Sharding here cross-references [08-data-partitioning-and-sharding.md](#).

1. The core problem: one box is never enough

A single server has a hard ceiling — CPU cores, RAM, NIC bandwidth, file descriptors. When demand exceeds it you have exactly two moves:

- **Scale up (vertical):** buy a bigger box.
- **Scale out (horizontal):** buy *more* boxes and spread the work.

Analogy. A popular restaurant with one chef can either hire a faster chef (vertical) or open more kitchens and route orders between them (horizontal). The faster chef is simple but eventually no chef is fast enough — and if that one chef gets sick, dinner stops. Multiple kitchens are more complex (who routes orders? what if a kitchen burns down mid-order?) but have no ceiling and survive a single failure.

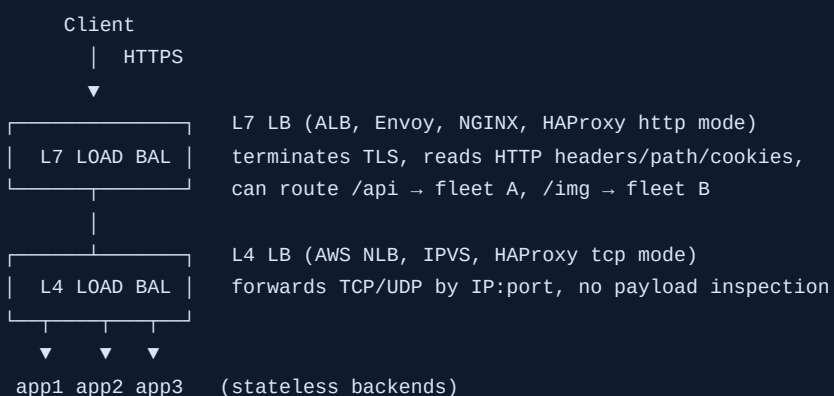
Vertical vs Horizontal

Dimension	Vertical (scale up)	Horizontal (scale out)
Mechanism	Bigger CPU/RAM/disk on one node	More nodes behind a balancer
Ceiling	Hard (largest instance, e.g. AWS u-24tb1 ~ 24 TB RAM, 448 vCPU)	Effectively unbounded
Fault tolerance	None — single point of failure	High — lose a node, keep serving
Cost curve	Super-linear (top-end hardware is priced like a luxury)	~Linear (commodity boxes)
Code complexity	Zero — app unchanged	Requires statelessness + LB + coordination
Latency	Lowest (no network hop, shared memory)	Adds a hop + distributed-systems problems
When to use	Databases, single-writer systems, early stage, latency-critical monoliths	Stateless web/API tiers, anything that must survive node loss

Rule of thumb: scale *up* the things that are hard to distribute (the primary database), scale *out* the things that are easy (stateless app servers). Most real systems do both. A 2× vertical bump buys you breathing room in an afternoon; horizontal is the long game.

2. Load balancing: the layers

A load balancer (LB) is the traffic cop in front of the fleet. The key distinction is *which layer of the network stack it inspects*.



L4 vs L7

	L4 (transport)	L7 (application)
Inspects	IP + TCP/UDP port	HTTP method, path, headers, cookies, TLS SNI
TLS	Pass-through (or termination on NLB w/ TLS listener)	Terminates TLS, re-encrypts to backend optionally
Routing smarts	Hash 5-tuple, sticky by IP	Path/host routing, header rules, cookie affinity, A/B, canary
Throughput	Very high, ~μs added latency, millions of conns	Higher CPU cost, more latency (parses every request)
Connection model	One client TCP conn often = one backend conn	Can multiplex; HTTP/2 connection coalescing
Examples	AWS NLB, Google Maglev, LVS/IPVS, F5	AWS ALB, Envoy, NGINX, HAProxy, Traefik
When to use	Raw TCP, gRPC at the edge, ultra-low latency, non-HTTP	HTTP APIs needing path routing, canaries, WAF, auth

Why this matters in practice: L7 lets you do *content-based* routing — send `/admin` to a hardened pool, `/designer` to GPU boxes, version-pin a canary by header. L4 can't see any of that; it only knows “packets to port 443.” Many stacks layer them: an L4 NLB for raw throughput/static IP, fronting an L7 Envoy mesh for smart routing. Google's **Maglev** paper describes an L4 software LB doing consistent hashing across a cluster, handling the full edge of Google with ECMP from the routers.

3. Balancing algorithms — and why “random” can beat “smart”

The LB must pick *which* backend gets each request. The naïve choices and the clever ones:

Algorithm	How it works	Strength	Weakness / failure mode
Round-robin	Cycle 1 → 2 → 3 → 1...	Trivial, even count distribution	Ignores request <i>cost</i> ; a slow box still gets 1/N
Weighted RR	Bigger boxes get more turns	Handles heterogeneous fleets	Static weights drift from reality
Least-connections	Send to backend with fewest in-flight	Adapts to slow/stuck backends	Needs accurate conn counts; herd onto a freshly-added empty node
Least-response-time / EWMA	Track exponentially-weighted moving avg latency, pick fastest	Reacts to real degradation (GC pause, hot disk)	Stale signal; noisy under bursty load
Consistent hashing	Hash key (user/URL) → fixed backend	Cache affinity, sticky w/o cookies, minimal reshuffle on scale	Hot keys overload one node; uneven without vnodes
Power of two choices (P2C)	Pick 2 backends at random, send to the less-loaded	Near-optimal balance, cheap, decentralized	Slightly worse than perfect global least-conn

The P2C insight (the staff-level answer)

Pure random assignment leaves the most-loaded server with $\approx \log n / \log \log n$ more load than average — noticeably lumpy. **Pick two at random and choose the less loaded**, and the max load collapses to $\approx \log \log n / \log 2$ — exponentially better, for almost no extra work and *no global coordination*. This “power of two choices” result (Mitzenmacher) is why modern proxies default to it.

```
Random:      max load ~ log n / log log n      (lumpy)
Power of 2:  max load ~ log log n              (nearly flat)
              ↳ one extra random pick → exponential improvement
```

Why not just always use global least-connections? Because at scale your LB tier is *itself* distributed — many LB instances. None has a perfect global view; sharing exact connection counts across them is a coordination cost and a stale-data trap. P2C needs only two local samples, so it degrades gracefully. NGINX (`random two least_conn`), Envoy (default LB policy with `choice_count`), HAProxy, and Netflix all use P2C variants. EWMA + P2C (“two random choices, pick lower EWMA latency”) is the Netflix/Finagle production sweet spot.

Consistent hashing deserves its own deep dive — see `08-data-partitioning-and-sharding.md` — but the load-balancing use is: route the *same* cache key / user to the *same* backend so its local cache stays warm, and when a backend is added/removed only K/N keys move instead of all of them. Add **virtual nodes** (each physical node owns many ring positions) to smooth out hot spots. **Bounded-load consistent hashing** (Google, 2017; used in Vimeo’s HAProxy) caps any node at $(1+\epsilon) \cdot \text{average}$ and overflows to the next ring node — fixing the hot-key problem.

4. Health checks: don't route to the dead

A backend can die silently — process up, app hung. The LB must *detect and eject* it.

```
LB —every 5s→ GET /healthz → backend
|
2 fails in a row → mark UNHEALTHY, stop routing
3 passes in a row → mark HEALTHY, route again (with slow-start)
```

Check type	What it proves	Cost	Trap
TCP connect	Port is open	Cheap	App can be hung but socket still accepts
HTTP shallow (/healthz → 200)	Process serving HTTP	Cheap	Doesn't prove DB/deps work
HTTP deep (checks DB, cache, queue)	Real readiness	Expensive	Cascading failure: DB blip → every backend reports unhealthy → LB ejects <i>all</i> of them → total outage

War story / design rule: keep **liveness** (am I running? — shallow) separate from **readiness** (can I serve traffic right now? — deeper, but *bounded*). Kubernetes formalizes this with `livenessProbe` vs `readinessProbe` vs `startupProbe`. Never let a shared-dependency health check eject the entire fleet — that turns a degraded dependency into a complete outage. Cap the blast radius: if >50% report unhealthy, many LBs (ALB “minimum healthy targets”, Envoy **panic threshold**, default 50%) **fail open** — they route to *all* backends anyway, on the theory that a half-broken fleet is better than zero capacity.

Slow-start: a freshly-healthy node shouldn't get full traffic instantly (cold caches, JIT not warmed, connection pools empty). NGINX/Envoy ramp it up linearly over a window (e.g. 30s).

5. Sticky sessions vs stateless design — the keystone

Here's the trick that makes horizontal scaling *actually work*.

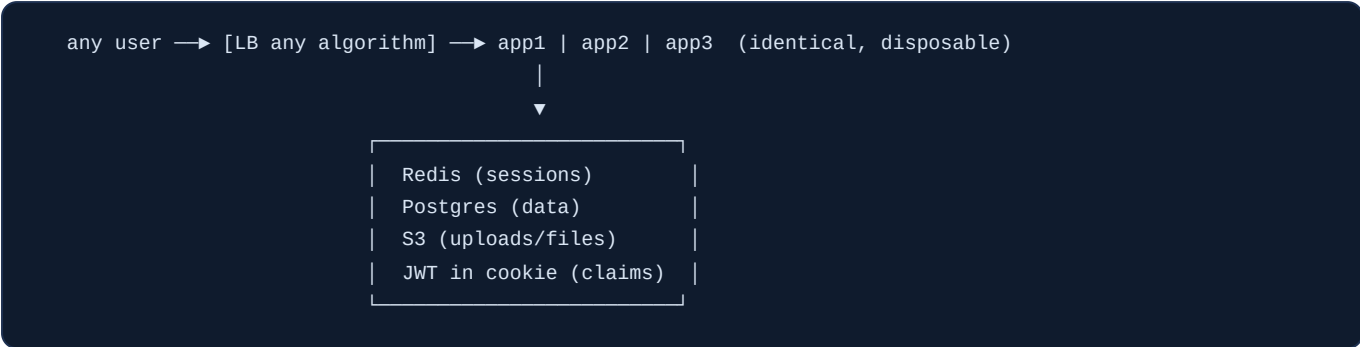
Stateful (sticky sessions): the LB pins a user to one backend (via cookie or source-IP hash) because that backend holds their session in local memory.

```
user A → app1 (session A in RAM) ← always routed to app1
user B → app2 (session B in RAM)
```

This is the easy path that bites you later:

- **No clean horizontal scale:** load is uneven (whales pin to one box), and you can't freely rebalance.
- **Deploy/restart loses sessions:** kill app1 and every pinned user is logged out / loses their cart.
- **Autoscaling fights affinity:** scale-in evicts users; scale-out gets ignored because existing users stay pinned.

Stateless + externalized state (the right way): backends hold *no* per-user state. Every request carries or can fetch everything it needs. Shared state lives in an external store.



Now any backend can serve any request → round-robin works, scale-in/out is free, a node can die mid-deploy and nobody notices.

	Sticky sessions	Stateless + external state
LB algorithm	Constrained (affinity)	Any (RR, least-conn, P2C)
Node death	Users lose session	Transparent
Deploys	Disruptive / drain-required	Rolling, painless
Autoscaling	Fights you	Clean
State store	None (in RAM)	Redis/DB/JWT (extra hop ~0.5–2ms)
When acceptable	Legacy apps, WebSocket connections (inherently stateful), short sessions	Default for all new HTTP/API tiers

Print-Flow-360 connection: Laravel's session driver is the concrete lever here. `SESSION_DRIVER=file` (local disk) is sticky-by-accident — it *forces* affinity. Move it to `redis / database` and the API tier becomes truly stateless and horizontally scalable. Sanctum token auth (a bearer token validated against the DB/cache, no server-side session) is even more stateless. JWTs push state into the *client* (claims in the token) so the server stores nothing — at the cost of revocation being hard (you can't un-issue a token; you need a short TTL + a denylist).

WebSockets are the honest exception: a live socket *is* a stateful connection bound to one node. You don't make the socket stateless — you externalize the *fan-out* (Redis pub/sub, a dedicated push tier like Pusher/Soketi/Ably) so any node can publish to any user's socket.

6. Autoscaling: reactive vs predictive

Horizontal scale is only useful if the fleet size *tracks demand*. Two philosophies:

	Reactive (target-tracking)	Predictive (forecast)
Trigger	Metric crosses threshold (CPU > 60%, RPS, queue depth)	ML/seasonal model forecasts load ahead of time
Latency to capacity	Slow — only acts <i>after</i> load arrives + boot time	Provisions <i>before</i> the spike
Handles flash spikes	Poorly (lag = brownout)	Better, if pattern is learnable
Handles novel spikes	Equally well (it's reactive)	Poorly (unseen pattern)
Complexity	Low (AWS Target Tracking, K8s HPA)	High (AWS Predictive Scaling, needs history)
Best for	Most workloads	Strong daily/weekly seasonality (e.g. business-hours SaaS)

The killer detail: the boot-time gap. Reactive scaling reacts to load you *already have*. If an instance takes 90s to boot + warm caches + register healthy, then for 90 seconds your overloaded fleet stays overloaded. Math: traffic doubles instantly, you're already at 70% CPU → you saturate immediately, but new capacity is 90s away. That 90s is a brownout (timeouts, 5xx). Mitigations:

- **Scale on a leading indicator**, not CPU. Queue depth or **requests-per-target** rises *before* CPU saturates. Best of all: scale on `request_concurrency / target_concurrency`.
- **Pre-warm / over-provision headroom** (run at 50% not 80% so you have slack while booting).
- **Predictive scaling** for known patterns (Black Friday, 9 a.m. login surge).
- **Faster cold starts**: pre-baked AMIs/images, snapshot-restore (AWS Lambda SnapStart), keep a warm pool.

Scale-in is dangerous too: terminating a node mid-request causes errors. This is where **connection draining** comes in (next section). Also beware **scaling oscillation (flapping)**: scale out → CPU drops → scale in → CPU spikes → scale out... Fix with a **cooldown period** and asymmetric thresholds (scale out fast at 60%, scale in slow at 30%).

7. Connection draining (graceful shutdown)

When you remove a node (deploy, scale-in, replace), in-flight requests must complete instead of being severed.

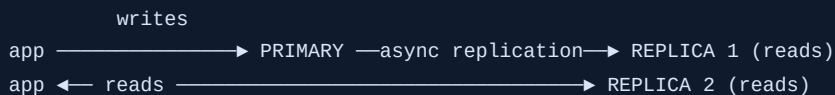
```
T0 LB: mark node "DRAINING" → stop sending NEW requests
T0 node still finishes the ~120 in-flight requests
T0+Δ all in-flight done (or drain timeout hit, e.g. 300s)
T0+Δ LB removes node → orchestrator sends SIGTERM
      app catches SIGTERM → finishes work → exits cleanly
      (if not exited by grace period → SIGKILL)
```

The subtle ordering bug everyone hits: the node must keep failing its **readiness** check (so the LB stops new traffic) but stay **alive** long enough to finish in-flight work *and* for the LB's deregistration to propagate. Common mistake: app exits on SIGTERM *immediately*, before the LB has noticed it's draining → the LB sends a request to a now-dead socket → connection reset → user-facing 502. Kubernetes handles this with a `preStop` hook (`sleep 15`) + `terminationGracePeriodSeconds` ; AWS target groups have a "deregistration delay" (default 300s). For HTTP keep-alive, send `Connection: close` during drain so clients reconnect to a live node.

8. Scaling the database — a different beast entirely

App servers scale out trivially once stateless. **The database is where statefulness is irreducible** — it *is* the state. Two orthogonal techniques:

Read replicas (scale reads)



- One primary takes all writes; replicas stream the write-ahead log and serve reads.
- **Scales reads, not writes** — the primary is still the write bottleneck.
- **Replication lag** is the gotcha: a user writes, immediately reads from a replica that hasn't caught up → "I just saved it, where did it go?" (read-your-own-writes violation). Fixes: route the writer's *own* reads to the primary for a few seconds, or use causal/session consistency tokens. See `09-cap-pacelc-and-consistency.md` .

Sharding / partitioning (scale writes)

Split the data itself across N independent primaries by a shard key.

```
shard key = hash(tenant_id) % N
tenant A,C → shard1 (own primary)
tenant B,D → shard2 (own primary)
```

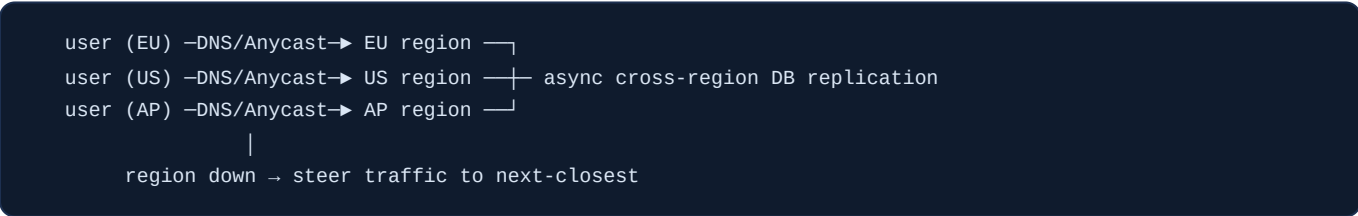
- **Scales writes** (each shard takes a fraction).
- Cost: cross-shard queries/joins/transactions become hard or impossible; rebalancing is painful; a bad shard key creates **hot shards**. Full treatment in `08-data-partitioning-and-sharding.md` .

Technique	Scales	Cost	When
Vertical (bigger DB box)	Reads + writes (to a ceiling)	\$\$\$, single failure	First move, always
Read replicas	Reads only	Replication lag, stale reads	Read-heavy (most web apps, ~10:1 read:write)
Sharding	Writes + storage	Lost cross-shard joins/txns, ops complexity	Write-bound or > single-box storage
CQRS / caching	Reads	Eventual consistency	Hot read paths (see 05-caching-and-cdns.md)

Real systems: PostgreSQL/MySQL do primary + read replicas natively. **Vitess** (YouTube) shards MySQL. **CockroachDB/Spanner** shard *and* replicate automatically with strong consistency (Spanner uses TrueTime + Paxos). **DynamoDB/Cassandra** shard by partition key with tunable consistency from day one. **Multi-tenant note (Print-Flow-360):** `stancl/tenancy` on a single shared Postgres means the `tenant_id` is effectively a logical shard key already — the natural physical-sharding axis if a single primary ever caps out is “tenants → shards.”

9. Global Server Load Balancing (GSLB)

Scaling across *regions*, not just nodes. The job: send each user to the nearest/healthiest region and fail over when one dies.



Mechanism	How users reach a region	Failover speed	Trap
GeoDNS / latency-based DNS (Route 53, NS1)	DNS returns region-specific IP	Slow (DNS TTL + client caching, minutes)	Clients ignore TTL; stale records send users to a dead region
Anycast (Cloudflare, Google)	Same IP announced from many sites; BGP routes to nearest	Fast (BGP reconverges)	Long-lived TCP can break if route flips mid-connection
Client-side / app steering	App fetches a region map, picks endpoint	Flexible, app-controlled	Requires smart client

The hard part of multi-region is never the routing — it’s the **data**. Cross-region replication is high-latency (~70–150ms US ↔ EU one way), so you choose: one global primary (simple, far writers are slow), regional primaries with conflict resolution (CRDTs / last-write-wins, eventual consistency), or a globally-consistent

store (Spanner/Cockroach paying latency for Paxos quorum across regions). This is a PACELC decision — see `09-cap-pacelc-and-consistency.md`.

10. Graceful degradation & load shedding

When demand exceeds *all* available capacity, you have two outcomes: **degrade on purpose, or collapse by accident**. A queue that grows unbounded doesn't "handle" overload — it adds latency until everything times out and the whole thing falls over (congestion collapse).

Load shedding = drop work you can't serve, fast, to protect the work you can. Better to serve 90% well and reject 10% immediately than to serve 100% so slowly that all of it times out.

```
incoming RPS → [admission control]
                | within capacity → serve
                | over capacity   → reject FAST with 503 + Retry-After
                |                               (cheap rejection, no queue buildup)
```

Techniques, from gentle to brutal:

Technique	What it does	Example
Graceful degradation	Turn off non-essential features under load	Drop personalization, serve cached/generic page, hide recommendations
Rate limiting	Cap per-client RPS	Token bucket per API key (see <code>06-...</code> if present)
Load shedding	Reject excess requests cheaply (503)	Envoy/Netflix concierge, prioritize by request criticality
Brownout	Reduce quality, not availability	Lower image res, shorter result lists
Circuit breaker	Stop calling a failing dependency, fail fast	Hystrix/resilience4j; open circuit after N failures
Backpressure	Push "slow down" upstream	Bounded queues, HTTP 429, reactive streams

Prioritized shedding (staff-level): not all requests are equal. Shed the cheap-to-lose first — drop a background prefetch before a checkout. Netflix and Google tag requests with criticality and shed low-criticality first. **Adaptive concurrency limits** (Netflix's `concurrency-limits`, Envoy's adaptive concurrency, Google's "Overload" chapter in the SRE book) measure latency and *automatically* lower the in-flight ceiling when latency rises — TCP-Vegas-style congestion control, but for requests. This beats a static limit because the right limit changes with backend health.

11. Common pitfalls / war stories

- **Deep health checks took down the fleet.** A 200ms DB blip made every backend's `/healthz` fail (it pinged the DB), the LB ejected 100% of targets, and a minor blip became a full outage. *Fix:* shallow liveness + bounded readiness + LB panic-threshold fail-open.
 - **The retry storm / thundering herd.** A backend slows; clients retry; retries triple the load; the backend dies; healthy backends inherit the load and die too — cascading collapse. *Fix:* exponential backoff **with jitter**, retry budgets (cap retries at e.g. 10% of requests), circuit breakers. AWS's “Timeouts, retries and backoff with jitter” is the canonical write-up.
 - **Sticky sessions made deploys a logout event.** `SESSION_DRIVER=file` + rolling deploy = every user logged out per node cycle. *Fix:* externalize sessions to Redis.
 - **Least-connections herded onto the new node.** A freshly-added empty backend has 0 connections, so least-conn dumped *all* new traffic onto it, overwhelming its cold caches. *Fix:* slow-start ramp.
 - **Autoscaling on CPU was always 90s late.** CPU is a lagging indicator; the brownout was over by the time capacity arrived. *Fix:* scale on requests-per-target / queue depth, keep headroom.
 - **Scale-in killed in-flight checkouts.** No drain → SIGKILL mid-request → 502s and double-charges. *Fix:* connection draining + SIGTERM grace + idempotency keys.
 - **Read replica showed stale data right after a write.** Classic read-your-writes bug. *Fix:* route post-write reads to primary or use session consistency.
 - **Unbounded queue = invisible outage.** “We never reject requests!” — instead every request takes 40s and times out. A bounded queue that sheds is healthier than an unbounded one that drowns.
 - **The LB became the SPOF.** One load balancer in front of a redundant fleet — and it died. *Fix:* redundant LBs (active-active via Anycast/ECMP, or active-passive with VIP failover).
-

Case Study: Google Maglev

The problem. By the early 2010s Google's edge had to terminate essentially *all* of the company's user-facing traffic — Search, Gmail, Maps, YouTube — at a single front door per cluster. That's millions of new connections per second, hundreds of Gbps per cluster, with hard requirements no hardware appliance could meet cheaply: it had to scale **horizontally on commodity servers** (not \$100k F5 boxes that scale *up* to a ceiling), survive any single machine dying without dropping live connections, and let Google add capacity by racking another identical box. The legacy answer — big hardware L4 load balancers in active/passive pairs — wasted the passive box, had a vertical ceiling, and turned the LB into the exact SPOF described in §11. Google's answer was **Maglev**: a distributed software L4 load balancer that runs as a normal Linux service on a fleet of regular servers.

This is the L4 tier from §2. Maglev is pure **L4** — it forwards packets by inspecting only the IP 5-tuple (src/dst IP, src/dst port, protocol). It never parses HTTP, never terminates TLS. That's the deliberate “raw TCP, ultra-low latency, millions of conns” column of the L4-vs-L7 table. The L7 smarts (path routing, TLS termination, canaries) happen *behind* it in the application/proxy tier. Maglev's only job is: take a packet for a service VIP (virtual IP) and pick a backend, fast, consistently, and survive failures.

How traffic reaches the fleet (the horizontal, stateless part of §1 and §5). Every Maglev machine in a cluster announces the *same* service VIPs to the router via BGP. The router uses **ECMP (equal-cost multi-path)** to spray incoming packets across *all* Maglev machines roughly evenly. There is no leader and no shared state between Maglev boxes — each is an identical, **stateless, disposable** node, exactly the “any node serves any request” model from §5. Add a machine → it announces the VIPs → ECMP starts handing it a share. Lose a machine → it stops announcing → ECMP reroutes. This is horizontal scaling of the *load-balancer tier itself*.



The hard part: connection affinity without shared state. Here’s the trap. ECMP routes by 5-tuple hash, but routes can change — a Maglev box is added, removed, or a link flaps — and ECMP may suddenly send an *existing* connection’s packets to a *different* Maglev machine than the one that saw its first packet. For a stateful TCP connection, if the new Maglev picks a *different* backend, the connection breaks. So every Maglev machine, with **no coordination**, must independently map the same connection to the same backend.

This is precisely the **consistent hashing for connection affinity** technique from §3 — but Google needed it *uniform* and *minimally disruptive*, so they built a variant called **Maglev hashing**. Instead of points on a ring, Maglev precomputes a fixed-size **lookup table** (e.g. 65,537 entries) where each backend “takes turns” claiming preferred slots until the table is full. The result is two properties that map straight onto §3’s consistent-hashing pitch:

- **Even spread:** every backend gets $\sim 1/N$ of the table (far more balanced than a naive hash ring, which is lumpy without virtual nodes).
- **Minimal disruption on change:** when a backend is added or removed, only $\approx 1/N$ of table entries move — the same “only K/N keys reshuffle” property that makes consistent hashing the right tool, here tuned for *fewer* disruptions than a ring at the cost of perfect minimality.

Because the table is **deterministic** from the backend set, every Maglev machine computes the **identical** table. So even if ECMP reroutes a live connection to a different Maglev box, that box hashes the 5-tuple and picks the *same* backend — connection survives. As a belt-and-suspenders measure each box also keeps a small local **connection-tracking** cache for connections it has seen, falling back to the consistent-hash table for anything new. This is the §3 insight (“route the same key to the same backend, and on membership change only K/N move”) applied to keep *TCP connections* — not cache keys — pinned.

Health checks and draining (§4 and §7). Maglev continuously **health-checks** each backend and recomputes the lookup table when the healthy set changes — a dead backend simply drops out of the table, which is §4’s “detect and eject the dead.” Draining a Maglev *machine* is graceful: it withdraws its BGP announcement so ECMP stops sending it new flows (the §7 “stop new traffic but stay alive to finish in-flight” pattern), then drains existing connections before going down. No connection is severed by a planned removal.

The trade-off they accepted. Maglev is **L4-only and stateless by design** — and they kept it that way on purpose. By refusing to do L7 (no HTTP parsing, no TLS termination, no per-request routing) and refusing shared connection state across machines, they bought raw throughput, trivial horizontal scale, and resilience. The price: Maglev *cannot* do content-based routing or guarantee perfect connection affinity under simultaneous backend churn — a small fraction of connections can still break if the backend set changes *while* ECMP also reshuffles. Google judged that occasional reset acceptable versus the cost of a coordinated, stateful, L7 design. They also accepted **kernel-bypass complexity** (a userspace packet path with a busy-polling NIC) to hit the latency target.

Real numbers. A single Maglev machine saturates a 10 Gbps NIC with small packets — roughly **line-rate**, on the order of millions of packets per second — by bypassing the Linux kernel networking stack and pinning a userspace forwarder to dedicated cores with batched packet processing. A cluster scales linearly: $N \text{ machines} \approx N \times \text{throughput}$, all behind the same VIPs via ECMP. It has served Google’s edge in production since ~2008, replacing hardware LBs while running on commodity servers.

Lessons

- **Push smarts down the stack when raw scale matters.** Keeping the front door strictly L4 and stateless is *why* it scales linearly on cheap boxes; do the L7 routing in a tier behind it.
- **Consistent hashing isn’t just for caches — it’s how a stateless fleet agrees without talking.** A deterministic hash lets independent nodes pick the same backend with zero coordination, the same property you’d use for cache affinity in §3.
- **ECMP + identical-announcing nodes is the horizontal-scaling pattern for the LB tier itself** — and it dissolves the “LB is the SPOF” war story from §11.
- **State the affinity trade-off explicitly:** Maglev accepts rare connection resets under simultaneous churn in exchange for no shared state. Knowing exactly what you gave up (perfect affinity) for what you got (statelessness) is the staff-level move.

12. Test yourself

1. **Why does “power of two choices” beat both pure-random and global least-connections in a large distributed LB tier?** (Hint: max-load math $\log \log n$ vs $\log n / \log \log n$; and least-conn needs a global view the distributed LBs don’t share.)
2. **Your `/healthz` checks the database. The DB has a 1-second hiccup. What happens, and how do you prevent it?** (Hint: every target fails → fleet ejected → outage; separate liveness/readiness + fail-open panic threshold.)

3. You set `SESSION_DRIVER=file` and added a second app server behind round-robin. Users report random logouts. Why? (Hint: session lives in one node's local disk; RR sends them elsewhere; externalize to Redis.)
 4. Traffic doubles instantly. You autoscale on CPU > 60% with a 90s boot time. Walk through the brownout, then fix it. (Hint: reactive lag + cold start; scale on leading indicator + headroom + predictive/warm pool.)
 5. When does sharding help and read replicas don't? (Hint: write-bound or storage-bound; replicas only scale reads, primary still takes all writes.)
 6. A backend slows down. Within 30s the whole fleet is down. Name the mechanism and three mitigations. (Hint: retry storm/cascading failure; backoff+jitter, retry budgets, circuit breakers, load shedding.)
 7. During scale-in, users get 502s and a few double-charges. Two fixes — one infra, one app. (Hint: connection draining + SIGTERM grace; idempotency keys.)
 8. Multi-region: GeoDNS vs Anycast for failover — which fails over faster and why is data still the hard part? (Hint: Anycast reconverges via BGP in seconds, DNS waits on TTL/client caching; cross-region replication latency + consistency = PACELC trade-off.)
-

13. Further reading

- **DDIA (Kleppmann)** — Ch. 5 (Replication: leaders, replication lag, read-your-writes), Ch. 6 (Partitioning/Sharding), Ch. 1 (Reliability/Scalability/Maintainability). The single best reference for §8.
- **Google SRE Book** — “Handling Overload” and “Addressing Cascading Failures” chapters (load shedding, adaptive concurrency, retry budgets). Free online.
- **Mitzenmacher**, “The Power of Two Choices in Randomized Load Balancing” (1996/2001) — the P2C result in §3.
- **Eisenbud et al.**, “Maglev: A Fast and Reliable Software Network Load Balancer” (NSDI 2016) — Google's L4 LB + consistent hashing (§2/§3).
- **Mirrokn**i et al., “Consistent Hashing with Bounded Loads” (Google, 2017) — fixes hot keys (§3).
- **AWS Builders' Library** — “Timeouts, retries, and backoff with jitter” and “Using load shedding to avoid overload” — the canonical war-story writeups for §10/§11.
- **Netflix concurrency-limits** (GitHub) and **Envoy docs** — adaptive concurrency, panic threshold, P2C/EWMA LB policies (§3/§4/§10).
- **Kubernetes docs** — liveness/readiness/startup probes, `terminationGracePeriodSeconds`, `preStop` (§4/§7).
- **Sibling modules:** `01-foundations-and-estimation.md`, `05-caching-and-cdns.md`, `08-data-partitioning-and-sharding.md`, `09-cap-pacelc-and-consistency.md`.

08 — Replication & Partitioning (Sharding)

Source: [readme/research/system-design/08-replication-and-partitioning.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains two big ideas that every large website relies on: keeping copies of your data on multiple computers (so the site stays up if one breaks), and splitting a huge amount of data across many computers (so no single machine gets overwhelmed). Understanding these ideas helps you reason about why databases sometimes show you old data, lose a write, or slow down — and what engineers do to fix it.

The main parts explained simply:

- **Replication (making copies)** — Storing the same data on several machines at once. If one machine crashes, the others still have the data and the site keeps running. The trade-off: keeping all copies perfectly in sync slows things down, so most systems let copies fall a little behind.
- **Partitioning / Sharding (splitting data)** — When your data is too big for one machine, you slice it into pieces and put each piece on a different machine. This is how companies like Amazon handle billions of records.
- **Single-leader replication** — One machine is in charge of all writes; the others just follow and copy. Simple and safe, but if the leader dies someone must take over — and during that handover, recent saves can be lost.
- **Multi-leader replication** — Two or more machines each accept writes, then sync with each other. Useful when you have offices in different countries, but messy: if both machines change the same record at the same time, you have a conflict to resolve.
- **Leaderless replication (quorums)** — No single boss. A write is saved to several machines at once, and a read checks several machines and picks the freshest answer. Amazon's shopping cart was built this way so it could never refuse a write, even during outages.
- **Replication lag problems** — When copies fall behind the leader, a user might save something, refresh the page, and see the old value — as if the save never happened. This is exactly the “saved but it's gone” bug. The fix is to send that user's reads to the main machine for a short time after they write.
- **Hot spots** — If data is split by date or a sequential number, all new writes pile onto one machine while the rest sit idle. The fix is to use a smarter key that spreads writes evenly.
- **Consistent hashing** — A clever way to split data across machines so that adding or removing a machine only moves a small fraction of data, not everything at once.
- **Conflict resolution** — When two machines accept conflicting writes, someone must decide which one wins. The lazy approach (latest timestamp wins) silently throws away data when clocks differ by even a few milliseconds. Better approaches track which write came first logically, or merge both versions.
- **Secondary indexes across partitions** — Searching by something other than the main key (e.g. “find all red products”) may require asking every machine and combining results — which is slow at scale.

What to do with this:

When adding read replicas to a database, always route a user's reads back to the main database for a moment after they save — otherwise users will see “stale” data and think their save was lost. When splitting data across machines, pick a partition key that spreads writes evenly, not one that funnels all new writes to a single machine.

What you'll learn: How to keep multiple copies of data correct across machines (replication) and how to split a dataset too big for one machine across many machines (partitioning), plus the failure modes — stale reads, write conflicts, hot shards, rebalancing storms — that turn a clean diagram into a 3 a.m. incident. By the end you should be able to reason about Dynamo-style quorums, Postgres streaming replication, and Cassandra's ring the way an interviewer or an on-call engineer would.

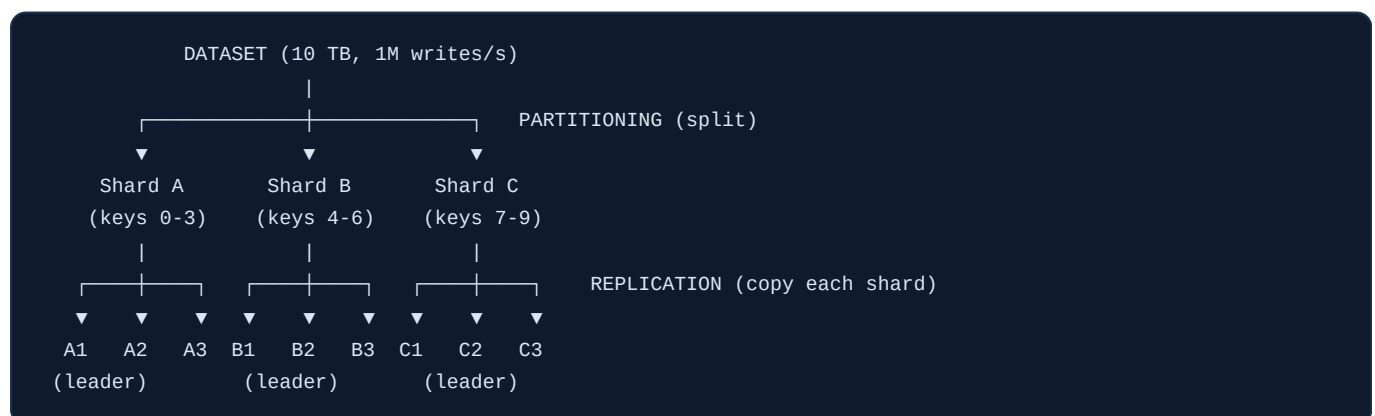
Prerequisites: Read [02-data-models-and-storage.md](#) (how a single node stores data) and [09-cap-pacelc-and-consistency-models.md](#) (consistency vocabulary — linearizability, causal, eventual). This module assumes you know what a leader, a commit, and a write-ahead log are. Replication conflict semantics lean on [10-distributed-consensus-and-time.md](#) (vector clocks, Lamport time).

0. Two orthogonal axes

These are different problems people constantly conflate:

- **Replication** = keep the *same* data on *multiple* nodes. Goals: availability (survive a node death), read throughput (serve reads from many copies), latency (serve from a geographically near copy).
- **Partitioning / sharding** = split *different* data across *multiple* nodes. Goal: scale beyond one machine's disk/RAM/CPU.

Real systems do **both**: partition the keyspace into N shards, then replicate each shard R times.



A failure of A2 loses one replica of shard A (no data loss). A failure of *all* of shard C's replicas loses 1/3 of the dataset. This is why replica placement (rack/AZ awareness) matters: never put all replicas of a shard in one fault domain.

PART 1 — REPLICATION

1.1 Single-leader (master–slave / primary–replica)

Intuition: One node is the boss. All writes go to the boss; the boss tells everyone else what changed. Like a teacher dictating notes — only the teacher writes the canonical version, students copy.

Mechanics: The leader appends every write to a *replication log* and ships it to followers. The log can be:

Log type	What ships	Pro	Con	Used by
Statement-based	The SQL string	Compact	Non-determinism (<code>NOW()</code> , <code>RAND()</code> , autoincrement) breaks replicas	Old MySQL <code>statement</code> mode
Write-ahead log (WAL)	Physical disk pages/byte changes	Exact, simple	Coupled to storage engine version → no rolling upgrades	Postgres streaming replication
Logical (row-based)	Logical row changes (this row, these new values)	Decoupled from engine, allows version skew + CDC	More bytes	MySQL <code>row</code> binlog, Postgres logical replication

Real: **Postgres** ships WAL records to standbys over a TCP stream (`walsender` → `walreceiver`).

MySQL ships binlog events. This is the workhorse of 90% of OLTP web apps — including most Laravel deployments (one primary, read replicas behind a connection proxy).

Sync vs async — the durability/latency knife-edge

The single most consequential replication choice. When does the leader tell the client “committed”?

Mode	Leader waits for...	Durability on leader failure	Write latency	Availability
Fully sync	ALL followers ack	Zero data loss	Worst (bounded by slowest follower; one slow node stalls <i>all</i> writes)	Worst (one dead follower blocks writes)
Semi-sync	≥1 follower acks	No loss if leader+that follower don't both die	Medium	Medium
Async	nobody — ack on local commit	Loss window = unreplicated tail of the log	Best	Best

Concrete: Async cross-region replication has a lag of typically 10–500 ms (network RTT dominated). If your leader in us-east dies, you can lose every write from the last ~lag-window. For an order checkout that's unacceptable; for a “last seen” timestamp it's fine. **Semi-synchronous** (one sync follower in the *same AZ* + async to other regions) is the standard production compromise: sub-millisecond local sync ack,

no single-machine data loss, async fan-out for geo. See [09-cap-pacelc-and-consistency-models.md](#) for why this is a PACELC “else latency” decision.

Failover — where the bodies are buried

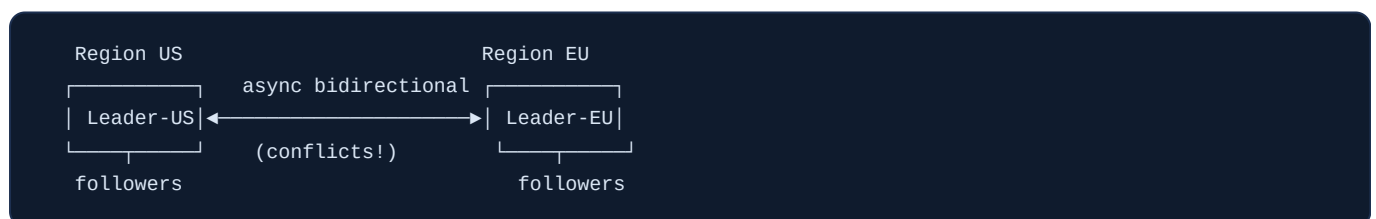
Leader dies → promote a follower. The hard parts:

1. **Detecting death** — timeout-based, so you tune the false-positive/MTTR trade-off. Too short → flapping; too long → downtime.
2. **Choosing the new leader** — pick the most up-to-date follower (smallest replication lag) to minimize loss. Needs consensus ([10-distributed-consensus-and-time.md](#)) or it splits.
3. **Lost writes** — async writes not yet replicated are *gone*. GitHub’s famous 2012 outage: a promoted replica was behind, and when the old leader rejoined, its un-replicated writes were discarded — autoincrement IDs got reused and pointed at the wrong rows.
4. **Split brain** — two nodes both think they’re leader and both accept writes. Fence the old leader (STONITH — “shoot the other node in the head”) or use a lease/epoch number so the cluster rejects writes from a stale leader.

1.2 Multi-leader (active–active)

Intuition: Multiple bosses, usually one per region. Each accepts writes locally and they sync to each other. Like two editors editing a shared doc offline, then merging.

When: Multi-datacenter (write locally, low latency, survive a whole DC outage), offline clients (your phone’s calendar is a leader; it syncs later), collaborative editing.



The catch: write conflicts. Two leaders concurrently write the same key with no global order. Now you must resolve. See §1.4. This is why multi-leader is a “use only if you must” topology — the conflict tax is real. Avoid it when single-leader-per-region with home-region routing works.

1.3 Leaderless (Dynamo-style)

Intuition: No boss at all. The client (or a coordinator) writes to *several* nodes and reads from *several* nodes, and uses overlap to stay correct. Like asking 3 of your 5 friends what time the party is and trusting the majority/newest answer.

Real: **Amazon Dynamo** (2007 paper), **Cassandra**, **Riak**, **ScyllaDB**, **Voldemort**.

Quorums — the core formula

- **N** = replication factor (copies per key)
- **W** = nodes that must ack a write
- **R** = nodes that must respond to a read

If $R + W > N$, every read overlaps every write by ≥ 1 node → you'll see the latest write *among acked nodes*.

```
N=3, W=2, R=2. R+W=4 > 3 ✓  
Write to nodes {1,2}. Read from nodes {2,3}.  
Overlap = node 2 → read sees the write.
```

Tunable knobs (per request in Cassandra via consistency level):

Choice	Effect	When
W=N, R=1	Fast reads, slow/fragile writes	Read-heavy, rare writes
W=1, R=N	Fast writes, slow reads	Write-heavy logging
W=R=quorum ($\lfloor N/2 \rfloor + 1$)	Balanced strong-ish	Default
$R+W \leq N$	No overlap guarantee → may read stale	Max availability, eventual-consistency OK

Crucial caveat: quorum gives you *staleness bounds*, **not linearizability**. Concurrent reads during a write can see different values; writes can be lost on certain failure interleavings; sloppy quorums (below) break the overlap entirely. For real linearizability you need consensus (Raft/Paxos), not quorums — see 09 and 10.

Anti-entropy: how stale replicas catch up

A node that was down or missed a write must heal:

- **Read repair** — on a read, if the coordinator sees one replica is stale, it writes the fresh value back. Fixes hot keys passively.
- **Hinted handoff** — if a target node is down at write time, another node holds a “hint” and delivers it when the node returns. (This + cross-DC reads is the “sloppy quorum”: the W acks may come from *non-home* nodes, so $R+W > N$ no longer guarantees overlap — availability over consistency.)
- **Merkle trees** — periodic background full-replica comparison. Hash tree over key ranges; compare root hashes, recurse only into differing subtrees → cheap diff of huge datasets. Cassandra `nodetool repair` uses these.

1.4 Replication lag & its anomalies

Async replication means followers trail the leader. Three classic read anomalies and their fixes:

Anomaly	Symptom	Guarantee needed	Fix
Read-your-own-writes	User posts a comment, refreshes, it's gone (read hit a lagging replica)	Read-your-writes	Route a user's reads to the leader for N seconds after their write; or track the user's last-write log position and only read replicas caught up past it
Monotonic reads	Time appears to go backwards — refresh shows newer data, refresh again shows older (two replicas at different lag)	Monotonic reads	Pin each user to one replica (e.g. hash userID → replica)
Consistent prefix reads	You see an answer before its question (cross-partition causal violation)	Consistent prefix	Keep causally-related writes in one partition, or track causal deps

Concrete bug class for this codebase: Laravel apps reading from a read replica after a write are a textbook read-your-writes trap. If a store owner edits a product price and the next request reads a lagging replica, they'll see the *old* price and think “Saved” was a lie — exactly the silent-lie / fake-success bug class this project obsesses over. The fix is “sticky to primary after write.” (We currently run single-DB, so this is latent, not live — but it's the first thing that breaks when read replicas get added.)

Conflict resolution strategies (multi-leader & leaderless)

Strategy	How it picks a winner	Loses data?	Notes / real systems
Last-Write-Wins (LWW)	Highest timestamp	Yes — silently discards concurrent writes; clock skew picks arbitrary winner	Cassandra default. Simple, dangerous. Two writes in the same millisecond → coin flip.
Vector clocks / version vectors	Detect <i>concurrent</i> vs <i>causal</i> ; surface conflicts (siblings) to the app	No (defers to app)	Riak returns siblings; DynamoDB used vector clocks in the paper
CRDTs (Conflict-free Replicated Data Types)	Math guarantees merge converges (G-Counter, OR-Set, sequence CRDTs)	No (semantic merge)	Riak data types, Redis CRDT (Enterprise), Automerge/Yjs for collab editing
Application merge function	App-specific (e.g. union of shopping carts)	No	Dynamo's shopping cart: union carts, “deleted” item resurrects — known wart

Why LWW is a trap: it relies on synchronized clocks. NTP skew is commonly 10–100 ms across machines; a “later” wall-clock timestamp can belong to an *earlier* write. Use logical clocks (Lamport/vector) when correctness matters — see [10-distributed-consensus-and-time.md](#). CRDTs are the modern answer for mergeable state; LWW is fine only for truly last-value-only fields where loss is acceptable (e.g. a presence “online” flag).

PART 2 — PARTITIONING (SHARDING)

Goal: no single node holds the whole dataset. Each partition is a mini-database. The two questions: **how do I assign keys to partitions** and **how do I find the partition for a key**.

2.1 Range partitioning vs hash partitioning

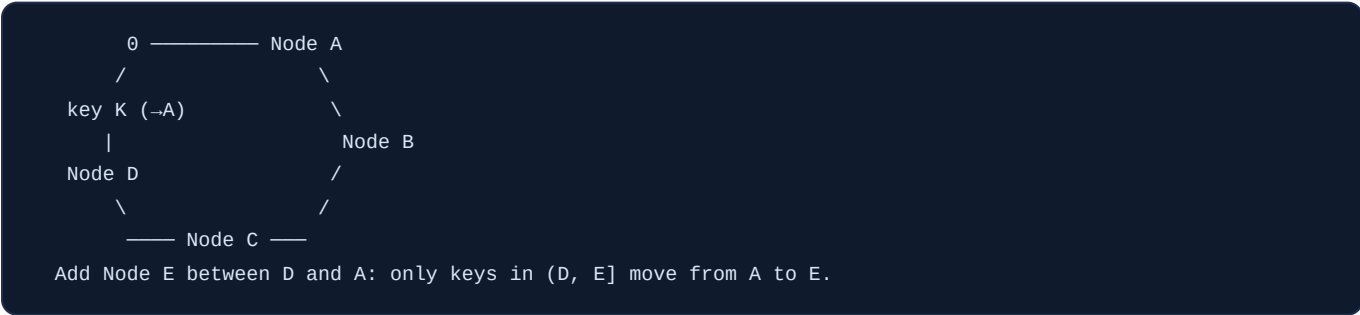
	Range partitioning	Hash partitioning
Key → partition	Sorted contiguous ranges (a–f, g–m, ...)	<code>hash(key) mod buckets</code>
Range scans	Efficient (<code>WHERE ts BETWEEN</code>) — contiguous	Scatter-gather across all partitions
Hot spots	Severe for sequential keys (timestamps, autoincrement) — all writes hit the last partition	Spread evenly by design
Rebalancing	Split/merge ranges dynamically	Harder (mod N moves everything)
Real systems	HBase, Bigtable, MongoDB (ranged), Spanner	Cassandra, DynamoDB, MongoDB (hashed)

The timestamp hot-spot war story: partition by `created_at` and 100% of your write traffic lands on the newest partition (today’s range). Old partitions are read-only and idle; one node melts. Fix: **compound key** — prefix with something high-cardinality (`sensorID` then `timestamp`). Now writes spread across sensors, and you can still range-scan within one sensor. DynamoDB’s “use a high-cardinality partition key, timestamp as sort key” guidance is exactly this.

2.2 The `hash mod N` disaster → consistent hashing

Naive `hash(key) mod N` : add or remove one node ($N \rightarrow N+1$) and **almost every key’s mod result changes** → ~all data must move. Catastrophic.

Consistent hashing: map both keys *and* nodes onto a ring (hash space $0 \dots 2^{32}$). A key belongs to the first node clockwise. Add a node → only the keys between it and its predecessor move (~1/N of data). Remove a node → its keys go to the next node only.



Problem: with few real nodes, the ring is lumpy — one node randomly owns a huge arc → imbalance.
Solution: virtual nodes (vnodes). Each physical node claims many (e.g. 256) points on the ring. Now

load averages out, and a dead node's load spreads across *all* survivors (not just its one neighbor), and a powerful node can own *more* vnodes (heterogeneous hardware). Cassandra uses 256 vnodes/node by default; Dynamo used vnodes ("tokens"); Riak too.

Note: Dynamo's actual scheme is "consistent hashing + vnodes," but many modern systems (DynamoDB managed, MongoDB) use **fixed hash buckets / chunks** that are reassigned, decoupling partition count from node count — same goal (cheap rebalance), different mechanism.

Scheme	Add-node data movement	Balance	Used by
hash mod N	~100% (catastrophic)	Perfect (when static)	naïve only
Consistent hashing (no vnodes)	~1/N	Poor (lumpy)	early Dynamo
Consistent hashing + vnodes	~1/N, spread over all nodes	Good	Cassandra, Riak
Fixed # of partitions, reassign	move whole partitions	Good, predictable	Elasticsearch, Kafka, Couchbase
Dynamic split/merge	move on threshold	Good, adaptive	HBase, MongoDB, Bigtable

2.3 Rebalancing — the dangerous part

Moving data is expensive and concurrent with serving traffic.

- **Fixed partition count (e.g. 1000 partitions for a 10-node cluster):** create *more* partitions than nodes up front; moving a node just reassigns whole partitions. Simple, no re-hashing. Elasticsearch/Kafka do this. Downside: you must guess the max scale (partitions are roughly immutable). Too many tiny partitions = overhead; too few = can't scale out.
- **Dynamic (split when partition exceeds size threshold, ~10 GB in HBase):** adapts to data volume; only the split partition moves. Risk: a cold cluster starts with one partition → no parallelism until first split (HBase "pre-splitting" mitigates).
- **Automatic vs manual:** fully automatic rebalancing + automatic failure detection is a feedback loop from hell — a node looks slow → rebalance away from it → more load on it → looks slower → cascade. Most prod systems keep a **human in the loop** for rebalancing (propose, operator approves) even if failover is automatic.

2.4 Request routing — "which node has key K?"

Three architectures:

(a) Routing tier / proxy
client → router → node
(Vitess, mongos,
Twemproxy)

(b) Smart client
client(map) → node
(Cassandra driver
token-aware)

(c) Any-node + forward
client → node → (forward)
(Cassandra coordinator;
Dynamo any-node)

Approach	Where routing logic lives	Pro	Con	Real
Routing tier	Dedicated proxy	Dumb clients; central control	Extra hop + a thing to scale/operate	Vitess (<code>vtgate</code>), Mongo <code>mongos</code> , Twemproxy
Partition-aware client	Driver holds the map	One fewer hop	Every client must learn topology	Cassandra token-aware driver, Kafka client
Any node forwards	Cluster itself	Simplest client	Internal redirect hop	Cassandra coordinator, Dynamo

The hard part is **keeping the map current as rebalancing happens**. Solution: a coordination service holds the authoritative routing table — **ZooKeeper** (HBase, Kafka pre-KRaft, older Vitess), **etcd**, or a **gossip protocol** where nodes spread topology peer-to-peer (Cassandra, Dynamo). Gossip = no central dependency but eventually-consistent topology view; ZooKeeper = strongly-consistent map but an extra critical dependency. See [10](#) for gossip vs consensus.

2.5 Secondary indexes across partitions

The data is partitioned by *primary* key. But you query by something else (`WHERE color='red'`). Where does the index live?

	Local (document-partitioned) index	Global (term-partitioned) index
Index lives	On each partition, covering only its own docs	Partitioned independently, by the <i>indexed term</i>
Write	Cheap — one partition	Expensive — index entry may be on a <i>different</i> partition (distributed write, often async)
Read by secondary key	Scatter-gather — query ALL partitions, merge (tail-latency killer)	Read one partition (the one owning that term)
Real	Cassandra secondary index, Elasticsearch (per-shard), MongoDB	DynamoDB Global Secondary Index, Riak search

Scatter-gather tail-latency math: if a single-partition query is $p_{99}=10$ ms, a scatter-gather across 100 partitions must wait for the *slowest* of 100 → its latency is governed by the $p_{99.99}$ of a single node. With independent partitions, $P(\text{all 100 fast}) = 0.99^{100} \approx 0.366$ — so **~63% of scatter-gather queries hit at least one slow partition**. This is why “just add a secondary index” doesn’t scale, and why global indexes (or denormalized query-specific tables, the Cassandra way) exist. DynamoDB’s GSI is async, so reads can be *stale* relative to the base table — a real correctness footgun.

3. Common pitfalls / war stories

1. **LWW silent data loss.** Two clients write the same Cassandra row “simultaneously” (within clock skew); one write vanishes with no error. Engineers debug for days because nothing logged an error. Fix: don’t use LWW for non-idempotent accumulating data; use counters/CRDTs or single-leader for that field.
2. **Reading from a replica right after writing.** The classic read-your-writes failure (§1.4). Manifests as “I saved it but it’s not there.” Pin to primary post-write.
3. **The celebrity / hot-key problem.** One key (Taylor Swift’s user row, a viral product) gets $10^6\times$ the traffic. Even perfect hashing can’t split a *single* key. Fix: app-level key splitting (`celebrityID#0..#99` random suffix on write, fan-in on read), per-key caching, or a dedicated replica for that key. DynamoDB added “adaptive capacity” partly for this.
4. **Cross-partition transactions you didn’t know you had.** A “transfer money between two accounts” that land on different shards now needs 2-phase commit or a saga (`11-distributed-transactions...md`). People shard, then discover their core invariant spans shards. *Shard along the transaction boundary* (e.g. by tenant/customer so all their data co-locates).
5. **Rebalancing during peak.** Auto-rebalance kicks in under load and the data-movement traffic competes with user traffic → cascading slowdown. Throttle rebalance bandwidth; schedule manual ones off-peak.
6. **Cross-AZ sync replication tax.** Someone sets `synchronous_commit=on` with a standby in another region → every write now pays a 60–80 ms cross-region RTT and write throughput collapses. Sync within an AZ, async across.
7. **Resharding without enough headroom.** Splitting a 10 TB shard while it’s at 95% CPU. There’s no spare capacity to copy data. Add capacity *before* you need to reshard.

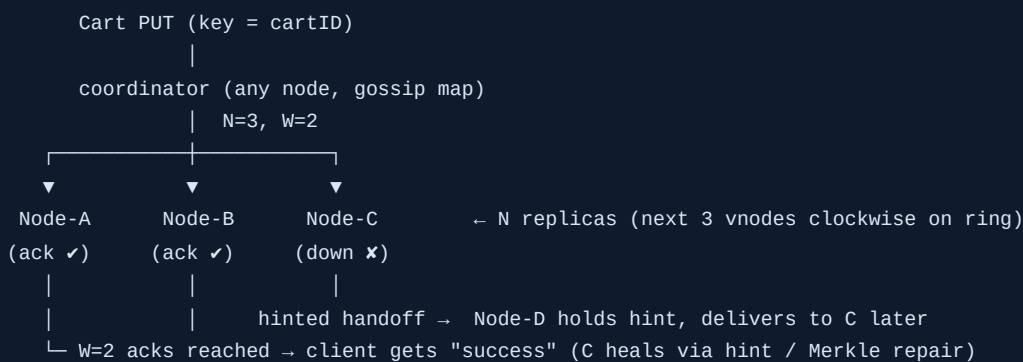
Case Study: Amazon Dynamo (the 2007 paper)

The problem. By the mid-2000s Amazon’s e-commerce platform ran on thousands of nodes across multiple datacenters, and its core services (the shopping cart, session state, the seller catalog, “best-seller” lists) were being hammered by **tens of millions of requests at peak shopping events** with strict SLAs stated at the **99.9th percentile, around a few hundred milliseconds**. The classic relational/single-leader stack kept failing the *availability* bar: a leader failover or a network partition meant writes were rejected, and “you can’t add this item to your cart” during the holiday rush is direct lost revenue. Amazon’s hard requirement was the opposite of most databases: **the write must always succeed**, even during failures, even at the cost of returning a slightly stale read. That requirement is what produced Dynamo.

Leaderless replication (§1.3) was the whole point. Dynamo has no boss — exactly the topology from the leaderless section. There is no leader to fail over, so there is no failover-induced write outage. A client request hits any node, which acts as a **coordinator** (the “any-node + forward” routing from §2.4, gossip-backed topology, no ZooKeeper dependency). The coordinator writes to N replicas and the cart stays writable as long as *some* nodes are reachable.

Consistent hashing with virtual nodes (§2.2). Keys and nodes are placed on a ring; a key's N replicas are the N distinct physical nodes walking clockwise from its hash. Dynamo uses **vnodes** (“tokens”) so that adding a node pulls $\sim 1/N$ of the data spread across *all* survivors rather than dumping it on one neighbor, and so beefier hardware can own more tokens. This is precisely the “consistent hashing + vnodes” row of the §2.2 table — and Amazon’s motivation (cheap, incremental, no-downtime capacity changes during traffic growth) is the exact reason the section says vnodes beat a plain ring.

$R + W > N$ quorums (§1.3), but tunable. Dynamo exposes N , R , W per service. The canonical config was $N=3$, $W=2$, $R=2$ — $R+W=4 > 3$, the overlap formula from the quorum section, giving bounded staleness for cart-class data. Write-heavy or “always-writable” services lowered W ; read-mostly services raised R . Crucially Dynamo also uses **sloppy quorums + hinted handoff** (§1.4 anti-entropy): if a home replica is down, the W acks may come from the *next available* nodes, which hold a “hint” and forward it later. This deliberately **breaks the $R+W>N$ overlap guarantee** to keep writing during a partition — the textbook “availability over consistency” sloppy-quorum trade the module warns about.



Conflict resolution with vector clocks (§1.4). Because anyone can write anywhere with no global order, concurrent writes to the same cart happen. LWW would silently drop one (and clock skew makes “latest” a lie — see the §1.4 LWW trap). Dynamo instead attaches a **vector clock** $[(node, counter), \dots]$ to every version. On read, the coordinator compares clocks: if one descends from the other it’s stale and discarded; if they’re **concurrent** it returns *both versions as siblings* to the application. The shopping cart’s app-level **merge is a union** of the carts — which is exactly the “application merge function” row of the §1.4 table, including its famous wart: a deleted item can resurrect because a union can’t tell “never added” from “added then removed.”

Anti-entropy to heal stale replicas (§1.4). Down or lagging nodes catch up via the same three mechanisms the module lists: **read repair** (coordinator pushes the freshest version back on a read), **hinted handoff** (above), and **Merkle trees** for periodic background range-by-range reconciliation between replicas without shipping whole datasets.

The key trade-off they accepted. Dynamo traded **strong consistency for always-on availability and low tail latency**. It is explicitly *not* linearizable — exactly the §1.3 caveat that “quorum gives staleness bounds, not linearizability.” Amazon accepted **stale reads, sibling versions the app must merge, and resurrected cart items** because, for a cart, showing a slightly old cart or re-adding a deleted item is recoverable, whereas *refusing the write is lost money*. They pushed conflict resolution up to the application instead of solving it in the datastore.

Results. Dynamo met its **99.9th-percentile latency SLA (single-digit-to-low-hundreds of milliseconds)** while staying writable through node failures, disk failures, and network partitions during peak holiday load — the “never reject a customer write” goal was achieved. The design directly seeded **Cassandra, Riak, Voldemort, and ScyllaDB**, and later the managed **Amazon DynamoDB** (which swapped vector-clock siblings for simpler server-side resolution and moved to fixed partitions, per the §2.2 note).

Lessons

- **Pick your CAP corner from the business, not the textbook.** Dynamo chose A over C because a rejected cart write costs revenue; a stale cart doesn't. Let the cost of *each* failure mode drive R/W/N and the conflict strategy — there's no universally “correct” consistency level.
- **R+W>N is a knob, and sloppy quorums quietly turn it off.** The clean overlap math holds only without hinted handoff; the moment you favor availability during partitions, you've accepted reading stale data. Know which mode you're actually in.
- **If you can't avoid conflicts, surface them — don't let LWW eat data.** Vector clocks + app-level merge keep concurrent writes; last-write-wins silently destroys them under clock skew. Choose siblings/CRDTs when the data accumulates.
- **Vnodes are what make leaderless clusters operable.** Spreading each node over many ring tokens is the difference between a rebalance that melts one neighbor and one that smoothly drains across the whole cluster.

4. Test yourself

1. You run $N=5$. What R and W give the strongest staleness guarantee while still tolerating one node being down for *both* reads and writes? (*Hint: need $R \leq 4$ and $W \leq 4$ to tolerate 1 down, and $R+W > 5$. Try $W=3, R=3$.*)
2. Why does `hash(key) mod N` force a near-total reshuffle when N changes, but consistent hashing only moves $\sim 1/N$? (*Hint: mod changes the bucket for almost every key; the ring only reassigns keys between adjacent points.*)
3. A user reports “I posted, refreshed, my post vanished, refreshed again, it's back.” Which two replication guarantees are being violated and what's the fix? (*Hint: read-your-writes + monotonic reads; pin user reads to one sufficiently-caught-up replica/primary.*)
4. Why does $R+W > N$ **not** give you linearizability? (*Hint: concurrent writes have no total order; sloppy quorums + read repair timing; failure interleavings can lose writes. Quorum bounds staleness, doesn't serialize.*)
5. You partition events by `created_at`. Writes are crushing one node. Diagnose and redesign the key. (*Hint: sequential hot spot; compound key high-cardinality-prefix + timestamp sort key.*)
6. When is multi-leader replication worth its conflict cost, and when is single-leader-per-region with home routing strictly better? (*Hint: offline clients / true active-active multi-write regions vs. users that mostly write in one region.*)
7. Why is a global secondary index a distributed write but a local one isn't — and what does that cost you on reads? (*Hint: global index entry lives on the term's partition, not the doc's; local index reads scatter-*

gather all partitions.)

8. Estimate the fraction of 50-way scatter-gather queries that hit a slow partition if each partition is slow 0.5% of the time. (*Hint: $1 - 0.995^{50} \approx 22\%$.*)
-

5. Further reading

- **DDIA** (Kleppmann), Chapter 5 (Replication) and Chapter 6 (Partitioning) — the canonical treatment; this module is a compressed, war-storied version of those two chapters.
- **Dynamo: Amazon’s Highly Available Key-value Store** (DeCandia et al., SOSP 2007) — consistent hashing + vnodes, vector clocks, hinted handoff, read repair, sloppy quorums.
- **Cassandra** docs — tunable consistency levels, `nodetool repair`, vnodes, gossip.
- **PostgreSQL** docs — “High Availability, Load Balancing, and Replication” (streaming + logical replication, `synchronous_commit`).
- **Bigtable** (Chang et al., OSDI 2006) and **Spanner** (Corbett et al., OSDI 2012) — range partitioning, tablet splits, and (Spanner) TrueTime-backed strong consistency; pairs with `09 / 10`.
- Kleppmann, “A Critique of the CAP Theorem” and his CRDT papers (Automerger) for conflict-free merge depth.
- Werner Vogels, “Eventually Consistent” (CACM 2009) — the philosophy behind leaderless/quorum design.

Next: `09-cap-pacelc-and-consistency-models.md` formalizes the consistency guarantees this module kept invoking; `10-distributed-consensus-and-time.md` covers Raft/Paxos and the logical clocks behind safe conflict resolution; `11-distributed-transactions-and-sagas.md` handles the cross-partition writes §3 warned about.

09 — CAP, PACELC & Consistency Models

Source: [readme/research/system-design/09-cap-pacelc-consistency-models.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains a fundamental problem every database faces: when computers storing the same data are spread across different locations, what happens when the connection between them breaks? It explains the rules that govern what you can and cannot guarantee — freshness of data, speed of reads and writes, and whether the system stays open for business during a failure. Understanding this helps you make smart choices about which database technology fits which job.

The main parts explained simply:

- **CAP Theorem** — A proven rule that says: when the network between database servers breaks, you must choose between staying open (accepting reads and writes) OR keeping data perfectly up to date. You cannot do both at the same time. Like two ticket clerks in different cities whose phone line drops — either stop selling (safe but customers leave) or keep selling and risk double-booking seat 42.
- **PACELC — a better, more practical framework** — An improvement on CAP that adds a second question: even when the network is healthy, do you prefer speed (low latency) or perfectly fresh data (consistency)? Most databases pay the freshness cost on every single request, not just during failures, so this everyday trade-off matters more than the rare network-break case.
- **The consistency spectrum (from strongest to weakest)** — “Consistent” is not one thing. From strongest to weakest: Linearizable (every read sees the absolute latest write, but slowest), Causal (you always see cause before effect), Session guarantees like Read-your-writes (you always see your own edits), and Eventual (data will match eventually, but maybe not right now). Most user complaints — “I saved it and it disappeared!” — are solved by simple session-level guarantees, not the expensive strongest level.
- **Quorum math ($W + R > N$)** — When data is copied to N servers, a write must reach W servers and a read must check R servers. If $W + R$ is greater than N , you are guaranteed to read fresh data. If not, reads can be stale. This is the formula databases use to tune the speed-versus-freshness dial.
- **Google Spanner (the “always consistent” pole)** — Google built a global database for its ad billing system where double-charging an advertiser was never acceptable. It pays extra latency on every single write (a small wait of a few milliseconds) to guarantee every read everywhere sees the absolute latest data. It uses GPS and atomic clocks to make this work.
- **Amazon DynamoDB / Dynamo (the “always available” pole)** — Amazon built its shopping cart to never reject a “Add to Cart” click, even if servers lose connection. It accepts that two copies of the cart might get out of sync briefly, and merges them afterward. Speed and availability were worth more than perfect freshness for this use case.
- **Tunable consistency (Cosmos DB example)** — Modern databases let you pick the freshness level per request, not once per database. You can use the strongest (and slowest) guarantee only for critical things like account balances, and use the fastest (and cheapest) guarantee for things like view counts. Pay for strength only where it matters.
- **ACID isolation vs CAP consistency — they are different things** — “Consistent” in database transactions (ACID) means your business rules are enforced. “Consistent” in the CAP sense means

every server shows the same up-to-date data. These are completely separate guarantees. A system can be strong on one and weak on the other. Mixing them up causes costly mistakes.

What to do with this:

Name your most important business rule first — “a customer must never be double-charged” demands the strongest (slowest) consistency; “a like count might be slightly off” is fine with the fastest (cheapest). Once you know your rule, the right database setting follows automatically. Never choose “strong consistency everywhere by default” — it is the most expensive option and most data does not need it.

What you'll learn: How to reason precisely about what a distributed datastore guarantees, why “strong vs eventual” is a beginner’s framing, and how to read a database’s marketing page and know exactly what it will do to your data during a network partition or a slow replica. You’ll leave able to place any system (Spanner, Dynamo, Cosmos DB, Postgres-with-replicas) on a precise consistency spectrum and defend the choice in an interview or a postmortem.

Prerequisites: Read [08-replication-and-partitioning.md](#) first (leader/follower, quorums, replication lag are assumed). Helpful: [07-distributed-time-and-ordering.md](#) (logical clocks, happens-before) and [10-distributed-transactions-and-consensus.md](#) (Raft/Paxos, 2PC) as a follow-on.

1. Intuition first: the cost of agreeing across a gap

Imagine two ticket-counter clerks for the same concert, one in London, one in Tokyo, connected by a phone line. There are 100 seats. To *never* oversell, every sale must be confirmed by both clerks before it’s final — that’s **consistency**, and it costs a round-trip every time. Now the phone line drops (a **partition**). Each clerk faces a choice:

- **Keep selling** (stay **available**) — but now they might both sell seat 42. They’ve chosen availability over consistency.
- **Stop selling** until the line is back (stay **consistent**) — customers walk away angry. They’ve chosen consistency over availability.

There is no third option that both stays open *and* never oversells while the line is down. That impossibility is the entire content of the CAP theorem. Everything else is precision about what “consistent,” “available,” and “partition” actually mean — and that precision is where every real engineering mistake hides.

2. CAP stated precisely (and what it does NOT say)

Brewer conjectured it (2000); Gilbert & Lynch proved it (2002). The precise statement:

In an asynchronous network, a system cannot simultaneously guarantee all three of:

- **C (Consistency)** = **linearizability** — every read sees the most recent completed write, as if there were a single copy of the data updated atomically.
- **A (Availability)** = *every* request to a *non-failing* node returns a non-error response, eventually.
- **P (Partition tolerance)** = the system keeps working when the network drops or delays messages between nodes arbitrarily.

The proof is a one-paragraph argument: partition the network into `{N1}` and `{N2}`. Client writes `x=1` to `N1`. Another client reads `x` from `N2`. `N2` cannot have heard the write (the partition blocks it). It must either return stale `x=0` (not linearizable → **not C**) or block/error (**not A**). QED.

The misinterpretations that get people fired

Myth	Reality
"Pick 2 of 3."	Wrong framing. P is not optional — networks <i>will</i> partition. You don't choose P; the network imposes it. The real choice is C vs A, but only during a partition . When there's no partition, you get both.
"CAP's C = ACID's C."	No. CAP-C is linearizability (a recency/ordering property on single objects). ACID-C is "transactions preserve invariants" (a <i>you-write-the-constraints</i> property). They are unrelated. See §6.
"My system is CP, so it's always strongly consistent."	CP only promises linearizability <i>when reachable</i> . A CP system can still serve stale reads from a follower if you read the wrong node, or if it's misconfigured.
"CAP applies all the time."	CAP only describes behavior during a partition . It says nothing about the 99.9% of the time the network is healthy — which is where latency, not partitions, dominates your design. That gap is why PACELC exists.
"Availability means low latency / high uptime."	CAP-A is binary and theoretical: <i>every</i> non-failing node <i>always</i> answers. A system that answers in 30 seconds is "available" to CAP and useless to you.

The headline: **CAP is a statement about one narrow failure mode (the partition) and one narrow consistency level (linearizability)**. It is necessary but far too coarse to design with.

3. PACELC — the framework you should actually use

Daniel Abadi's 2012 refinement fixes CAP's biggest blind spot: latency. Read it as a sentence:

```

if (Partition)
    P → [ A vs C |
         | available |
         | -or- |
         | consistent | ]
else (normal operation)
    E → [ L vs C |
         | low-latency |
         | -or- |
         | consistent | ]
  
```

If Partition, choose A or C; Else, choose L (latency) or C (consistency). The ELC half is the insight: even with a perfectly healthy network, **linearizability costs latency** — to be linearizable you must coordinate (a quorum round-trip, a leader hop, a clock-wait), and coordination has a price every single request, partition or not. Most of your latency budget is spent in the ELC world, not the PAC world.

Classifying real systems with PACELC

System	PAC (partition)	ELC (normal)	One-line rationale
DynamoDB / Cassandra (default)	PA	EL	Stays up under partition; tunable but defaults favor latency. AP/EL.
Google Spanner	PC	EC	Refuses to violate external consistency even under partition; pays commit-wait latency normally. PC/EC.
MongoDB (default majority)	PC	EC	Leader-based, majority writes; blocks rather than split-brain.
Cosmos DB	tunable	tunable	Five consistency levels you pick per request (\$7).
Postgres single-node	n/a (one node)	EC	No partition between replicas of the primary; CAP doesn't apply to a single node.
PNUTS (Yahoo)	PC	EL	Abadi's classic example of the "weird" PC/EL corner — consistent under partition but latency-optimized normally.

PACELC gives you **four meaningful buckets** (PA/EL, PA/EC, PC/EL, PC/EC) instead of CAP's two. PA/EL (Dynamo) and PC/EC (Spanner) are the two ends most people know; the cross corners (PC/EL, PA/EC) are where interesting designs live.

4. The consistency spectrum: “strong vs eventual” is too coarse

“Eventual consistency” only promises that *if writes stop, replicas eventually converge*. It says nothing about what you see *in the meantime* — and the meantime is your whole production lifetime. Between linearizable and eventual lies a lattice of useful guarantees. From strongest to weakest:

STRONGEST

Linearizability — single-object, real-time order ("most recent write")

Sequential — one global order, but not tied to real time

Causal — if A happened-before B, everyone sees A before B

Session guarantees (client-centric)

Read-your-writes

Monotonic reads

Monotonic writes

Writes-follow-reads

WEAKEST

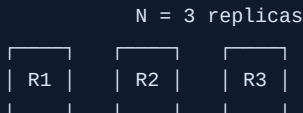
Eventual — converges someday; no ordering promise meanwhile

Model	Plain guarantee	What it forbids	Cost	Real example
Linearizability	Every read returns the latest completed write; operations appear instantaneous in real-time order.	Stale reads, observing operations out of real-time order.	Coordination per op (quorum/leader/clock-wait). Highest latency; unavailable under partition (CP).	etcd, ZooKeeper, Spanner, single-key Dynamo with <code>ConsistentRead</code>
Sequential (Lamport '79)	All clients agree on <i>one</i> order of all ops, consistent with each client's program order — but that order needn't match wall-clock time.	Different clients seeing different orders.	Cheaper than linearizable (no real-time tie).	Rarely sold directly; theoretical baseline
Causal	Causally-related ops ($A \rightarrow B$ via happens-before) are seen in that order everywhere; concurrent ops may be seen in any order.	Seeing an effect before its cause (e.g., a reply before the message).	No coordination on writes; track dependencies (vector clocks). Highest consistency still available under partition (Mahajan et al.).	COPS, MongoDB causal sessions, Cosmos "Consistent Prefix"+session
Read-your-writes	You always see your own prior writes.	"I saved my profile, refreshed, it's gone."	Sticky routing or version token.	Cosmos "Session"
Monotonic reads	Time never goes backward for a reader: once you've seen v5, you won't later see v3.	Refresh showing an <i>older</i> value.	Pin reads to one replica or carry a version.	Session-level guarantee in most stores
Monotonic writes	Your writes are applied in the order you issued them.	Reorder of your own updates.	Per-client write serialization.	—
Writes-follow-reads	A write you make after reading v5 lands <i>after</i> v5 everywhere.	Replying to a comment that "predates" the comment.	Causal metadata.	—
Eventual	Replicas converge if writes stop. Nothing about now.	Nothing meaningful in the interim.	Cheapest; always available.	Dynamo default, DNS, Cassandra ONE

The four session/client-centric guarantees (Terry et al., Bayou, 1994) are the **practical sweet spot**: they're cheap (often implemented with sticky sessions or a version cookie) and they kill the bugs users actually notice ("where did my edit go?"). Causal consistency is the theoretical sweet spot: it's the *strongest* model that an **always-available** (AP) system can provide — you cannot do better than causal without sacrificing availability under partition.

5. Quorum consistency: the math

Replicate each key to **N** nodes. A write must be acknowledged by **W** nodes; a read must query **R** nodes and take the freshest (highest version/timestamp).



Write W=2: ✓ R1, ✓ R2 (R3 lags)

Read R=2: query any 2 → at least one overlaps the write set

The strong-consistency condition is $W + R > N$. This forces the read set and write set to **overlap by at least one node**, so any read is guaranteed to touch a replica that saw the latest acknowledged write.

- $N=3, W=2, R=2 \rightarrow 4 > 3$ ✓. Tolerate 1 node down for both reads and writes. The classic balanced choice.
- $N=3, W=3, R=1 \rightarrow$ fast reads, but a single down node blocks **all** writes.
- $N=3, W=1, R=1 \rightarrow 2 > 3$ ✗. Fast and highly available, but **eventually consistent** — reads may miss the latest write. This is Dynamo/Cassandra's default-ish “fast” mode.
- $W = N/2 + 1$ for both reads and writes = **majority quorum**, the basis for tolerating f failures with $2f+1$ nodes (see Raft/Paxos in [10-distributed-transactions-and-consensus.md](#)).

Why quorum overlap is *not* full linearizability. Even with $W+R>N$, Dynamo-style sloppy quorums have failure modes:

- **Read overlap ≠ atomic visibility.** Two concurrent reads during an in-flight write can return *different* values (one hits the new replica, one doesn't) — non-monotonic. Overlap guarantees you *can* see the latest, not that everyone sees the *same* thing simultaneously.
- **Sloppy quorum + hinted handoff.** Under partition, Dynamo writes to *any* N reachable nodes (not the “home” nodes), so W and R sets may not overlap at all — availability over consistency, by design.
- **Last-write-wins clobbering.** With wall-clock LWW conflict resolution, a write with a skewed-backward clock silently erases a newer write. Use version vectors instead (see [07-distributed-time-and-ordering.md](#)).

Back-of-envelope: with $N=3$ across 3 AZs in one region, a $W=2 / R=2$ majority write costs **~1–2 ms** intra-region. Stretch that quorum across two continents (e.g., us-east ↔ eu-west, ~80 ms RTT) and *every* strongly-consistent write now costs **~80 ms minimum** — the **ELC** tax made concrete. This is why cross-region strong consistency is so painful and why systems either pin a leader per region or relax to causal/eventual.

6. Isolation vs consistency: ACID-I is NOT CAP-C

This conflation derails interviews constantly. They're orthogonal axes:

Axis	Question it answers	Spectrum	Failure it prevents
Consistency (CAP)	"Across <i>replicas</i> of one object, how fresh/ordered is what I read?"	linearizable → causal → eventual	Reading stale data from a lagging replica
Isolation (ACID)	"Across <i>concurrent transactions</i> on multiple objects, how much can they interfere?"	serializable → snapshot → read-committed → read-uncommitted	Lost updates, dirty reads, write skew

A system can be **serializable but not linearizable** (e.g., serializable snapshot isolation on a single node with async replicas — perfect isolation, stale reads on a follower). It can be **linearizable but not serializable** (a single register with compare-and-swap — perfectly fresh, but no multi-object transaction semantics).

The gold standard that combines both is **strict serializability** = serializable + linearizable: transactions are equivalent to *some* serial order, *and* that order respects real-time. **Spanner** delivers exactly this and calls it **external consistency** (§8). When someone says “strongly consistent,” ask: *do you mean linearizable (single-object recency) or strict-serializable (multi-object transactions in real-time order)?* The answer changes the architecture.

7. Cosmos DB: tunable consistency as a product

Azure Cosmos DB is the clearest teaching artifact because it exposes **five named levels** as a per-request knob, and Microsoft publishes (TLA±verified) definitions:

```
Strong → Bounded Staleness → Session → Consistent Prefix → Eventual
(linearizable) (lag-bounded) (per-client) (no gaps) (converge)
highest latency / lowest availability —————> lowest latency / highest availability
```

Level	Guarantee	Use when
Strong	Linearizable. Reads see the latest committed write. Confined to a single region or synchronous-replica config.	Financial balances, inventory counters where overselling is unacceptable.
Bounded staleness	Reads lag the latest write by at most K versions or T seconds — quantified staleness.	“Near-real-time is fine, but never more than 5 s / 100 ops behind.” Leaderboards, dashboards.
Session (default)	Read-your-writes + monotonic reads + consistent prefix, scoped to a session token .	The 90% case: a logged-in user must see their own edits. Cheap and globally available.
Consistent prefix	You never see writes out of order (no gaps), but may see an older snapshot.	Activity feeds where order matters more than recency.
Eventual	Converges; no order guarantee. Lowest latency, highest availability.	Like counts, view counters, where staleness is harmless.

The lesson: consistency is **per-operation, not per-database**. Mature systems let you pay for strength only where it matters and bank the latency/availability everywhere else.

8. Spanner & TrueTime: buying linearizability with a clock

Spanner is the canonical PC/EC, strict-serializable, globally-distributed SQL database — the one that “shouldn’t exist” by naive CAP reasoning. Its trick is **TrueTime**: instead of a single timestamp, the clock API returns an **interval** `[earliest, latest]` with a bounded uncertainty ϵ (kept to **~1–7 ms** via GPS + atomic clocks in every datacenter).

To commit a transaction at timestamp t , Spanner does **commit-wait**: it picks $t = \text{TT.now().latest}$, then *waits out the uncertainty* — it blocks until $\text{TT.now().earliest} > t$, i.e. for roughly $2\epsilon \approx 1\text{--}14\text{ ms}$ — guaranteeing that when the commit is acknowledged, t is unambiguously in the past everywhere on Earth. That’s how it gets **external consistency** (global real-time order) without a global clock.

```
T1.commit picks t=100, ε=5ms
|— commit-wait until earliest > 100 (≈2ε) —|
▼                                     ▼ ack
t=95.....t=100.....t=105 (now certainly past)
Any later T2 anywhere gets t' > 100. Real-time order preserved.
```

The trade-offs Spanner accepts: every read-write transaction pays the $\sim 2\epsilon$ commit-wait (the **ELC** tax, in latency form), and it needs **specialized clock hardware**. CockroachDB and YugabyteDB reproduce the design *without* GPS/atomic clocks by using a looser **~250–500 ms** max-clock-offset bound (HLCs + NTP), then handling uncertainty with read restarts instead of commit-wait — cheaper hardware, occasionally retried reads. Under partition, Spanner stays **CP**: a minority partition simply can’t commit (Paxos needs a majority), so it sacrifices availability, never consistency.

9. Dynamo: the other pole

Amazon's Dynamo (2007 paper — the design DNA of DynamoDB *and* Cassandra/Riak/Voldemort) is the PA/EL archetype: **always writeable**, even during partitions, because shopping-cart adds must never fail. It achieves this with consistent hashing, sloppy quorums + hinted handoff, anti-entropy via Merkle trees, and **conflict resolution pushed to the application** via version vectors (the famous “two divergent carts get merged” example). Dynamo's lesson is the inverse of Spanner's: if your business value is *taking the write*, you accept divergence and design a merge function. (DynamoDB the AWS product later added opt-in strongly-consistent reads and transactions, moving it toward *tunable*.)

10. Common pitfalls / war stories

- **“We're CP so we never lose data.”** CP sacrifices *availability*, not durability — and only guarantees linearizability *if you read through the leader/quorum*. Teams read from a follower for performance and silently reintroduce stale reads. Reading from a replica is an eventual-consistency decision *no matter what the database is labeled*.
 - **The read-your-writes refresh bug.** User updates their profile, the write goes to the leader, the immediate read load-balances to a lagging follower, the UI shows the *old* value, the user re-submits → duplicate/confusion. Fix: route the user to the leader (or to the same replica) for a few seconds post-write, or pass a version token. This is *exactly* what session consistency exists to prevent.
 - **Last-write-wins data loss.** LWW on physical clocks means a node with a clock 2 s in the future wins every conflict, and a backward-skewed node's newer writes vanish — silently. Cassandra users have lost data to NTP glitches this way. Prefer version vectors or app-level merges.
 - **Cross-region synchronous quorum = latency outage.** Stretching a majority quorum across continents makes every write pay the inter-region RTT; under load this looks like an outage even though nothing is “down.” Keep the consistency boundary inside one region; replicate across regions asynchronously.
 - **Confusing ACID isolation with replica consistency** (§6): “We use serializable transactions, so reads are always fresh” — false if reads hit async replicas. Two different guarantees.
 - **Believing the marketing word “consistent.”** Always translate to a precise model: *linearizable?* *causal?* *bounded-stale by how much?* *session-scoped?* If the vendor can't say, assume eventual.
 - **Choosing strong consistency by default “to be safe.”** It's the most expensive default — latency on every op and unavailability under partition. Most reads (likes, feeds, counts) are happy with session or eventual; reserve linearizability for the few invariants that truly need it (balances, inventory, locks, leader election).
-

🧩 Case Study: Google Spanner vs Amazon DynamoDB — the same trade-off, opposite answers

By 2011 Google’s ad business (AdWords) had outgrown a sharded MySQL deployment that held **tens of TB** of revenue-critical data spread over thousands of shards. Resharding it took the team **over two years of manual work**, and the application had to tolerate eventual-consistency anomalies that engineers hated — because money was at stake, an advertiser’s budget read had to reflect the latest debit *exactly*, across continents. Google wanted a system that gave SQL, ACID transactions, and **external consistency** (strict serializability) at global scale. Naive CAP reasoning says this is impossible: a globally distributed store *must* drop C or A under partition. Spanner is the answer to “what if we just pay the C price everywhere, on purpose.”

Amazon had faced the mirror-image problem a few years earlier. During the 2006 holiday peak, the shopping-cart service — backed by a relational store — saw availability dips that translated directly into lost sales at a scale of **millions of requests per second** across hundreds of services. Amazon’s verdict: *an “add to cart” must never fail, even if the network is partitioned*, because a rejected write is a rejected dollar. Dynamo (and its product descendant DynamoDB) is the answer to “what if we just pay the A price everywhere, on purpose.”

These two systems sit at the exact poles this whole module is about. Map them onto the **PACELC** framing from §3:

	PARTITION (PAC)	NORMAL (ELC)	
SPANNER →	C (refuse to split-brain; minority can't commit)	C (commit-wait ~2ε per txn)	= PC/EC
DYNAMODB →	A (sloppy quorum + hinted handoff)	L (W=1/R=1 by default; no coordination)	= PA/EL

Spanner applies the module’s concepts as follows. Its C is the strict-serializable gold standard from §6 — linearizable *and* serializable — which Google brands *external consistency*. It buys that with the **linearizability-costs-latency** law of the **ELC** half (§3): every read-write transaction pays **commit-wait** ≈ **2ε** (§8), where TrueTime keeps **ε** to ~1–7 ms via GPS + atomic clocks. That commit-wait is the **ELC** tax made literal — coordination on the healthy-network common case, not just during partitions. Under partition it lands in the **PC** corner: each Paxos group needs a majority to commit, so a minority partition simply blocks (the “stop selling” clerk from §1). Spanner never serves the stale-read that would violate linearizability; it gives up availability instead.

DynamoDB applies the *inverse* set of concepts. Its default reads are eventually consistent — the bottom of the §4 spectrum — implemented with the **W=1, R=1** quorum from §5, where **W + R = 2 ≤ N**, deliberately *failing* the **W+R>N** overlap test so reads never wait on a coordinating replica. That’s the **EL** corner: lowest latency, single-digit-millisecond reads regardless of replica count. Under partition it uses

sloppy quorums + hinted handoff (§5, §9) to stay writeable on *any* reachable nodes — the **PA** corner, the “keep selling” clerk. The cost it accepts is divergence, resolved by version vectors and app-level merges (the two-divergent-carts example).

The key trade-off each accepted. Spanner gave up *latency and availability* to get *real-time global ordering*: it cannot commit a write in a minority partition, and it pays the commit-wait tax on every transaction plus needs specialized clock hardware. DynamoDB gave up *recency and single-system-image semantics* to get *unconditional write availability and flat low latency*: a default read can be stale, and concurrent writes can diverge, pushing conflict resolution onto you.

Real numbers. Spanner runs Google’s production workloads (AdWords, Play) on databases spanning **millions of tables and trillions of rows**, with TrueTime ϵ typically **under 7 ms** and commit-wait on the order of **a few ms** — the price of external consistency. DynamoDB powers Amazon retail and serves customers like Prime Day at **tens of millions of requests per second** with **single-digit-millisecond** P99 read latency for eventually-consistent reads; opting into a strongly-consistent read (added later, moving DynamoDB toward the *tunable* column of §3) roughly **doubles** the read cost and gives up some availability — the consistency tax, billed per request.

The deepest lesson is that **neither is “better.”** They answered the *same* PACELC question with opposite values because their business invariants differed: Google’s invariant was “an advertiser is never double-charged and balances are globally ordered” (an invariant that *requires* linearizability); Amazon’s was “a cart add never fails” (an invariant that *requires* availability). Pick the pole your invariant demands — and, as Cosmos DB (§7) shows, mature systems now let you pick per operation rather than per database.

Lessons

- **The C-vs-A choice is dictated by your invariant, not your taste.** “Never double-charge” forces linearizability (Spanner); “never drop a write” forces availability (Dynamo). Name the invariant first, then the pole follows.
- **The **ELC** tax is the one you pay daily.** Partitions are rare; Spanner’s commit-wait and DynamoDB’s consistent-read surcharge are paid on the healthy network, every request. Most of your latency budget lives in **ELC**, not **PAC**.
- **Strong consistency at global scale is *achievable* but never free** — Spanner needed atomic clocks and accepts commit-wait to make naive-CAP-“impossible” SQL real. If you can’t pay that, relax to causal/session (the AP ceiling, §4).
- **The market converged on tunable.** Both poles drifted toward the middle — DynamoDB added strong reads and transactions; this is why per-operation consistency (§7), not a per-database label, is the modern design stance.

11. Test yourself

1. **A team says their database is “CA — consistent and available, we don’t need partition tolerance.”** Why is this almost always a confused statement? (*Hint: P isn’t a choice on a real network; “CA” only describes a single node or a system that gives up the moment the network hiccups.*)

2. **Give a concrete user-facing bug that read-your-writes prevents but monotonic-reads does not, and vice versa.** (Hint: RYW = seeing your own edit; monotonic reads = never seeing time go backward across two reads, even of others' data.)
 3. **With $N=5$, list two (W, R) pairs that give strong consistency and one that doesn't, and state each pair's availability trade-off.** (Hint: need $W+R>5$; e.g. (3,3) balanced, (5,1) fast reads/fragile writes, (1,1) eventual.)
 4. **Why does Spanner's commit-wait make latency proportional to clock uncertainty ϵ , and what would happen if ϵ were 200 ms?** (Hint: it blocks $\sim 2\epsilon$ per commit; 200 ms $\epsilon \Rightarrow \sim 400$ ms per write — why CockroachDB uses read-restarts instead.)
 5. **Is “serializable” stronger than “linearizable”?** Explain the trick in the question. (Hint: they're different axes — multi-object isolation vs single-object recency; strict serializability is both.)
 6. **Which is the strongest consistency model an always-available (AP) system can offer, and why can't it go stronger?** (Hint: causal — anything stronger needs coordination that a partition can block, forcing you to drop availability.)
 7. **Your shopping cart service occasionally shows two divergent carts after a network blip. Is this a bug or a design choice, and what's the standard remedy?** (Hint: Dynamo's deliberate AP behavior; remedy is version vectors + an app-level merge function, not LWW.)
 8. **Place MongoDB (default), DynamoDB (default), and etcd in PACELC and justify each.** (Hint: MongoDB PC/EC majority leader; DynamoDB PA/EL; etcd PC/EC Raft.)
-

12. Further reading

- **DDIA (Kleppmann), Chapter 5** (Replication — leaders, quorums, read-your-writes) and **Chapter 9** (Consistency & Consensus — linearizability, the CAP debate, ordering). The single best treatment; read these two chapters in full.
- Gilbert & Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services” (2002) — the CAP proof.
- Daniel Abadi, “Consistency Tradeoffs in Modern Distributed Database System Design” (IEEE Computer, 2012) — the PACELC paper. Also his blog “Problems with CAP.”
- Brewer, “CAP Twelve Years Later: How the ‘Rules’ Have Changed” (2012) — the author’s own correction of the “pick 2” myth.
- Corbett et al., “Spanner: Google’s Globally-Distributed Database” (OSDI 2012) — TrueTime & external consistency.
- DeCandia et al., “Dynamo: Amazon’s Highly Available Key-value Store” (SOSP 2007).
- Terry et al., “Session Guarantees for Weakly Consistent Replicated Data” (1994) — the four client-centric guarantees.
- Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs” (1979) — sequential consistency; and “Time, Clocks, and the Ordering of Events” (1978).
- Azure Cosmos DB official docs, “Consistency levels” — the five-level model with TLA+ specs.

- Peter Bailis et al., *“Highly Available Transactions”* (HAT, VLDB 2014) — what isolation levels survive AP. [Jepsen.io](#) analyses — empirical consistency torture-tests of real databases.

Next: [10-distributed-transactions-and-consensus.md](#) builds on quorums and linearizability to cover Paxos, Raft, and 2PC — the machinery that actually implements the CP/EC corner.

10 — Consensus & Distributed Coordination

Source: [readme/research/system-design/10-consensus-and-coordination.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains how a group of computers can agree on one correct answer — even when some of them crash, slow down, or lose messages. It matters because almost every feature that “must never go wrong” (picking one leader, locking a shared record, keeping replicas in sync) depends on this agreement machinery working reliably.

The main parts explained simply:

- **Why agreement is hard** — Computers can't tell if a peer is dead or just very slow. This single fact makes perfect, always-on agreement mathematically impossible, and everything else in this document is a practical workaround for that limitation.
- **FLP impossibility** — A 1985 proof that says: you cannot guarantee both “we will always decide” and “we will always be safe” when even one machine might crash. Real systems keep the “always safe” side and accept that progress may pause briefly during failures.
- **Paxos** — The original agreement algorithm. It works by having machines vote in rounds until a majority agrees. Proven correct but famously hard to turn into real code because many details are left undefined.
- **Raft** — A cleaner, fully specified version of the same idea. One machine is “leader” and tells the others what to record; if it dies, the others elect a new leader within a second or two. This is what etcd, Consul, and CockroachDB actually run.
- **ZooKeeper / etcd / Consul** — Ready-made services that run Raft or a similar protocol for you. Your app asks them to store small, critical facts (who is the leader? is this lock held?) and they guarantee those facts are consistent across machines. Think of them as a tiny, ultra-reliable shared notebook.
- **Leader election** — A recipe built on top of the above: whoever grabs a special key first is the leader; if they go quiet, the key expires and someone else takes over automatically.
- **Distributed locks — and why they fail** — A “lock” across machines is not safe by itself. A machine can pause (garbage collection, network hiccup), the lock expires, a second machine grabs it, and now two machines both think they own it. The only reliable fix is a fencing token — a number that rises with each new grant; the shared resource rejects any write carrying a stale, lower number.
- **Leases** — A lock with a built-in expiry time. The holder must keep renewing it; if it crashes, the lease simply runs out and someone else takes over. Leaders can use a valid lease to answer reads instantly without asking the whole group first — a useful speed-up.
- **Failure detection (Gossip / SWIM / Phi-accrual)** — Methods for spreading “who is alive” information across hundreds or thousands of machines cheaply. Gossip works like rumour — each machine tells a few neighbours, and the news spreads quickly. Phi-accrual gives a sliding suspicion score instead of a hard timeout, so the system adapts to slow networks without false alarms.
- **Clocks for ordering** — Wall clocks on different machines drift and can jump, so you cannot use them to decide which event happened first. Lamport timestamps (a simple counter), vector clocks (one counter per machine), and Hybrid Logical Clocks (a mix of real time + counter) each give stronger guarantees at higher cost. Google's Spanner uses GPS-backed atomic clocks and waits out the remaining tiny uncertainty before confirming a transaction.
- **Common traps** — Using Redis as a correctness lock (it is not one), running an even number of consensus nodes, storing large data in etcd/ZooKeeper, and hand-coding Raft while missing a subtle

safety rule that causes silent data overwrite.

- **Kubernetes case study** — Kubernetes stores all cluster state in a 3-or-5-node etcd cluster. Every write goes through Raft, so the state is always consistent. Controllers use etcd leases for their own leader election, and every object carries a version number that rejects stale writes — the same fencing-token idea, built into the API.

What to do with this: When a system must never do something twice or must have exactly one active worker, reach for a proper consensus-backed service (etcd or ZooKeeper) rather than a plain Redis lock. Always pair it with a fencing token — let the resource reject stale writers — because a lock alone cannot stop a paused process from waking up and causing damage.

What you'll learn: How a group of machines that can crash, pause, and lie about timing nonetheless agree on a single value or an ordered sequence of decisions — the bedrock under leader election, distributed locks, config stores, and replicated databases. We go from the FLP impossibility result through Paxos and Raft, into the real coordination services (ZooKeeper, etcd, Consul), the *dangerous* world of distributed locks, and the clock machinery (Lamport, vector, HLC) that makes ordering possible.

Prerequisites: Read [02-networking-and-rpc.md](#) (timeouts, partial failure), [08-replication-and-partitioning.md](#) (replicated state, quorums), and [09-cap-pacelc-and-consistency-models.md](#) (linearizability, the C/A trade-off) first. Consensus is the *mechanism* that buys you the strong-consistency end of CAP.

1. Why consensus at all? (intuition)

Imagine three bank tellers sharing one ledger, but they can only pass notes, notes can be lost or arrive late, and any teller can faint mid-sentence. They must never both hand out the last \$100. Consensus is the protocol for the notes: a way to reach **one agreed decision** even though messages are unreliable and participants fail.

Formally, a consensus protocol over a set of nodes must satisfy:

- **Agreement** — no two correct nodes decide different values.
- **Validity (integrity)** — the decided value was proposed by some node (no inventing values).
- **Termination (liveness)** — every correct node eventually decides.

Almost everything hard in distributed systems reduces to consensus: electing a leader, committing a transaction across shards, ordering a replicated log, fencing a lock, agreeing on cluster membership. If you can totally-order a log of commands, you can build any replicated state machine (RSM) — feed the same ordered commands to deterministic replicas and they stay identical. That's the **State Machine Replication** theorem and it's why "consensus = a replicated log" in practice.

```

Clients
  |  commands (append "x=5", "y=7", ...)
  v
+----- consensus layer -----+
| leader  ----> replicated, totally-ordered log |
| |      idx: 1    2    3    4 |
| |      [x=5] [y=7] [x=9] [z=1]  <- identical on majority |
+-----+
      |      |      |
      v      v      v
SM rep1 SM rep2 SM rep3      apply in order -> identical state

```

2. FLP: the impossibility you must respect

Fischer–Lynch–Paterson (1985): In an *asynchronous* system (no bound on message delay or relative processor speed) where even **one** process may crash, there is **no deterministic protocol that guarantees consensus**. You cannot have safety + liveness + fault tolerance all three, all the time, in pure asynchrony.

The intuition: you can never distinguish “node is dead” from “node is slow / message is in flight.” A scheduler can always delay the one message that would let the system decide, holding it in an indecisive (“bivalent”) state forever.

This is not a counsel of despair — it tells you *what to give up*. Real systems keep **safety always** and accept that **liveness is conditional**: progress happens only during periods of *partial synchrony* (Dwork–Lynch–Stockmeyer), i.e. when the network behaves “well enough” for long enough. Paxos, Raft, ZAB all live here: they never decide two different values (safe), but during a bad partition or a leader-election storm they may simply stall (no liveness) until things calm down. **When an interviewer asks “how does Raft beat FLP?” the answer is: it doesn’t — it sacrifices guaranteed liveness, using randomized timeouts to make permanent stalling astronomically unlikely.**

Distinction: FLP assumes crash faults. If nodes can lie/corrupt arbitrarily you need **Byzantine** consensus (PBFT, Tendermint) requiring $3f+1$ nodes to tolerate f liars, vs $2f+1$ for crash faults. Most infra (Raft/Paxos) is crash-fault-tolerant only — fine inside one trusted datacenter, not for open blockchains.

3. Paxos — the canonical algorithm (intuition over proof)

Lamport’s Paxos solves consensus for a single value under partial synchrony. Roles: **proposers**, **acceptors** (the voting majority), **learners**. Decisions are made by **any majority (quorum)**; any two majorities of $2f+1$ nodes intersect in ≥ 1 node, which is the trick that prevents two conflicting decisions.

Two phases:

```

Phase 1 (Prepare): proposer picks ballot n, sends Prepare(n)
                  acceptor: if n > highest seen, PROMISE not to accept < n,
                           and reply with any value it already accepted
Phase 2 (Accept):  proposer picks value v (its own, OR the highest-ballot
                  value any acceptor reported back),
                  sends Accept(n, v)
                  acceptor: accept(n,v) unless it promised a higher ballot
Chosen: a value is CHOSEN once a majority has accepted it.

```

The “adopt the highest already-accepted value” rule in Phase 2 is the heart of safety: once a value *could* have been chosen, every future ballot is forced to re-propose it. So Paxos can never un-decide.

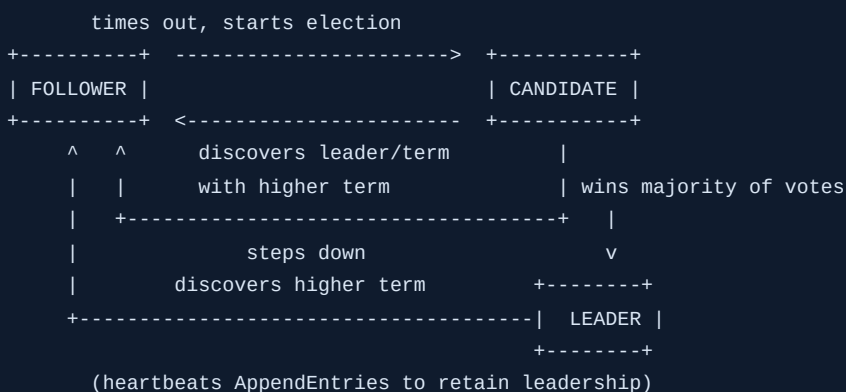
Why people hate basic Paxos: it decides *one* value. Real systems need a *log* of values → **Multi-Paxos:** elect a stable leader once, then skip Phase 1 for every subsequent slot (the leader’s promise covers all future slots), so steady state is one round-trip per command. But Lamport’s papers leave Multi-Paxos under-specified — leader election, membership changes, and log compaction are “left as an exercise,” which is exactly why production engineers found it painful and why Raft exists.

4. Raft — consensus you can actually implement

Raft (Ongaro & Ousterhout, 2014, “In Search of an Understandable Consensus Algorithm”) makes the same guarantees as Multi-Paxos but is decomposed into three understandable pieces: **leader election**, **log replication**, **safety**. It is what etcd, Consul, CockroachDB, TiKV, and MongoDB (variant) actually run.

4.1 Terms and leader election

Time is divided into **terms** (monotonic integers) — a logical clock for “epochs of leadership.” Each term has at most one leader. Nodes are *follower*, *candidate*, or *leader*.



A follower that hears no heartbeat within its **election timeout** (randomized, e.g. 150–300 ms) increments its term and becomes a candidate, votes for itself, and requests votes. A node grants its vote only if the candidate’s log is **at least as up-to-date** as its own (the *election restriction* — this is what keeps a stale node from winning and erasing committed entries). Win a majority → leader. The randomized timeouts make split votes rare; on a tie, everyone backs off a random amount and retries. This randomization is Raft’s pragmatic escape hatch around FLP’s liveness limit.

4.2 Log replication & commit

The leader appends a client command to its log and sends `AppendEntries(term, prevLogIndex, prevLogTerm, entries[], leaderCommit)` to followers. A follower accepts only if its log matches at `prevLogIndex/prevLogTerm` — the **Log Matching Property**: if two logs share an entry at the same index+term, all *preceding* entries are identical. Mismatches cause the leader to walk back and overwrite the follower's divergent tail.

An entry is **committed** once it's replicated on a majority *and* it's from the leader's current term. Committed entries are applied to the state machine in index order. The subtle “current term” rule (Figure 8 in the paper) prevents a leader from considering an older-term entry committed just because it's on a majority — a later leader could still overwrite it. This is a classic source of buggy hand-rolled Raft.

4.3 Safety guarantees Raft enforces

Property	What it guarantees
Election Safety	≤ 1 leader per term
Leader Append-Only	leaders never delete/overwrite their <i>own</i> log
Log Matching	same index+term $\Rightarrow$ identical prefixes
Leader Completeness	a committed entry is present in all future leaders' logs
State Machine Safety	no two nodes apply different commands at the same index

Paxos vs Raft

Dimension	Multi-Paxos	Raft
Mental model	flexible, role-based, proof-first	strong-leader, log-first
Leader	optional / multiple proposers OK	mandatory single leader
Log holes	can commit out of order	log is contiguous, no holes
Spec completeness	leaves a lot unspecified	full algorithm incl. membership + snapshots
Membership change	ad-hoc	joint consensus / single-server change
Implementations	Chubby, Spanner, Megastore	etcd, Consul, CockroachDB, TiKV, RethinkDB
When to use	you need flexible quorums (see Flexible Paxos / EPaxos for WAN)	you want a correct, maintainable, single-DC log

EPaxos / Flexible Paxos are the advanced answer when a single leader is a WAN bottleneck: they exploit the fact that Phase-1 and Phase-2 quorums only need to *intersect*, allowing smaller Phase-2

quorums or leaderless commits for non-conflicting commands.

5. ZAB & ZooKeeper

ZAB (ZooKeeper Atomic Broadcast) predates Raft and powers Apache ZooKeeper. Same shape — a single elected leader broadcasts a totally-ordered stream of state changes to followers, committed by majority. Differences worth knowing:

- ZAB is built around **primary-order atomic broadcast** with explicit *recovery* and *broadcast* phases; on leader change it runs a **synchronization** phase to make followers' logs match the new leader before serving.
- ZooKeeper transaction IDs (**zxid**) are 64-bit: high 32 bits = epoch (leader generation, like Raft's term), low 32 bits = counter. The epoch-in-the-id trick makes stale-leader writes trivially detectable.
- ZooKeeper exposes a hierarchical **znode** filesystem, **watches** (one-shot change notifications), **ephemeral** nodes (auto-deleted when the creating session's heartbeat lapses), and **sequential** nodes (server-assigned monotonic suffixes). These four primitives are the building blocks for almost every coordination recipe.

6. Coordination services: ZooKeeper vs etcd vs Consul

You rarely implement Raft yourself. You run a coordination service and use *its* primitives. All three are CP systems (favor consistency; refuse writes during a quorum-losing partition).

Feature	ZooKeeper (ZAB)	etcd (Raft)	Consul (Raft)
Data model	hierarchical znodes	flat sorted KV (MVCC, revisions)	KV + rich service catalog
Watch	one-shot watches	streaming watch from a revision	blocking queries
Liveness primitive	ephemeral nodes + sessions	leases (TTL, must renew)	TTL/health-check sessions
Linearizable reads	sync() then read	yes (default) / serializable opt-in	default vs stale modes
Service discovery	DIY	DIY-ish	first-class (DNS, health checks)
Built-in DNS/mesh	no	no	yes (+ Connect mTLS mesh)
Used by	Kafka(<3.x), HBase, Hadoop	Kubernetes , CoreOS, M3	HashiCorp stack, multi-DC infra
When to use	mature JVM ecosystems, ZK recipes	k8s-native, simple strong KV + leases	service discovery + multi-DC + mesh

Typical cluster size is **3 or 5 nodes** (odd, to make majorities clean): tolerates 1 or 2 failures respectively. Going to 7 buys little fault tolerance and *slows* writes (bigger quorum to ack). Keep these clusters small and dedicated; they are the brain stem.

7. Leader election as a recipe

You don't need a full RSM for "pick one coordinator." Two production patterns:

1. **Ephemeral + sequential nodes (ZooKeeper):** every contender creates an ephemeral-sequential znode under `/election/`. The one with the lowest sequence number is leader. Each other contender *watches only the node just below it* (not the leader — avoids a thundering herd of N watches firing at once). When the leader's session dies its ephemeral node vanishes, the next-lowest is notified and promotes.
2. **Lease key (etcd):** `PUT /leader value=me` bound to a lease with a TTL; the holder must `KeepAlive` the lease. If it crashes, the lease expires, the key disappears, and a watcher takes over. This is **lease-based leadership** (§9).

Both give you a single coordinator *only while the coordination service's consensus holds* — which is exactly the safety you want.

Lock type	Backing	Safe for correctness?	Use when
<code>SET NX PX</code> (single Redis)	best-effort, no consensus	No (no fencing, drift)	cheap “avoid duplicate work” hint
Redlock (N Redis)	quorum, no fencing	No (Kleppmann)	rarely justified
etcd/ZK lease lock + fencing	Raft/ZAB + monotonic token	Yes	money, file writes, leader-only mutations
DB row lock / <code>SELECT FOR UPDATE</code>	the DB’s own consensus	Yes (same transaction)	when the resource <i>is</i> the DB

9. Lease-based leadership (advanced)

A **lease** is a lock with a built-in expiry: “you are leader/holder until time T unless you renew.” It converts *external* failure detection (others must run a protocol to notice you died) into *time-based* self-expiry (your authority simply lapses).

Why leases matter for **reads**: a leader holding a valid lease can serve **linearizable reads locally without a quorum round-trip** — because the lease guarantees no other node can have become leader yet. This is the *leader lease / leader read lease* optimization in Spanner, CockroachDB, and TiKV, turning a network round-trip per read into a clock check.

The catch is **clock safety**. Leases assume bounded clock drift between grantor and holder. If a holder’s clock runs slow it might think its lease is still valid after the grantor expired it → two leaders. Mitigations:

- **Conservative expiry**: the holder treats the lease as expired *earlier* than the grantor does (subtract max-drift ϵ), creating a guard band where nobody acts.
- **Bounded-uncertainty clocks**: Spanner’s **TrueTime** exposes `now()` as an interval `[earliest, latest]` and *waits out* the uncertainty (commit-wait) so intervals can’t overlap — hardware (GPS/atomic clocks) keeps ϵ ~a few ms.
- Tie the lease to the consensus **term/epoch** so a stale leader’s writes are rejected by the log anyway (defense in depth).

Leader leases are the bridge between §8’s “locks are dangerous” and “we still want fast strong reads”: same danger (clock-dependent), tamed by intervals + epochs + conservative expiry.

10. Membership & failure detection

Consensus needs to know *who is in the cluster* and *who is alive*. For small clusters (3–5) the consensus group tracks itself. For large fleets (hundreds–thousands) you use a separate, scalable layer.

Gossip / SWIM

In **gossip** protocols each node periodically picks a few random peers and exchanges state; information spreads epidemically — full propagation in $O(\log N)$ rounds, traffic stays roughly constant per node regardless of cluster size. **SWIM** (Scalable Weakly-consistent Infection-style Membership) refines this: a node pings a random target; on no reply it asks k other nodes to *indirectly* ping it (filtering transient network blips and false positives) before marking it suspect, then dead, gossiping the verdict. Used by **Serf/Consul**, **Cassandra**, **DynamoDB**, **Akka Cluster**, **Redis Cluster** (variant).

Phi-accrual failure detection

Binary “alive/dead” timeouts are brittle: too short → false positives, too long → slow detection. **Phi (ϕ) accrual** (Hayashibara et al.) outputs a *continuous suspicion level* ϕ derived from the statistical distribution of recent inter-arrival times of heartbeats. The app picks a threshold (e.g. act at $\phi=8 \approx 1\text{-in-}10^8$ chance the node is actually alive). It self-tunes to the network’s real jitter. Used by **Cassandra** and **Akka**.

```
heartbeat gaps -> running mean/variance -> phi rises as silence grows
phi: 0 ..... 2 ..... 5 ..... 8 (SUSPECT) ..... 12 (DEAD)
```

Concern	Gossip/SWIM	Phi-accrual	Raft heartbeats
Scale	thousands of nodes	per-link detector	small group only
Output	membership view (eventually consistent)	tunable suspicion score	binary leader-alive
Use	large dynamic fleets, service mesh	adaptive detection layer	the consensus group itself

11. Clocks: ordering without a global stopwatch

Distributed ordering is impossible with wall clocks alone — they drift and skew. The clock toolbox, in order of power:

Clock	Tells you	Cost	Limitation
Physical (NTP)	approximate wall time	cheap	drift/skew 10s–100s ms; can jump backward
Lamport (scalar)	<i>happens-before</i> (one-way)	1 int	$a < b$ doesn't prove causality (can't detect concurrency)
Vector clock	full causality + concurrency detection	$O(N)$ per event	size grows with #nodes
Hybrid Logical Clock	causality and close-to-wall-time	2 ints	needs loosely-synced physical clocks
TrueTime (Spanner)	bounded real-time interval	GPS/atomic HW	infra cost; commit-wait latency

- **Lamport timestamps:** $L = \max(L, \text{received}) + 1$ on every event/receive. Guarantees: if $a \rightarrow b$ (causally), then $L(a) < L(b)$. The converse is *not* true — equal-or-ordered timestamps don't prove causality. Good enough to build a *total* order (break ties by node id) for things like a single leader's log.
- **Vector clocks:** each node keeps a vector $V[i]$; compare element-wise. $V(a) < V(b) \Rightarrow$ causal; **incomparable** $\Rightarrow$ **concurrent** — which is exactly what **DynamoDB/Riak** need to *detect conflicting writes* and surface siblings for resolution.
- **Hybrid Logical Clocks (HLC):** combine a physical component (so timestamps track wall time and are human-meaningful / comparable across systems) with a logical counter (so causality is never violated even when physical clocks jitter). HLC stays within a bounded distance of physical time but never goes backward. Used by **CockroachDB**, **YugabyteDB**, **MongoDB** (cluster time). HLC is the pragmatic default when you want causality *and* timestamps that look like time but you don't have Google's atomic-clock budget.
- **TrueTime:** Spanner's clocks expose uncertainty ϵ ; to make an externally-consistent (linearizable) commit, Spanner **waits out** ϵ (commit-wait, typically a few ms) so no later transaction can get an earlier timestamp. This is how Spanner offers global linearizability — by paying latency to make clock uncertainty safe rather than pretending it's zero.

See [09-cap-pacelc-and-consistency-models.md](#) for how these clocks underpin linearizability vs causal consistency.

12. Common pitfalls / war stories

- **Treating a distributed lock like a mutex.** No fencing token $\rightarrow$ a GC pause hands the resource to two writers $\rightarrow$ silent corruption. *Always fence at the resource.*
- **Using best-effort Redis for correctness locks.** A single **SET NX** is a hint, not a guarantee; if money or file integrity rides on it, you've shipped a data-loss bug that only fires under load (when pauses happen).
- **Even-sized consensus clusters.** A 4-node cluster tolerates the *same* 1 failure as 3 but needs a larger majority and risks more split-vote pain. Always run odd numbers (3/5).
- **Putting too much in the coordination service.** ZooKeeper/etcd are not databases. Storing large values or high write-rate data there saturates the consensus log and freezes the cluster everything depends on. Keep values small (KB) and writes modest.
- **Hand-rolling Raft and missing Figure 8.** The “only commit current-term entries via the majority rule” subtlety is omitted by most blog reimplementations $\rightarrow$ committed data gets silently overwritten. Use a vetted library (etcd/raft, dragonboat, Atomix); if you must roll your own, run the **Jepsen** and TLA+/-model-checked test suites.
- **Trusting NTP for ordering or leases without a guard band.** A backward clock jump (leap second, VM resume) expires a lease in someone's frame but not another's $\rightarrow$ two leaders. Subtract max-drift; prefer HLC/epoch fencing.

- **Thundering-herd watches in leader election.** N contenders all watching the leader node means N notifications + N re-elections churn on every failure. Watch only your immediate predecessor.
- **Confusing failure detection with consensus.** Gossip/SWIM gives you an *eventually-consistent* membership guess; acting on it as if it were a consensus decision (e.g. removing a node from quorum) without going through the consensus group can split-brain you.

✖ Case Study: etcd + Raft as the brain of Kubernetes

The problem. Kubernetes is a cluster orchestrator: it must hold the *desired state* of everything — every Pod, Deployment, Service, Secret, Node, ConfigMap — and let dozens of controllers (the scheduler, kube-controller-manager, kubelets, autoscalers) read that state, react, and write back the *observed* state, concurrently, forever. A production cluster can run **thousands of nodes** (Kubernetes officially supports up to 5,000 nodes / 150,000 Pods / 300,000 containers per cluster) with **hundreds of controllers** all watching for changes. The hard requirement: there must be exactly **one** source of truth, it must be **linearizable** (a controller that reads “Pod X is unscheduled” must not be reading stale data and double-schedule it), and it must survive control-plane machine failures without ever losing a committed write or splitting brain. Two schedulers each “winning” the same Pod, or a stale apiserver serving a deleted Secret, is a correctness disaster.

Kubernetes solves this by **not** baking coordination into the apiserver. It delegates *all* of it to one small, dedicated component: **etcd**. This is exactly the “why a coordination service exists” argument from §6 — the apiserver and controllers are stateless and horizontally scalable precisely *because* the hard consensus problem is concentrated in a 3-or-5-node etcd cluster that is the brain stem.

```
kubectl / controllers / scheduler / kubelets
| (watch + read + compare-and-swap writes)
v
+-----+ kube-apiserver (stateless, N replicas) -----+
| the ONLY component that talks to etcd; serializes all access |
+-----+-----+
| gRPC (linearizable read / txn)
v
+===== etcd cluster (Raft, 3 or 5 nodes) =====+
| LEADER --AppendEntries--> follower follower |
| | replicated, totally-ordered Raft log |
| idx: [Pod A] [Deploy B] [Lease kcm] [Node C] ... |
| committed once a MAJORITY (quorum) acks |
+=====+
```

How the module’s concepts map to it, one to one:

- **Raft log replication & commit (§4.2)** — Every write (`kubectl apply` , a controller status update) becomes an entry the etcd **leader** appends and replicates via `AppendEntries` . It is **committed only when a majority** of etcd members have it and it’s from the current term — the exact quorum + current-term rule from the module. This is what guarantees that once `kubectl apply` returns success, that object survives any single control-plane node loss.

- **Raft leader election (§4.1)** — etcd members run terms with randomized election timeouts. Lose the leader (machine dies, network blip) and a follower whose log is at-least-as-up-to-date (the election restriction) wins a new term in ~hundreds of ms. During that window etcd refuses writes — Kubernetes is a textbook **CP system** that picks consistency over availability, the C/A trade-off from §6.
- **Linearizable reads (§9 / §6)** — etcd serves linearizable reads by default; the apiserver relies on this so a `GET` never returns a value older than a prior committed write. Latency-sensitive paths (informer **list-watch**) instead stream a **watch from a revision** (etcd's MVCC revision numbers), the streaming-watch primitive from §6.
- **Leases (§9) + leader election as a recipe (§7)** — kube-controller-manager and kube-scheduler each run as multiple replicas but must have only **one active leader** (you don't want two schedulers). They implement leader election on top of an etcd-backed **Lease** object with a TTL the holder must renew (`leaseDurationSeconds` , `renewDeadline`). Lose the renew (crash, GC pause) → the lease expires → a standby acquires it. This is precisely the “lease key in etcd” pattern from §7, and etcd's own internal session liveness uses leases too. Kubernetes even exposes a first-class `coordination.k8s.io/Lease` API object for it.
- **Fencing via resourceVersion (§8)** — Kubernetes can't fence at the disk like the abstract example, but it does the equivalent at the API layer: every object carries a `resourceVersion` (backed by etcd's monotonic revision). Updates use **optimistic concurrency** — a write must state the version it read; if etcd's revision moved on, the write is **rejected with a 409 Conflict** and the controller must re-read and retry. A zombie controller that paused and woke with a stale view carries a stale `resourceVersion` and **loses the compare-and-swap** — the monotonic-token fencing idea from §8, enforced by the `resource` (etcd), not the caller.

The key trade-off they accepted. Kubernetes funnels *every* state operation in the entire cluster through one Raft consensus log. That gives bulletproof linearizable consistency and a single source of truth — but it makes etcd the **scalability ceiling and single brain stem** (§6's “keep it small, don't overload it”). Raft's single-leader write path means write throughput does **not** scale by adding etcd nodes (more nodes = *bigger* quorum = *slower* writes), which is why etcd clusters stay at 3 or 5. They traded write scalability and the convenience of a general database for absolute correctness, and they pay for it operationally: etcd has a hard default storage quota (**8 GiB**) and is acutely sensitive to disk fsync and network latency. Overloading it — storing huge objects, high-churn writes, or too many watches — has caused real large-cluster outages where the whole control plane freezes because the brain stem is saturated. This is §12's “putting too much in the coordination service” war story, lived at scale.

Real numbers / results. etcd targets and achieves **~10,000 writes/sec** on commodity SSDs with single-digit-millisecond commit latency when disk fsync is fast; reads are far higher. The 5,000-node Kubernetes scalability target is gated largely by etcd and apiserver behavior, not the kubelets. SLO work on huge clusters consistently traces tail latency back to etcd fsync/compaction and watch fan-out — reinforcing that the consensus layer is the governor on the whole system.

Lessons

- **Concentrate consensus, then scale the stateless parts around it.** Kubernetes is “scalable” because the *only* place that does Raft is a tiny etcd cluster; apiservers, schedulers, and controllers are cheap to

replicate precisely because they own no consensus state.

- **Optimistic concurrency + a monotonic version is fencing in disguise.** `resourceVersion` 409-conflict retries are the same safety property as fencing tokens — the resource rejects stale writers — without needing a distributed mutex.
- **A consensus store is a brain stem, not a database.** Its quota, write-throughput, and latency are hard ceilings; large values, high churn, or watch storms can freeze everything that depends on it. Keep entries small and write rates modest.
- **CP is a deliberate choice.** Kubernetes would rather refuse writes during an etcd quorum loss than risk two schedulers acting on divergent state — availability sacrificed for the one-source-of-truth invariant.

13. Test yourself

1. Why can't a deterministic protocol guarantee consensus in a fully asynchronous system with one possible crash? What do Raft/Paxos give up to cope? (*Hint: FLP; can't distinguish slow from dead; they keep safety, make liveness conditional on partial synchrony + randomized timeouts.*)
 2. Two majorities of a 5-node cluster — how many nodes must they share, and why does that guarantee agreement? (*Hint: ≥ 1 ; intersecting quorums force a later proposer to see any value that could've been chosen.*)
 3. A Raft follower is far behind and a leader dies. Why can't the stale follower win the election and erase committed entries? (*Hint: election restriction — vote only for a candidate whose log is at least as up-to-date; Leader Completeness.*)
 4. A client holds an etcd lease-based lock, GC-pauses 20s past the TTL, wakes, and writes. With and without fencing tokens, what happens? (*Hint: without $\rightarrow$ double-writer corruption; with $\rightarrow$ resource rejects the stale lower token.*)
 5. When is a single-Redis `SET NX PX` lock acceptable, and when is it a bug waiting to happen? (*Hint: fine for efficiency/dedupe hints; unsafe for correctness — no fencing, clock/pause assumptions.*)
 6. Lamport timestamp `L(a) < L(b)`. Does that prove `a` causally happened before `b`? What clock would let you detect they're concurrent? (*Hint: no, converse doesn't hold; vector clocks detect concurrency via incomparability.*)
 7. How does a leader lease let a node serve a linearizable read without contacting a quorum, and what's the failure mode? (*Hint: lease guarantees no newer leader yet; failure = clock drift $\rightarrow$ two leaders; mitigate with conservative expiry / TrueTime / epoch fencing.*)
 8. Why is phi-accrual detection better than a fixed heartbeat timeout, and what does ϕ represent? (*Hint: adaptive to network jitter; ϕ is the log-scale suspicion / probability the node is wrongly suspected.*)
-

14. Further reading

- Fischer, Lynch, Paterson — *Impossibility of Distributed Consensus with One Faulty Process* (1985).
- Lamport — *The Part-Time Parliament* (1998) and *Paxos Made Simple* (2001).

- Ongaro & Ousterhout — *In Search of an Understandable Consensus Algorithm (Raft)* (2014); plus raft.github.io visualizations.
 - Junqueira, Reed, Serafini — *ZAB: High-performance broadcast for primary-backup systems* (2011); ZooKeeper paper (Hunt et al., 2010).
 - Burrows — *The Chubby Lock Service* (2006).
 - Corbett et al. — *Spanner: Google's Globally-Distributed Database* (2012) — TrueTime & commit-wait.
 - Kulkarni et al. — *Logical Physical Clocks (HLC)* (2014).
 - Lamport — *Time, Clocks, and the Ordering of Events in a Distributed System* (1978).
 - Das, Gupta, Motivala — *SWIM* (2002); Hayashibara et al. — *The ϕ Accrual Failure Detector* (2004).
 - Martin Kleppmann — *How to do distributed locking* (Redlock critique, 2016); antirez's rebuttal.
 - Kleppmann, *Designing Data-Intensive Applications* — Ch. 8 (trouble with distributed systems) & Ch. 9 (consistency and consensus).
 - Official docs: etcd Raft & lease docs; ZooKeeper recipes; Consul architecture; Jepsen reports (jepsen.io).
-

Next: [11-distributed-transactions-and-sagas.md](#) builds on consensus to commit across shards (2PC, Percolator, sagas). Back-reference [09-cap-pace1c-and-consistency-models.md](#) for the consistency guarantees consensus delivers.

Module 11 — Distributed Transactions, Sagas & Idempotency

Source: [readme/research/system-design/11-distributed-transactions-and-idempotency.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains what happens when a software system needs to save data in two different places at the same time — and why that is much harder than it sounds. It matters because when this goes wrong, customers get charged twice, orders disappear silently, or stock never gets reserved — exactly the kind of invisible damage that kills trust in a store.

The main parts explained simply:

- **The dual-write problem** — Writing to two systems (say, your database and a message queue) in a row is never safe. If the first write succeeds and the second one fails, your data is out of sync. This causes orders that were paid for but never fulfilled. There is no simple fix — you have to change how you write data in the first place.
- **Two-Phase Commit (2PC)** — An old approach where a “coordinator” asks all involved systems “are you ready?” before everyone commits together. The fatal flaw: if the coordinator crashes mid-way, every system is stuck, holding locks and blocking all other work. Expensive and unreliable at scale — avoided on critical paths.
- **The Saga pattern** — Instead of one big all-or-nothing transaction, you break the work into small steps. If a later step fails, you run “undo” steps in reverse to clean up. Think of it like a paper trail of receipts — if something goes wrong, you issue a refund or cancel rather than pretending it never happened.
- **Orchestration vs Choreography** — Two ways to coordinate saga steps. Orchestration means one central “brain” directs each step (easier to trace and debug). Choreography means each service reacts to events from the previous one (simpler but harder to follow when something goes wrong). Use orchestration once you have four or more steps.
- **Transactional Outbox** — The correct fix for the dual-write problem. Instead of writing to the database and sending a message separately, you write both into the database in one step. A background process then reads the database and sends the message. One write, no split.
- **Idempotency keys** — A unique ID attached to each operation so the system can recognise a retry and ignore it instead of running the action again. Without this, a customer clicking “Pay” twice, or a network timeout triggering a retry, results in a double charge.
- **Reconciliation** — A background job that periodically compares your records against external systems (like a payment gateway) and flags or fixes mismatches. Treat your live system as “best effort” and reconciliation as the safety net that catches drift before it becomes a serious problem.

What to do with this: Never write to two separate systems in the same code block without using the Outbox pattern. Always attach idempotency keys to payment and order operations so retries are safe and double-charges are impossible.

What you'll learn: Why the ACID transaction you take for granted in a single Postgres database becomes nearly impossible the moment your write spans two services or two databases, and what real systems do instead. You'll learn 2PC/3PC and why they block, the Saga pattern with compensations, the Transactional Outbox + CDC pattern that fixes the dual-write problem, the three delivery semantics, and how idempotency keys turn an unreliable at-least-once world into something you can reason about.

Prerequisites: Read [09-cap-pace1c-and-consistency-models.md](#) (consistency models, linearizability vs eventual) and [10-replication-and-consensus.md](#) (Raft/Paxos, why consensus is expensive) first. Familiarity with [08-message-queues-and-streaming.md](#) (Kafka, delivery guarantees) helps for the Outbox/CDC section.

1. Intuition: why a single-DB transaction is a luxury

In a monolith on one Postgres instance you write:

```
BEGIN;  
UPDATE accounts SET balance = balance - 100 WHERE id = 1;  
UPDATE accounts SET balance = balance + 100 WHERE id = 2;  
COMMIT;
```

Either both rows change or neither does. The database gives you **atomicity** (all-or-nothing) and **isolation** (concurrent transactions don't see each other's half-done state) for free, because one process — the storage engine — owns all the data, the write-ahead log, and the lock manager. There is a single point that knows the truth.

Now split `accounts` across two services — `BillingService` (Postgres A) and `WalletService` (Postgres B), each with its own database. There is no shared lock manager and no shared WAL. You want “debit A, credit B, all-or-nothing,” but you now face the **two generals problem**: any message between the two can be lost, delayed, or duplicated, and a service can crash *between* doing its work and telling the other side. There is no protocol that gives you guaranteed agreement over an unreliable network with crash faults in bounded time — this is the FLP impossibility result in practical clothing.

The analogy: A single-DB transaction is two people in the same room signing a contract — one person watches both pens touch paper. A distributed transaction is two people signing by mail, where letters get lost and either signer might have a heart attack mid-signature. You cannot make “both sign or neither signs” bulletproof; you can only make it *recoverable* and *eventually consistent*.

2. The dual-write problem (the bug behind 80% of these incidents)

The most common naive attempt:

```
// Inside one request handler – looks innocent, is broken
DB::transaction(fn () => $order->save()); // write 1: local DB
Kafka::publish('order.created', $order); // write 2: message broker
```

These are **two separate systems with no shared transaction**. Four failure interleavings:

	DB commit OK	DB commit FAIL
Kafka OK	✓ consistent	✗ event without data (phantom)
Kafka FAIL	✗ data without event (silent lost update)	✓ consistent

The two failure cells are the killers. “Data without event” means inventory never gets reserved and the customer’s paid order silently never ships — exactly the *silent-lie* bug class this codebase guards against (see [CLAUDE.md §5](#)). Retrying after a partial failure doesn’t help: if the DB committed but Kafka failed, retrying the whole block double-saves the order.

There is no ordering of two independent writes that is safe. Whichever you do first can succeed while the second fails. The fix is to make one of them the source of truth and *derive* the other from it — that’s the Transactional Outbox (§6).

3. Two-Phase Commit (2PC) and why it blocks

2PC is the textbook attempt at real distributed atomicity. A **coordinator** drives all participants through two phases.

Coordinator	Participant A	Participant B
--- PREPARE ----->		
--- PREPARE ----->		
<-- VOTE YES -----		
<-- VOTE YES -----		(both flush
(decision: COMMIT, logged to disk)		to WAL,
--- COMMIT ----->		hold locks)
--- COMMIT ----->		
<-- ACK -----		
<-- ACK -----		

Phase 1 (prepare/voting): coordinator asks everyone “can you commit?” Each participant does the work, writes it to its WAL in a *prepared* (durable but uncommitted) state, holds its locks, and votes YES — a promise it can no longer back out of. **Phase 2 (commit):** if all voted YES, coordinator logs COMMIT and tells everyone to finalize; any NO → ABORT.

The fatal flaw: blocking on coordinator failure

If the coordinator crashes *after* a participant voted YES but *before* delivering the decision, the participant is stuck. It promised to commit, so it cannot unilaterally abort; it doesn’t know the decision, so it cannot

commit. It **holds its locks indefinitely**, blocking every other transaction touching those rows. This is the **uncertainty window**, and it is why 2PC is called a *blocking protocol*.

Concrete cost: each commit needs at minimum 2 round trips + 2 forced WAL flushes per participant. On a LAN that's ~2–10 ms; across regions (~70 ms RTT) a 2PC commit costs **150 ms+** and pins locks the whole time, collapsing throughput. This is why high-scale systems banned 2PC for the request path.

3PC and why nobody uses it

Three-Phase Commit inserts a **pre-commit** phase so participants learn the decision is imminent before committing, removing the indefinite block. But 3PC assumes **synchronous, bounded-delay networks** and is *not* safe under network partitions (it can produce split-brain commits). Modern systems either accept 2PC's blocking with a highly-available consensus-backed coordinator, or abandon distributed atomicity entirely.

Protocol	Round trips	Blocking?	Partition-safe?	Real use
2PC	2	Yes (coordinator crash pins locks)	No	XA/JTA, Postgres <code>PREPARE TRANSACTION</code> , Spanner intra-commit
3PC	3	No (in theory)	No (split-brain)	Essentially none in production
Saga	n (async)	No	Yes	Microservices everywhere
Consensus-backed 2PC	2 + Paxos	No (coordinator is replicated)	Yes	Google Spanner , CockroachDB, FoundationDB

Spanner's trick: it still runs 2PC, but the coordinator and each participant are themselves Paxos groups (see `10-replication-and-consensus.md`). A “coordinator crash” no longer loses the decision because the decision is replicated by consensus. It buys real cross-shard ACID at the cost of TrueTime atomic clocks and ~10 ms commit-wait. Most teams cannot afford that, which is why sagas dominate.

4. The Saga pattern: give up atomicity, keep recoverability

A **saga** replaces one ACID transaction with a *sequence of local transactions*, each in its own service/DB. If step N fails, you run **compensating transactions** for steps N-1 ... 1 to semantically undo them. There is no global rollback — there is forward progress or backward un-doing.

The key shift in thinking: a saga is **ACD, not ACID — it drops Isolation**. Intermediate states are visible to other transactions. An order can be “payment captured, inventory not yet reserved” for a moment, and another reader can see that. You manage the missing isolation with application-level techniques (semantic locks, status fields, commutative updates), covered in §8.

Compensations must be **semantic, not physical**. You can't “un-send” an email or “un-charge” a captured card — you send an apology / issue a refund. A compensation is a new business fact that counteracts a prior one.

Orchestration vs Choreography

ORCHESTRATION (central brain)

Saga Orchestrator

cmd|cmd|cmd| (explicit commands + replies)

▼ ▼ ▼

Pay Inv Ship

CHOREOGRAPHY (event chain, no brain)

```
OrderSvc --order.created--> [bus]
[bus] --> PaymentSvc
PaymentSvc --payment.ok--> [bus]
[bus] --> InventorySvc
InventorySvc --reserved--> [bus]
[bus] --> ShippingSvc
(each svc reacts to events,
no one owns the flow)
```

	Orchestration	Choreography
Control flow	Central orchestrator issues commands	Each service reacts to events
Visibility	Whole saga in one place — easy to trace/audit	Flow is emergent; hard to see end-to-end
Coupling	Orchestrator knows all participants	Services only know event contracts
Failure handling	Orchestrator runs compensations explicitly	Each service must emit failure events; compensation logic scattered
Cyclic-dependency risk	Low	High (events triggering events)
Best when	≥4 steps, complex branching, need audit trail	2–3 steps, simple linear flow, max decoupling

Rule of thumb: choreography for short, stable flows; **orchestration once it gets complex** — the central state machine is worth its weight when you’re debugging a stuck order at 2 a.m. Real implementations: **AWS Step Functions, Temporal/Cadence** (durable orchestration with automatic retries + compensation), **Netflix Conductor, Camunda/Zeebe**.

5. Worked example: Order → Payment → Inventory saga (orchestrated)

This is the canonical interview question and maps directly to this project’s order/payment/inventory chain (see [readme/ORDER_AND_JOB_WORKFLOW.md](#) , [readme/PAYMENT_GATEWAYS.md](#)).

Happy path + state machine

```
[PENDING] --create--> [PAYMENT_AUTHORIZING]
  payment.authorize OK -> [INVENTORY_RESERVING]
    inventory.reserve OK -> [SHIPPING_SCHEDULING]
      shipping.schedule OK -> [CONFIRMED]  ✓ terminal success

Failure at inventory step (out of stock):
[INVENTORY_RESERVING] --reserve FAIL--> [COMPENSATING]
  -> payment.refund (compensate step 1)
  -> [CANCELLED]  ✓ terminal failure
```

Steps, compensations, and the tricky asymmetry

Step	Forward action	Compensation	Notes
1	<code>payment.authorize</code> (hold, don't capture)	<code>payment.void</code> / <code>refund</code>	Authorize-then-capture lets you void cheaply before capture
2	<code>inventory.reserve</code> (decrement available)	<code>inventory.release</code>	Reserve, don't deduct stock outright
3	<code>shipping.schedule</code>	<code>shipping.cancel</code>	Pivot point — see below
4	<code>payment.capture</code>	(none — past the pivot)	After capture, no compensation; only forward fixes

Pivot transaction: the step after which the saga *must* complete and can no longer be compensated. Order steps so all **compensatable** actions come first, then the **pivot**, then **retriable** (must-eventually-succeed) actions. Capturing payment after inventory is reserved is the classic pivot: once you've committed to the customer's money, you push forward (retry shipping) rather than unwind.

Why authorize-before-capture matters: if you *capture* in step 1 and inventory fails in step 2, your compensation is a refund — slow, fee-incurring, and a bad customer experience. Authorizing first makes the compensation a cheap *void*. This sequencing decision is the difference between a clean saga and a refund-storm incident.

Pseudocode for the orchestrator (idempotent + durable)

```
function runOrderSaga(string $orderId): void {
    $saga = SagaState::loadOrCreate($orderId);    // durable, resumable
    try {
        $saga->step('payment.authorize', fn() =>
            $payments->authorize($orderId, key: "$orderId:auth")); // idempotency key
        $saga->step('inventory.reserve', fn() =>
            $inventory->reserve($orderId, key: "$orderId:reserve"));
        $saga->step('shipping.schedule', fn() =>
            $shipping->schedule($orderId, key: "$orderId:ship"));
        $saga->step('payment.capture', fn() => // pivot
            $payments->capture($orderId, key: "$orderId:capture"));
        $saga->complete();
    } catch (StepFailed $e) {
        $saga->compensateInReverse(); // void/release only the completed steps
    }
}
```

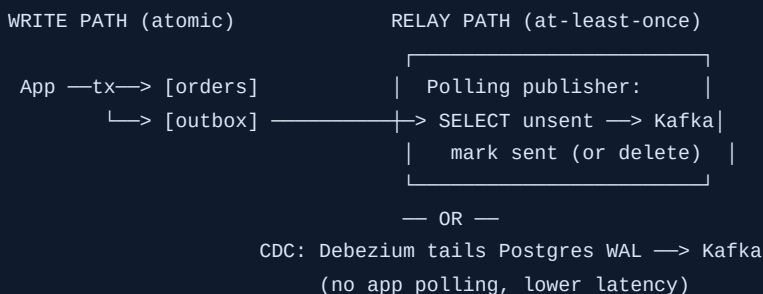
Every external call carries an **idempotency key** (§7) and the saga state is **persisted after each step** so a crashed orchestrator resumes mid-flight instead of restarting the saga and double-charging.

6. Transactional Outbox + CDC — the correct fix for dual-write

You cannot atomically write your DB *and* publish to Kafka. So **only write the DB**, and include the event as a row in the *same local transaction*:

```
BEGIN;  
INSERT INTO orders (...) VALUES (...);  
INSERT INTO outbox (id, topic, payload, created_at)  
VALUES (uuid(), 'order.created', '{...}', now());  
COMMIT; -- one atomic local transaction. No dual write.
```

A separate **relay** then ships outbox rows to the broker and marks them sent. Two ways to relay:



Relay method	Latency	Load	Ops cost	Use when
Polling publisher	100 ms–1 s (poll interval)	Periodic SELECTs	Low — just your app	Small scale, no CDC infra
CDC (Debezium tailing WAL)	10–50 ms	Near-zero on DB	Higher (Kafka Connect)	High throughput, low latency

Critical property: the outbox guarantees **at-least-once** delivery, never exactly-once. The relay can crash *after* publishing but *before* marking the row sent → it republishes on restart → consumers see duplicates. This is unavoidable, and it's *why idempotency on the consumer is mandatory*, not optional. Outbox + idempotent consumers = effectively-once processing.

Real systems: Debezium (the de-facto CDC tool), DynamoDB Streams, MongoDB change streams, Postgres logical replication slots. The “listen to the log” idea is the same one behind Kafka itself — see [08-message-queues-and-streaming.md](#) and Kleppmann's “Turning the database inside-out.”

7. Delivery semantics & idempotency keys

The three guarantees

Semantic	What it means	Cost	Reality
At-most-once	Fire and forget; may drop	Cheapest, no retries	OK for metrics/logs where loss is fine
At-least-once	Retries until ack; may duplicate	Moderate	The default for any reliable system
Exactly-once	Delivered+processed exactly 1×	Expensive / often a myth	Only <i>within</i> a closed system (Kafka EOS)

The deep truth: **exactly-once delivery over a network is impossible** (you can't distinguish a lost message from a lost ack — the two generals problem again). What you *can* get is **exactly-once processing** = at-least-once delivery + **idempotent / deduplicated consumers**. Kafka's “exactly-once semantics” works only because producer, broker, and consumer offsets are all inside Kafka's transactional boundary; the moment your side effect leaves Kafka (charge a card, send an email), you're back to needing idempotency.

Idempotency keys & the dedup store

An **idempotency key** is a client-supplied unique ID for an *operation* (not a record). The server records the result keyed by it; a retry with the same key returns the stored result instead of re-executing.

```
POST /charges    Idempotency-Key: 9f3c-...    amount=100
```

```
server:
  SELECT result FROM idem WHERE key='9f3c-...';
  if found -> return stored result (no re-charge)
  else:
    BEGIN;
    INSERT INTO idem(key, status='in_progress') -- UNIQUE constraint!
    ... do the charge ...
    UPDATE idem SET status='done', result=...;
    COMMIT;
    return result
```

Design rules that bite people:

- **Scope the key to the operation, store the response.** Stripe returns the *original* response (even the original error) for 24 h. The key must capture intent so a retry of “charge \$100” doesn’t accidentally satisfy a later “charge \$200.”
- **Use a UNIQUE constraint as the dedup gate**, plus a “in_progress” state with a TTL/lease, so two *concurrent* retries don’t both execute (the constraint makes one lose). Don’t rely on `SELECT then INSERT` — that’s a race.
- **Persist before side effect.** Write the `in_progress` row *before* charging; otherwise a crash after charging but before recording the key lets the retry charge again.
- **Dedup store choices:** Redis `SETNX key ... EX ttl` (fast, bounded window — fine if your retries arrive within the TTL); a relational table with UNIQUE (durable, slower); Kafka consumer = `(topic, partition, offset)` or a business key in a processed-IDs table.

This project’s payments audit flagged exactly this gap — see

`readme/PAYMENTS_BILLING_FINANCIAL_CORRECTNESS.md` : a missing idempotency key on charge means a retried checkout can double-charge.

8. Eventual consistency, reconciliation & distributed deadlocks

Sagas leak intermediate state — manage it

Because sagas have no isolation, you need countermeasures (from the saga literature, Garcia-Molina & Salem 1987):

- **Semantic lock:** mark the record `PENDING` / `reserved` so other transactions know it’s mid-saga and either wait or skip it.
- **Commutative updates:** design operations so order doesn’t matter (`balance += x` commutes; `balance = x` doesn’t).
- **Reread / version check:** before compensating, reread to handle the case where another actor changed the row (optimistic concurrency, version columns).

- **By-value vs by-status:** decide whether a reader trusts the in-flight value or waits for terminal status.

Reconciliation: the safety net you must build

Distributed state drifts. A reserve succeeds but the “reserved” event is lost; a refund fires twice. **Run a periodic reconciliation job** that compares systems of record (e.g. payment-gateway settled amounts vs your orders) and emits a repair or an alert. Treat your live flow as best-effort and reconciliation as the eventual truth — every serious payments/inventory system has a nightly recon. Idempotency makes repair re-runnable; without it, recon itself can double-apply.

Distributed deadlocks

When transactions hold locks across services, you can deadlock with no single lock manager to detect the cycle:

```
Saga A: holds lock(order-1) on SvcX, waits lock(inv-9) on SvcY
Saga B: holds lock(inv-9) on SvcY, waits lock(order-1) on SvcX
→ cycle spanning two services, nobody sees it
```

Mitigations: **lock ordering** (always acquire resources in a global canonical order — kills cycles), **timeouts + retry with backoff** (the pragmatic default; let one victim abort), **wait-die / wound-wait** timestamp schemes (older txn priority, prevents cycles deterministically), or **avoid cross-service locks entirely** by using sagas with semantic locks + compensation instead of holding real DB locks.

9. Common pitfalls / war stories

- **The dual-write that “worked in testing.”** DB-then-publish passes every test because failures are rare in dev. In prod, the broker hiccups for 200 ms during a deploy and 4,000 paid orders never get inventory reserved. Fix: outbox.
- **Non-idempotent compensation.** Refund handler with no idempotency key; the orchestrator retries the compensation after a timeout and refunds twice. The void/refund must carry `"$orderId:refund"`.
- **Capturing before the pivot.** Capturing payment in step 1, then inventory fails → refund storm, gateway fees, angry customers. Authorize first, capture after the pivot.
- **Treating Kafka “exactly-once” as end-to-end.** EOS covers Kafka-to-Kafka. Your `sendEmail()` inside the consumer is a side effect outside Kafka — at-least-once → duplicate emails. Add a dedup store.
- **Idempotency key = the record ID.** Using `order_id` as the key means a legitimate second, different operation on the same order is silently swallowed. The key is *per-operation*.
- **Orchestrator with non-durable state.** In-memory saga state → orchestrator restarts → saga re-runs from step 0 → double charge. Persist state after every step; use Temporal/Step Functions if you can.
- **Forgetting the in_progress lease.** Two concurrent retries both pass the `SELECT`, both charge. Use the UNIQUE constraint + status, not check-then-act.
- **2PC across regions on the hot path.** Someone adds XA to “make it correct”; p99 latency triples and a coordinator crash freezes the table. 2PC is a last resort, not a default.

🧩 Case Study: DoorDash order fulfillment & payments

The problem. DoorDash processes on the order of **2+ billion orders a year** (peaks of millions of orders per day, tens of thousands of orders per minute during dinner rush). A single “order” is not one write — it spans a sprawl of independently-owned microservices: payment authorization, merchant order injection (the restaurant’s POS), Dasher (courier) assignment, delivery routing, and post-delivery capture/payout. Each lives in its own service with its own database. There is **no shared transaction** across “charge the customer,” “tell the restaurant to cook,” and “assign a Dasher.” Early on, DoorDash hit exactly the failure cells from §2: the classic dual-write bug where an order committed locally but the downstream event (charge, or POS injection) was lost or fired twice — producing customers charged for food never cooked, restaurants cooking orders never paid, and duplicate charges on retries.

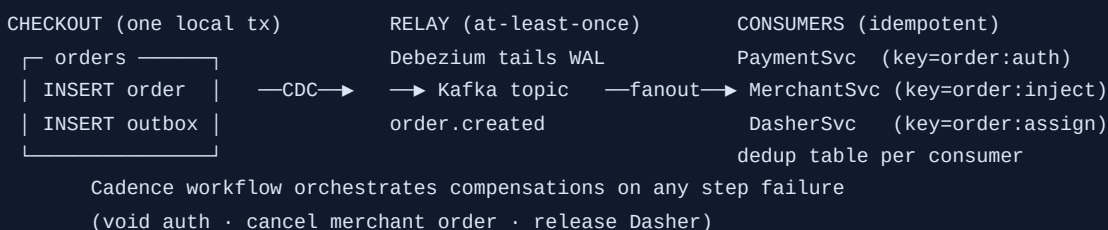
Applying this module’s concepts.

This is the §5 worked example at industrial scale. The fulfillment flow is a **saga** (§4): a sequence of local transactions — authorize payment, inject order to merchant, assign Dasher, deliver, capture payment — with **compensations** when a step fails (void the auth, cancel the merchant order, refund). DoorDash drops Isolation exactly as §4 describes: an order legitimately sits in “paid-but-not-yet-assigned” for seconds, visible to other systems, managed with **status fields and semantic locks** (§8) rather than cross-service DB locks.

For the control flow they moved toward **orchestration over choreography** (§4) using **Cadence** (the durable-workflow engine that Temporal forked from). This is the rule-of-thumb in action: with ≥ 4 steps, branching (no-Dasher-available, restaurant-rejected, customer-cancel), and a hard need to trace a stuck order at dinner-rush 7 p.m., the central state machine earns its keep. Cadence persists workflow state after every step, so an orchestrator crash **resumes mid-saga** instead of restarting from step 0 and double-charging — the “durable orchestrator” requirement from §5’s pseudocode, solved by an off-the-shelf engine.

The **pivot transaction** (§5) is real money: DoorDash *authorizes* at checkout and *captures* after delivery. Authorize-then-capture means a pre-pivot failure (restaurant closed, no Dasher) compensates with a cheap **void**, not a fee-incurring **refund** — precisely the asymmetry §5 warns about.

To kill the dual-write, DoorDash adopted the **transactional outbox + CDC** pattern (§6). Services write their state row and an outbox event **in one local transaction**, and **Debezium tails the Postgres/Aurora WAL** to publish to **Kafka** — no app polling, low latency. Their internal event-bus / “Iguazu” pipeline is this idea generalized: the database log is the source of truth, events are *derived*, never dual-written.



Because the outbox is **at-least-once** (the relay can republish after a crash), every consumer is **idempotent** (§7). Each downstream action carries an **idempotency key** scoped to the *operation* (`order:auth` , `order:capture` , `order:inject`) — not the order id — backed by a UNIQUE-constrained dedup row so two concurrent retries can't both execute. This is the **exactly-once illusion** spelled out in §7: at-least-once Kafka delivery + idempotent consumers = effectively-once *processing*, even though exactly-once *delivery* over the network is impossible.

Finally, DoorDash runs **reconciliation** (§8) as the eventual truth: nightly jobs compare gateway-settled amounts against captured orders and merchant payouts, repairing drift. Idempotency is what makes those repairs safely re-runnable.

The key trade-off. They **gave up cross-service atomicity and Isolation** (no 2PC on the order hot path — §3's blocking cost is unaffordable at tens of thousands of orders/minute) in exchange for **availability, low latency, and horizontal scale**. The price is real: temporary inconsistency windows, compensation logic to maintain, and a hard dependency on idempotency + reconciliation being correct everywhere. They accepted “eventually consistent and recoverable” over “atomic but blocking” — the central bargain of §4.

Results (qualitative + reported). Moving off dual-writes to outbox/CDC + Cadence-orchestrated sagas eliminated the phantom-event and double-charge incident class; the CDC pipeline streams change events at very high throughput (their event platform handles billions of events/day) with WAL-tailing latency in the tens of milliseconds rather than poll-interval seconds. Durable workflows let a crashed orchestrator resume without re-charging, and idempotency keys collapse retry storms during partial outages instead of amplifying them into duplicate charges.

Lessons

- **Outbox + CDC + idempotent consumers is the production-grade fix for dual-write** — at DoorDash's scale it's not optional; the moment a side effect leaves Kafka (charging a card), you're back to needing your own idempotency key.
- **Reach for a durable orchestrator (Cadence/Temporal/Step Functions) once the saga has ≥4 steps and branching** — hand-rolled in-memory saga state will eventually restart-from-zero and double-charge.
- **Order steps so all compensatable actions precede the pivot** — authorize-before-capture turns a costly refund into a cheap void, which is the difference between a clean cancel and a refund storm.
- **Treat the live flow as best-effort and build reconciliation as the real source of truth** — distributed state always drifts; idempotency is what makes nightly repair safe to run.

10. Test yourself

1. Why is exactly-once *delivery* impossible but exactly-once *processing* achievable? (*Hint: lost message vs lost ack are indistinguishable; idempotent consumer collapses duplicates.*)
2. A participant voted YES in 2PC and the coordinator vanished. What can the participant safely do, and what is the cost? (*Hint: nothing — it must block holding locks; this is the uncertainty window.*)
3. In the order saga, why authorize payment before reserving inventory rather than capturing it? (*Hint: cheap void vs costly refund as the compensation.*)

4. Your service does `INSERT order; publish event` in code (not the same tx). Give two distinct failure outcomes and the fix. (*Hint: phantom event / lost event; transactional outbox.*)
 5. Outbox guarantees at-least-once. What *must* the consumer do, and why can't the outbox give exactly-once itself? (*Hint: relay can crash after publish before marking sent → dedup on consumer.*)
 6. Design an idempotency-key flow that's safe under two concurrent retries. (*Hint: persist `in_progress` row under a `UNIQUE` constraint before the side effect; second insert loses.*)
 7. What does a saga give up that an ACID transaction provides, and name two ways to compensate for the loss. (*Hint: Isolation; semantic locks, commutative updates, reread/version.*)
 8. Two sagas deadlock across SvcX and SvcY with no global lock manager. Give two mitigations. (*Hint: global lock ordering; timeout+backoff or wound-wait.*)
-

11. Further reading

- **DDIA (Kleppmann)** — Ch. 7 *Transactions* (isolation levels, 2PC), Ch. 8 *The Trouble with Distributed Systems*, Ch. 9 *Consistency and Consensus* (atomic commit, FLP). The single best treatment.
- **Garcia-Molina & Salem, “Sagas” (1987)** — the original paper; compensation and semantic locks.
- **Chris Richardson, *Microservices Patterns*** — Ch. 4 (Saga), Ch. 3 (Transactional Outbox); also [microservices.io](https://microservices.io/patterns) patterns pages.
- **Gray & Lamport, “Consensus on Transaction Commit”** — Paxos-Commit, the consensus-backed coordinator that Spanner-class systems use.
- **Spanner paper (Corbett et al., OSDI 2012)** — TrueTime + 2PC over Paxos groups for real cross-shard ACID.
- **Kleppmann, “Turning the database inside-out”** — the log-as-source-of-truth mental model behind Outbox/CDC.
- **Stripe** — “**Idempotency**” docs and “**Designing robust and predictable APIs with idempotency**” (Brandur Leach) — production idempotency-key design.
- **Debezium documentation** — outbox event router + CDC patterns.
- **Confluent** — “**Exactly-once semantics in Kafka**” — what EOS does and doesn't cover.
- Cross-links: [09-cap-pacelc-and-consistency-models.md](#) , [10-replication-and-consensus.md](#) , [08-message-queues-and-streaming.md](#) .

12 · Messaging, Queues & Event-Driven Architecture

Source: [readme/research/system-design/12-messaging-and-event-driven.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains how software systems pass work between their parts without making everything wait for everything else. Think of it like a print shop tray: a customer drops off an order and leaves, while the press operator picks it up when ready — nobody blocks anyone. Understanding this matters because it is the foundation of how large systems stay fast and reliable even when parts fail.

The main parts explained simply:

- **Message queue vs message log** — A queue (like RabbitMQ or Amazon SQS) hands a job to one worker and deletes it once done — good for one-time tasks like “generate this thumbnail.” A log (like Apache Kafka) keeps all messages in order and lets many different readers each read at their own speed — good for events that many systems care about, like “an order was placed.”
- **Pub/Sub vs Competing workers** — You can send one message to every interested system at once (pub/sub, like broadcasting), or split work across many workers so each one handles a different piece (competing consumers, like a team sharing a task list). Most systems use both depending on the situation.
- **Kafka internals: topics, partitions, and offsets** — Kafka splits a stream into “partitions” (buckets), each handled by one worker at a time. A worker tracks how far it has read using an “offset” (like a bookmark). More partitions = more workers = more speed, but there are trade-offs.
- **Partition key — your most important Kafka decision** — Which bucket a message goes into depends on the key you choose. Choosing badly can overload one worker while others sit idle (called a “hot partition”). Choose a key that spreads work evenly or groups related events correctly.
- **Delivery guarantees** — Systems can promise: “maybe deliver once” (fast but lossy), “deliver at least once” (safe but may duplicate), or “deliver exactly once” (hard and complex). In practice, build for “at least once” and make your code safe to run twice instead of trusting “exactly once.”
- **Backpressure and bounded queues** — When messages arrive faster than workers can handle them, the backlog grows. An unbounded backlog eventually crashes the system. The fix: limit the queue size, monitor how far behind workers are falling, and fail gracefully rather than silently.
- **Retries with jitter** — When something fails, do not retry immediately and all at once. Spread retries randomly over time (called “jitter”) to avoid all workers hammering a recovering system at the same second, which would just break it again.
- **Dead-letter queues (DLQ)** — Some messages can never be processed (bad data, a deleted record, a bug). After a set number of failed attempts, move them to a “dead-letter” holding area instead of retrying forever. Always monitor this area and build a way to replay those messages once the problem is fixed.
- **Kafka exactly-once — what it really covers** — Kafka can guarantee a message is processed exactly once within Kafka itself, but the moment your code writes to a database or calls another service, that guarantee does not cross over. You still need to make those external actions safe to repeat.
- **LinkedIn and the origin of Kafka** — LinkedIn invented Kafka because they had dozens of systems all trying to share the same data, wired together in a tangled mess. A central replayable log collapsed

that mess from “every pair wired separately” to “everyone reads from one place.” The lesson: a shared event log simplifies integrations and lets late-arriving systems catch up by replaying history.

What to do with this: When building any feature that fires off background work (emails, thumbnail generation, order processing), use a queue or log rather than doing it inline — it keeps the user-facing response fast and the system resilient. Always make background workers safe to run twice on the same job, and always set up alerts on failed/stuck messages so problems surface immediately rather than silently.

What you’ll learn: How asynchronous messaging decouples services, the deep distinction between a *queue* (RabbitMQ/SQS) and a *log* (Kafka), and the full set of guarantees — delivery semantics, ordering, exactly-once — with the failure modes that break them in production. By the end you should be able to choose a partition key under pressure, design a dead-letter strategy, reason about backpressure, and defend a Kafka topology in a staff-level design review.

Prerequisites: Read [09-cap-pacelc-and-consistency-models.md](#) (consistency vocabulary), [10-replication-and-consensus.md](#) (ISR/Raft, leader election), and [11-idempotency-and-distributed-transactions.md](#) (idempotency keys, the outbox pattern). This module leans on all three.

1. Why asynchronous messaging exists (intuition)

Imagine a print shop. A customer drops off an order at the counter. The clerk does *not* make the customer stand there while the order is designed, printed, dried, and boxed. The clerk writes a ticket, drops it in a **tray**, and the customer leaves. The press operator picks up tickets from the tray when free. The tray is a **message queue**: it decouples *arrival rate* from *processing rate*, survives the operator going on break (durability), and lets you add a second operator to drain the tray faster (horizontal scale).

Synchronous RPC is the customer standing at the counter. It couples availability (if the press is down, the counter is down), couples latency (slow press = slow counter), and couples capacity (one customer at a time). Messaging breaks all three couplings. The price you pay is **eventual** processing and a whole zoo of new failure modes — duplicates, reordering, poison messages, unbounded backlogs — which is what the rest of this module is about.

SYNCHRONOUS (coupled)

Client —RPC→ Service
(blocks until done)
fate-shared availability

ASYNCHRONOUS (decoupled)

Client → [QUEUE/LOG] → Consumer
(returns immediately) (drains at its own pace)
independent availability

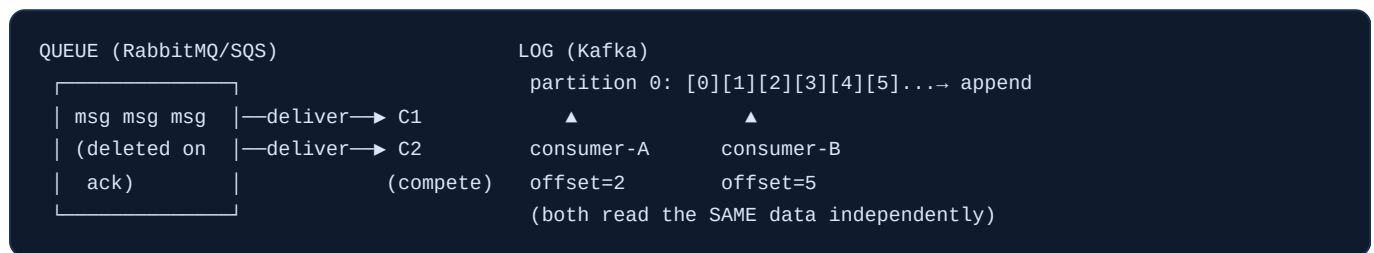
In Print-Flow-360 terms: an order is placed → an event is published → thumbnail generation (the [pdf-service](#)), notification email, inventory decrement, and analytics all react *independently*. None blocks checkout; none can take checkout down.

2. Two fundamentally different primitives: Queue vs Log

This is the single most misunderstood thing in the space. They look similar (you put messages in, you get messages out) but the data model is opposite.

A **queue** (RabbitMQ, ActiveMQ, AWS SQS) is *destructive read*: a broker holds messages, hands each to one consumer, and **deletes** it on acknowledgment. The broker tracks per-message state. It is fundamentally a smart broker / dumb consumer.

A **log** (Apache Kafka, AWS Kinesis, Redpanda, Apache Pulsar) is an *append-only, replayable* ordered sequence. Reads are **non-destructive** — the consumer tracks its own position (offset). Messages stay until a retention policy evicts them, independent of who read them. It is a dumb broker / smart consumer.



Dimension	Queue (RabbitMQ / SQS)	Log (Kafka / Kinesis)
Read model	Destructive (deleted on ack)	Non-destructive (offset cursor)
Replay history	No (gone after ack)	Yes (re-seek to old offset)
Multiple independent consumers	Needs fan-out (one queue each)	Native (each group has own offsets)
Ordering	Per-queue, lost under competing consumers	Strict per partition
Throughput ceiling	~10k–50k msg/s/queue typical	Millions/s (sequential disk + batching)
Per-message TTL / priority / delay	First-class (RabbitMQ)	Awkward (no per-msg priority)
Retention model	Until consumed	Time/size/compaction based
Best for	Task distribution, RPC-style work, priority jobs	Event streaming, event sourcing, replayable pipelines, many subscribers

Rule of thumb: if the message is a *command to do one unit of work once* (“generate this thumbnail”), a queue fits. If it’s a *fact that many parties care about and may want to re-read* (“order #123 was placed”), a log fits. Real systems often use both: Kafka as the event backbone, SQS/Celery for discrete worker jobs.

3. Pub/Sub vs Point-to-Point

Two delivery topologies, orthogonal to the queue/log choice:

- **Point-to-point (competing consumers):** one message → exactly one of N consumers. This is how you *parallelize* work. N workers pull from one queue; each message handled once. Add workers to scale throughput linearly (until the broker or downstream DB is the bottleneck).
- **Publish/subscribe (fan-out):** one message → *every* subscriber gets a copy. This is how you *decouple* independent reactions to the same fact.

COMPETING CONSUMERS (point-to-point)

```
Q → W1 (gets msg 1,4,7)
Q → W2 (gets msg 2,5,8)
Q → W3 (gets msg 3,6,9)
→ scale THROUGHPUT
```

PUB/SUB (fan-out)

```
topic → EmailSvc (all get every msg)
      → InventorySvc
      → AnalyticsSvc
      → DECOUPLE reactions
```

Kafka unifies both with the **consumer group**: consumers *in the same group* compete (point-to-point — each partition goes to one member); consumers in *different groups* each see the full stream (pub/sub). RabbitMQ models it with exchanges: a **direct** queue gives competing consumers; a **fanout** exchange copies to every bound queue. SNS → SQS on AWS is the canonical cloud fan-out (SNS publishes, multiple SQS queues subscribe, each drained by its own worker fleet).

4. Kafka internals (the depth interviewers probe)

Topics, partitions, offsets

A **topic** is a named stream. It's split into **partitions** — that's the unit of parallelism, ordering, and storage. Each partition is an ordered, immutable, append-only log of records, each with a monotonically increasing **offset** (0, 1, 2, ...). A partition lives entirely on one broker's disk (plus replicas).

Topic "orders" (3 partitions)

```
P0: [0][1][2][3][4]→ (leader on broker-1, replicas on broker-2,3)
P1: [0][1][2]→ (leader on broker-2)
P2: [0][1][2][3]→ (leader on broker-3)
```

Producer hashes key → picks partition. Order guaranteed **WITHIN** a partition only.

The throughput ceiling is the partition count. A topic with 6 partitions can be drained by at most 6 active consumers in a group — partition #7's consumer sits idle. Pick partition count for your *peak future* parallelism; you can increase but never decrease, and increasing breaks key → partition stability (see §5).

Consumer groups, rebalancing, offset commits

Each consumer group gets each partition assigned to **exactly one** member. The group coordinator (a broker) runs the **rebalance protocol** when membership changes. Classic ("eager") rebalancing is *stop-the-world*: everyone revokes all partitions, then reassigns — pauses can be seconds. **Cooperative incremental rebalancing** (KIP-429, default since Kafka 2.4+) only moves the partitions that actually need to move. Use **static membership** (`group.instance.id`) so a pod restart inside `session.timeout.ms` doesn't trigger a rebalance at all.

Consumers **commit offsets** (to the internal `__consumer_offsets` topic) to record progress. The semantics hinge on *when* you commit relative to *when* you process:

```
At-least-once: process(msg) → then commit(offset)
                crash between = reprocess on restart = DUPLICATE    (safe default)
At-most-once:  commit(offset) → then process(msg)
                crash between = message LOST                        (rarely acceptable)
```

`enable.auto.commit=true` commits on a timer regardless of processing — a quiet source of *both* loss and duplicates. For correctness, **disable auto-commit and commit explicitly after processing**.

Replication: ISR, acks, and the durability dial

Each partition has one leader and N-1 followers. The **In-Sync Replica (ISR)** set is the followers caught up within `replica.lag.time.max.ms`. Durability is a producer/broker negotiation:

acks	Guarantee	Failure cost	Throughput
0	Fire-and-forget	Loses data on any hiccup	Highest
1	Leader persisted	Loses data if leader dies before replication	Medium
all (-1)	All ISR persisted	No loss while ≥1 ISR survives	Lowest (still fast)

`acks=all` alone isn't enough: if ISR shrinks to just the leader, "all" means "the one". Pair it with `min.insync.replicas=2` (on a replication-factor-3 topic) so a write fails fast rather than silently risking loss. And set `unclean.leader.election.enable=false` — otherwise a stale out-of-sync replica can be elected leader and *silently truncate committed data* (the famous availability-vs-durability knob; see consensus trade-offs in `10-replication-and-consensus.md`).

Retention vs log compaction

Two eviction policies, often confused:

- **Retention** (`cleanup.policy=delete`): drop segments older than `retention.ms` (default 7 days) or beyond `retention.bytes`. Time-series / events.
- **Compaction** (`cleanup.policy=compact`): keep the *latest value per key* forever, garbage-collecting superseded versions. Turns the log into a durable, replayable **changelog / snapshot** — perfect for "current state of every order keyed by order_id". Kafka Streams' state stores and `__consumer_offsets` rely on it. A `null` value is a **tombstone** that deletes the key after `delete.retention.ms`.

```
Compaction of key=user42:
before: [u42=A][u7=X][u42=B][u42=C][u7=Y]
after:  [u7=Y][u42=C]    ← only the newest per key survives
```

5. Ordering guarantees and partition-key choice

Kafka guarantees order only within a partition. Across partitions, all bets are off. So the partition key is the most consequential design decision you'll make. Records with the same key always land on the same partition (key hash mod partition count), so they're ordered relative to each other.

- **Order matters per entity** (all events for `order_123` must apply in sequence): key by `order_id`. Now status transitions never race. The cost: an order with a hot stream of events can hot-spot one partition.
- **Order doesn't matter / you want max parallelism:** key by something high-cardinality and uniform (e.g. a random/round-robin key), or null key (round-robin batches).

The **partitioning skew trap**: keying by `tenant_id` in a multi-tenant SaaS (like this codebase) means one whale tenant saturates a single partition while others idle — a classic hot-partition incident. Mitigation: composite key (`tenant_id:order_id`) when per-order ordering is enough, or salt the key.

Subtle producer gotcha: even single-partition order can break. With `retries>0`, a failed batch retried after a later batch succeeded reorders records — unless `max.in.flight.requests.per.connection ≤ 5` **with the idempotent producer enabled** (Kafka then preserves order on retry via sequence numbers). With the idempotent producer *off*, you need `max.in.flight=1` for strict order, which kills pipelining throughput.

6. Delivery semantics: at-most / at-least / exactly-once

	duplicates?	loss?	use when
at-most-once	no	possible	metrics you can lose, max throughput
at-least-once	POSSIBLE	no	default; pair with idempotent consumers
exactly-once	no	no	money, dedup-expensive side effects

The honest senior take: **in a distributed system you get at-least-once delivery, and you engineer *effectively-once processing* by making consumers idempotent.** The Two Generals problem means the producer can never *know* its message arrived without an ack, and the ack itself can be lost — so it must retry, so duplicates are inevitable on the wire. See `11-idempotency-and-distributed-transactions.md`.

Idempotent consumers (the real-world EOS)

Make processing safe to repeat: dedup on a business key (`event_id` / `order_id`) stored in a `processed_events` table with a unique constraint, or design naturally idempotent operations (`SET status='paid'` is idempotent; `balance = balance + 10` is *not*). This is the technique you reach for 95% of the time.

Kafka exactly-once (EOS) — what it actually covers

Kafka offers genuine exactly-once *within Kafka* via two mechanisms:

1. **Idempotent producer** (`enable.idempotence=true`, default in modern clients): the broker assigns each producer a PID and tracks per-partition sequence numbers, so a retried duplicate batch is

dropped broker-side. Kills producer-retry duplicates — but only within a single producer session, single partition.

2. **Transactions** (`transactional.id` + `initTransaction` / `beginTransaction` / `sendOffsetsToTransaction` / `commitTransaction`): atomically write to multiple partitions **and** commit consumer offsets in one transaction. Consumers set `isolation.level=read_committed` to skip aborted records. This makes the **consume** → **process** → **produce** loop atomic — the foundation of Kafka Streams EOS (`processing.guarantee=exactly_once_v2`).

The crucial caveat: Kafka EOS is exactly-once *for the Kafka → Kafka path*. The moment your consumer does a side effect outside Kafka — write to Postgres, call Stripe, send an email — Kafka transactions don't cover it. For that boundary you still need the **transactional outbox** + idempotency keys (covered in module 11). Don't let "Kafka has exactly-once" lull you into skipping consumer idempotency on external effects.

7. Backpressure, retries, DLQs, and poison messages

Backpressure

Producers can outrun consumers. What happens to the backlog?

- **Log (Kafka):** the backlog just sits on disk (cheap — TBs are fine). Backpressure surfaces as **consumer lag** (offset distance behind the log head). Alert on lag *trend*, not absolute value: rising lag = consumers losing the race. Kafka has effectively no producer-side backpressure beyond buffer-full blocking (`buffer.memory` , `max.block.ms`).
- **Queue (RabbitMQ):** unbounded queues swell RAM until the broker hits its memory high-watermark and **blocks publishers** — backpressure propagates to producers. Use bounded queues / `x-max-length` with an overflow policy (drop-head or reject-publish), and consumer **prefetch** (`basic.qos`) so one greedy consumer doesn't hoard 10k unacked messages.
- **Pull vs push:** Kafka consumers *pull* (natural flow control — fetch when ready). RabbitMQ *pushes* (must bound with prefetch or you flood the consumer). Pull-based is generally more robust under backpressure.

The cascade failure: an unbounded queue in front of a slow DB hides the problem until the queue's storage fills, then *everything* upstream blocks at once — a thundering collapse. Bounded buffers + load shedding fail *gracefully* instead. "An unbounded queue is a memory leak with extra steps."

Retry with exponential backoff + jitter

A poison/transient failure shouldn't hammer a struggling downstream. Back off exponentially, and **always add jitter** to avoid the retry **thundering herd** (10k consumers all retrying at exactly t+1s, t+2s ... synchronize into traffic spikes that re-kill the service).

```

delay = min(cap, base * 2^attempt)      # exponential
delay = random_between(0, delay)        # FULL JITTER (AWS-recommended)
# base=100ms, cap=20s, attempt 0..6 → ~100ms, 200, 400, 800, 1.6s, 3.2s, 6.4s (then randomized)

```

AWS’s “Exponential Backoff and Jitter” article showed **full jitter** minimizes both contention and completion time vs naive backoff. Cap total attempts; infinite retry of a permanent error is a tight loop in disguise.

Dead-letter queues & poison messages

A **poison message** can never succeed — malformed payload, a referenced row that was deleted, a bug. Retrying it forever blocks the partition/queue (head-of-line blocking) and burns CPU. After N attempts, **route it to a Dead-Letter Queue (DLQ)** and move on.

```

main queue —process—> fail
                        |
                        v attempt < N? —yes—> retry (backoff queue)
                        |
retry?                  |
| attempt ≥ N           |
|                       |
|—————> [ DLQ ] (alert + human/automated triage)

```

- **SQS**: native — set a redrive policy with `maxReceiveCount` ; after N receives the message auto-moves to the DLQ.
- **RabbitMQ**: `x-dead-letter-exchange` ; messages dead-lettered on reject/nack/TTL-expiry. Delayed-retry tiers via per-queue TTL + DLX chaining.
- **Kafka**: no native DLQ — *you* publish failed records to a `topic.DLT` (Spring Kafka / Kafka Connect do this for you). Critically, **never block the partition** on a poison record; commit past it and side-line it, or you stall every key behind it.

DLQ discipline: a DLQ with no alarm and no owner is a black hole where data goes to die silently. Monitor DLQ depth, capture the failure reason + stack on the message, and build a *redrive* path (fix bug → replay DLQ back to main). For Kafka, replayability means you can also just re-seek the offset.

8. Throughput tuning (concrete knobs & numbers)

Kafka’s speed comes from **sequential disk I/O**, the OS page cache, **zero-copy** (`sendfile`), and **batching**. The producer-side levers:

Knob	Effect	Typical
<code>batch.size</code>	Bytes per partition batch	64KB–256KB (up from 16KB default)
<code>linger.ms</code>	Wait to fill a batch	5–50ms (trade latency for throughput)
<code>compression.type</code>	Shrink payload	<code>lz4</code> / <code>zstd</code> (zstd best ratio)
<code>buffer.memory</code>	Producer send buffer	bump for bursty load
consumer <code>fetch.min.bytes</code>	Wait for N bytes before returning	raise to batch fetches

Back-of-envelope: 1KB messages, 100MB/s/broker sustained sequential write → ~100k msg/s/broker, and a partition tops out around 10–50MB/s. Need 1M msg/s? ~10–20 partitions across enough brokers, batched and lz4-compressed. **More partitions ≠ free**: each adds open file handles, replication overhead, longer rebalances, and higher end-to-end latency; thousands of partitions per broker degrade controller failover. Size partitions for required parallelism plus headroom, not “as many as possible.”

The dominant real bottleneck is rarely the broker — it’s the **downstream the consumer writes to** (Postgres, an API). Tune consumer concurrency and downstream batching together; a faster broker just fills your DB’s queue sooner.

9. Common pitfalls / war stories

- **Auto-commit ate my messages.** `enable.auto.commit=true` committed offsets on its 5s timer mid-batch; a crash skipped the un-processed tail. Fix: manual commit *after* processing.
- **The hot partition.** Keyed events by `tenant_id`; the biggest customer’s traffic pinned one consumer at 100% while five sat idle. Lag climbed only for that tenant. Fix: composite/salted key.
- **Reordered writes despite one partition.** Idempotent producer was off and `max.in.flight=5`; a retried batch overtook a later one, applying an old status after a new one. Fix: `enable.idempotence=true`.
- **The retry storm.** A flaky payment API blipped; 50k consumers retried in lockstep (no jitter) and turned a 2s blip into a 20-minute self-DDOS. Fix: full jitter + circuit breaker.
- **The silent DLQ.** Poison messages piled into a DLQ nobody watched; finance noticed missing orders a week later. Fix: alarm on DLQ depth, store failure reason, build redrive.
- **Treating Kafka like a database.** Querying by scanning a topic, or keeping infinite retention as a “source of truth” without compaction or an external store. Kafka is a *log*, not a queryable DB.
- **“We have exactly-once.”** Enabled Kafka transactions, then a consumer also charged Stripe — double charges on rebalance reprocessing because the external call wasn’t idempotent. Kafka EOS doesn’t cross the Kafka boundary.
- **The dual-write.** Service writes to Postgres *and* publishes to Kafka in two steps; a crash between them leaves DB and stream inconsistent. Fix: the **transactional outbox** + CDC/Debezium (see module 11).
- **Rebalance storms.** Aggressive `session.timeout.ms` + GC pauses caused constant rebalances; throughput collapsed under stop-the-world reassignment. Fix: cooperative rebalancing + static membership + tuned timeouts.

Case Study: LinkedIn’s creation of Apache Kafka (the central log)

The problem. By 2010 LinkedIn had ~90 million members and a data-integration mess that grew worse every quarter. Dozens of systems needed to know about the same events — a member viewing a profile, sending a connection request, a page impression, a search. The activity-tracking and operational-metrics

pipelines fed the newsfeed, relevance/search, ad targeting, fraud detection, the data warehouse, and a fleet of Hadoop jobs. Each producer → consumer link was a bespoke point-to-point integration. With N sources and M sinks you tend toward an $O(N \times M)$ spaghetti of pipelines, each with its own format, batching cadence, and failure behavior. Their existing tools didn't fit: traditional message *queues* (ActiveMQ) buckled under the volume and, being destructive-read, couldn't feed both a real-time consumer *and* a batch Hadoop loader from the same stream; log-aggregation tools (Scribe/Flume) were one-directional toward HDFS and weren't built for low-latency online consumers. Volumes were already enormous — hundreds of billions of messages/day, and within a few years over a trillion/day and petabytes flowing through the cluster.

The insight: make the *log* the primitive (not the queue). This is exactly the **log vs queue** distinction from §2. Instead of a smart broker that deletes on ack, LinkedIn built a dumb broker holding an **append-only, replayable, non-destructive log**, with *smart consumers* tracking their own **offsets**. The same event stream could now serve a sub-second newsfeed consumer and a once-an-hour Hadoop batch loader off identical data, each at its own position. The $O(N \times M)$ web collapsed into $O(N + M)$: every producer writes once to Kafka; every consumer reads independently. That is the **decoupling of producers from consumers** the module opens with — neither knows the other exists.

BEFORE: $O(N \times M)$ point-to-point pipelines

```
tracking → search
db-CDC   → newsfeed
metrics  → Hadoop
ads      → fraud ...every pair wired
```

AFTER: $O(N + M)$ via the central log

```
tracking ↘
db-CDC   ↘      ↗ search (group, offset)
metrics  → KAFKA → newsfeed (group, offset)
ads      ↘      ↗ LOG   → Hadoop (group, offset)
                               ↘ fraud (replay any time)
```

Partitions, consumer groups, offsets — the scaling levers. A topic like `PageViewEvent` was split into **partitions** (§4), the unit of parallelism *and* of ordering. Each **consumer group** (newsfeed, search-indexer, Hadoop ETL) gets every partition assigned to exactly one member, so members of one group *compete* (point-to-point parallelism) while different groups each see the whole stream (pub/sub) — the **consumer-group duality** from §3. Throughput scales by adding partitions and consumers; LinkedIn ran topics with hundreds of partitions to drain a single high-volume stream across many machines.

Ordering per key — chosen deliberately, not globally. LinkedIn did *not* want a global total order (it doesn't scale — one partition, one disk). They accepted **ordering only within a partition** (§5) and chose partition keys to match the entity that needed ordering: events keyed by member or by entity id land on the same partition and stay in sequence, while unrelated events spread across partitions for parallelism. This is the precise trade-off the module hammers: per-key order *or* unbounded parallelism, picked per stream.

Throughput from boring mechanics. Kafka's numbers came from the levers in §8 — **sequential disk I/O**, the OS **page cache**, **zero-copy** (`sendfile`) straight from page cache to socket, and aggressive producer **batching** with compression. LinkedIn's published 2014 benchmark pushed **~2 million writes/sec on three cheap commodity brokers**; their production cluster carried over a trillion messages/day. Crucially they leaned on retention being *cheap on disk* — keep days of data and let consumer **lag** (not a swelling queue) be the backpressure signal (§7).

Replay was the killer feature. Because reads are non-destructive, a consumer that shipped a bug could fix it and **re-seek to an old offset** to reprocess history — the **replay** capability from §2/§7. A new derived

dataset (say, a fresh search index) could be built by replaying the whole topic from offset 0. This same property turned the log into the backbone of **event sourcing and stream processing** (later Samza, then Kafka Streams) and of **CDC via Debezium** feeding the outbox pattern (module 11).

The trade-off they accepted. Kafka gave up the conveniences of a queue: **no per-message acknowledgment, priority, TTL, or selective delete**, and (originally) only **at-least-once** delivery — duplicates were the consumer's problem to dedup (the *effectively-once via idempotent consumers* stance of §6; transactions/EOS came years later in KIP-98). They also gave up *global* ordering and strong queryability — Kafka is a log, not a database (§9). In exchange they got linear horizontal throughput, replayability, and one stream feeding many independent consumers. For a data-integration backbone, that bargain was overwhelmingly correct: Kafka was open-sourced (2011), graduated from Apache incubation, and became the de-facto streaming standard, with Confluent spun out to commercialize it.

Lessons

- **Pick the log when many parties care about the same facts and may re-read them; pick the queue when it's one-unit-of-work-once.** LinkedIn's whole win was recognizing their events were *facts*, not *commands* — so a replayable, non-destructive log beat a destructive queue.
- **A shared log turns $O(N \times M)$ integration spaghetti into $O(N+M)$.** Standardize on one event backbone with self-tracking consumers instead of wiring every source to every sink.
- **Partition key = your ordering-vs-parallelism decision.** Order per entity costs you a hot-partition risk; global order costs you scale. Choose per stream, not platform-wide.
- **Cheap sequential disk + replay changes what's possible.** Treat the log as durable, re-readable history — it enables reprocessing, new derived views, and CDC — but don't mistake it for a queryable database or assume duplicates won't happen.

10. Test yourself

1. **You need per-order event ordering AND high throughput across millions of orders. What's the partition key, and what breaks it?** Hint: key by `order_id` for per-entity order; producer `max.in.flight` + idempotence can still reorder on retry; increasing partition count later breaks key → partition stability.
2. **Why is “exactly-once delivery” impossible, and what do we actually build instead?** Hint: Two Generals / lost-ack; at-least-once + idempotent (effectively-once) processing.
3. **`acks=all` is set but you still lost data. How?** Hint: ISR shrank to the leader and `min.insync.replicas` wasn't set, or unclear leader election truncated committed records.
4. **When do you choose RabbitMQ/SQS over Kafka?** Hint: discrete task distribution, per-message priority/TTL/delay, no replay needed, modest throughput; vs replayable event stream with many independent subscribers.
5. **A poison message is stuck at the head of a Kafka partition. What happens and what do you do?** Hint: head-of-line blocking stalls every key behind it; commit past it / route to a DLT after N attempts, alert + redrive.

6. **Why add jitter to exponential backoff?** *Hint: synchronized retries form a thundering herd that re-kills the recovering service; full jitter spreads load.*
 7. **Difference between log compaction and retention, and when do you want compaction?** *Hint: retention drops old; compaction keeps latest-per-key → durable changelog / current-state snapshot; tombstones delete keys.*
 8. **Your consumer does exactly-once into Kafka but also writes Postgres. Is it exactly-once end-to-end?** *Hint: no — Kafka transactions don't cover external side effects; need outbox + idempotency keys at that boundary.*
-

11. Further reading

- **DDIA** (Kleppmann), *Designing Data-Intensive Applications* — Ch. 11 “Stream Processing” (logs vs queues, exactly-once, fault tolerance) and Ch. 4 (message schemas). The canonical treatment.
- **Apache Kafka docs** — Design, Replication, Exactly-Once Semantics (KIP-98), Idempotent Producer, Log Compaction. <https://kafka.apache.org/documentation/>
- **Kreps, “The Log: What every software engineer should know about real-time data’s unifying abstraction”** — the foundational essay on the log abstraction.
- **AWS Architecture Blog, “Exponential Backoff and Jitter”** — the definitive empirical case for full jitter. <https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/>
- **KIP-98** (Kafka transactions / EOS) and **KIP-429** (cooperative incremental rebalancing) — primary-source design rationale.
- **AWS SQS Developer Guide** — dead-letter queues, redrive policy, FIFO vs standard queues.
- **RabbitMQ docs** — Consumer Prefetch, Dead Letter Exchanges, Quorum Queues, Flow Control.
- **Confluent, “Transactions in Apache Kafka”** and **“Designing Event-Driven Systems”** (Stopford, free O’Reilly e-book).
- Sibling modules: `09-cap-pacelc-and-consistency-models.md` , `10-replication-and-consensus.md` , `11-idempotency-and-distributed-transactions.md` .

13 — Event Sourcing & CQRS

Source: [readme/research/system-design/13-event-sourcing-and-cqrs.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains two related ideas for storing data in software: “Event Sourcing” (saving a history of everything that happened instead of just the current state) and “CQRS” (using a separate system for reading data vs. writing data). These ideas solve real problems in finance, order tracking, and audit trails — but they also add a lot of complexity, so the document is honest about when NOT to use them.

The main parts explained simply:

- **The core idea (ledger vs. balance)** — A normal app saves only the current number (e.g. your account balance). Event Sourcing instead saves every transaction that ever happened — like a bank ledger. The current balance is calculated from the history. This means you can always answer “why is this number what it is?” and travel back in time to any past state.
- **Event Store** — A special database that only lets you add new records, never change or delete old ones. Every action is saved as a past-tense fact (e.g. “Order Shipped”, “Money Withdrawn”). Each item (like one customer account) gets its own list of events.
- **Snapshots** — If an item has millions of events, replaying all of them every time is slow. A snapshot saves a “checkpoint” of the state at a point in time, so you only replay the recent events after that checkpoint.
- **CQRS (Command Query Responsibility Segregation)** — A pattern where you use one system to save/change data (the “write side”) and separate, optimized systems to read and display data (the “read side”). This lets you shape the read data differently for search, dashboards, reports, etc.
- **Eventual consistency** — Because the read side catches up to the write side with a small delay, a user might save something and not see it immediately in a list. The document covers ways to handle this gracefully in the UI.
- **Rebuilding read models** — Because read data is derived from events, you can always throw it away and rebuild it from scratch. This is powerful but risky — any code that sends emails or notifications must not run again during a rebuild, or customers get duplicate messages.
- **Event versioning** — Events saved in 2019 must still be readable in 2029. The document explains how to handle changes to event formats over time without breaking old records.
- **When NOT to use this** — Most apps should not use Event Sourcing. It adds major complexity. Use it only where a full history is essential (financial ledgers, order lifecycles). Do not use it for simple things like product catalogs or settings pages.
- **Common mistakes** — Sending emails inside the read-model code (causes duplicate emails on rebuild), making events that just copy the whole record (wastes the benefit), and using it everywhere instead of only where history truly matters.

What to do with this: Only apply Event Sourcing to the 1–2 parts of your system where the full history is the product itself (like an order history or invoice ledger). Leave everything else as simple database rows. If you use it, plan for how you will handle old event formats from day one, and never put email-sending or other side effects inside your read-model builders.

What you'll learn: How to model state as an immutable log of *facts that happened* instead of a mutable row you overwrite, why splitting your write model from your read model (CQRS) is a natural consequence, and — most importantly — the brutal operational costs these patterns impose so you can tell when they pay for themselves and when they're résumé-driven over-engineering.

Prerequisites: Read `06-replication-and-consistency.md` (eventual consistency, log shipping), `09-cap-pacelc-and-consistency-models.md` (consistency models), and `12-message-queues-and-the-outbox-pattern.md` (the outbox pattern, exactly-once delivery) first. Event sourcing leans hard on all three.

1. Intuition: your bank statement vs your account balance

A normal CRUD app stores the *current* balance: a row `accounts(id, balance)`. When you withdraw \$20, the app does `UPDATE accounts SET balance = balance - 20`. The old value is gone forever. The number “180” tells you *what* but never *why* or *when*.

A bank doesn't actually work that way. The source of truth is the **ledger** — an append-only list of transactions:

```
+5000 salary      2026-01-01
-200  rent        2026-01-03
-20   atm withdrawal 2026-01-05
```

The balance `4780` is **derived** by folding (summing) the ledger. The ledger is the truth; the balance is a *cached projection* of it. You can never “edit” a past transaction — to fix a mistake you append a *correcting* transaction (a reversal). This is **event sourcing**: store the events (the facts), derive the state.

The defining inversion: CRUD stores state and throws away the transitions. Event sourcing stores the transitions and derives the state. Everything else follows from that one decision.

2. The event store

An **event** is an immutable, past-tense fact: `MoneyWithdrawn`, `OrderShipped`, `ProductPriceChanged`. Not a command (`WithdrawMoney` — an *intent* that may be rejected), not a row mutation. Events have already happened; they cannot fail or be rejected.

The **event store** is an append-only database of events, partitioned into **streams** — usually one stream per aggregate instance (e.g. one stream per bank account, `account-7f3a...`).

```
Stream: account-7f3a (one stream per aggregate)
+-----+-----+-----+
| # | event type | payload | version |
+-----+-----+-----+
| 1 | AccountOpened | {owner: "Jo", ccy:USD} | 1 |
| 2 | MoneyDeposited | {amount: 5000} | 2 |
| 3 | MoneyWithdrawn | {amount: 200} | 3 |
| 4 | MoneyWithdrawn | {amount: 20} | 4 |
+-----+-----+-----+
^ append-only, never UPDATE/DELETE
```

Core operations (this is the *entire* write-side API):

- `append(streamId, events, expectedVersion)` — append with optimistic concurrency.
- `readStream(streamId)` — read all events for one aggregate, in order, to rebuild it.
- `readAll(fromPosition)` — read the global ordered log to build projections.

Optimistic concurrency is non-negotiable. Two requests both load account version 4, both decide a withdrawal is valid, both try to append version 5. The store enforces “append only if current version == expectedVersion (4)”; the second `append` fails with a concurrency conflict and the command is retried against fresh state. This is how you preserve the aggregate’s invariants without locking. It’s a compare-and-swap on the stream tail.

Real implementations: `EventStoreDB` / `Kurrent` is purpose-built for this. Most teams instead use **Postgres** (`events` table, `(stream_id, version)` unique constraint = optimistic concurrency for free), **DynamoDB** (conditional writes on a sort key), or **Kafka** (one topic-partition per aggregate type, log-compacted — but Kafka makes single-aggregate reads and concurrency checks awkward, so it’s better as the *projection transport* than the system of record). Marten (Postgres + .NET) and Axon (JVM) are popular frameworks.

Rebuilding state (the “fold”)

To get the current state of an aggregate, you read its stream and replay events through a pure function:

```
state = events.reduce(apply, emptyState)

// apply is a pure (state, event) -> state function:
apply(s, AccountOpened e) => { ...s, balance: 0, owner: e.owner }
apply(s, MoneyDeposited e) => { ...s, balance: s.balance + e.amount }
apply(s, MoneyWithdrawn e) => { ...s, balance: s.balance - e.amount }
```

This is a left fold — functional programmers will recognize `foldl`. The `apply` functions **must never reject** an event or run validation; the event already happened. Validation lives in the *command handler* (see §5), not in `apply`.

Snapshots: the replay-cost problem

Replaying 4 events is free. Replaying an aggregate with **500,000 events** (a hot account, a long-lived shopping cart, a 10-year-old loan) on every command is not — you’d read half a million rows to decide one withdrawal.

A **snapshot** is a serialized copy of the folded state at a known version, written periodically (e.g. every 100 events):

```
load(streamId):
  snap = latestSnapshot(streamId)           // state @ version 400
  tail = readStream(streamId, from=snap.v) // events 401..427 only
  return tail.reduce(apply, snap.state)
```

Back-of-envelope: at ~0.1 ms per event applied in memory, replaying 500k events ≈ **50 ms of pure CPU** *plus* the cost of reading 500k rows from disk/network (easily hundreds of ms). With a snapshot every 100 events, you replay ≤100 events ≈ **10 ms + a single snapshot read**. Snapshots are a *cache*, not truth — you must be able to delete every snapshot and rebuild from events alone. Never let a snapshot become load-bearing.

3. CQRS: why reads and writes must diverge

CQRS = Command Query Responsibility Segregation. Separate the model you write through (commands) from the model(s) you read from (queries). It's older and simpler than event sourcing (Greg Young / Bertrand Meyer's CQS scaled up) and does *not* require event sourcing — but ES forces it, because a stream of events is a terrible thing to query.

Try answering “show me all accounts with balance > \$10,000, sorted by balance” from an event store. You'd have to replay *every* stream. Impossible at scale. So you build **read models** (a.k.a. **projections** / materialized views): denormalized, query-optimized tables built by consuming the event log.



Key consequences:

- **One write model, many read models.** The same events feed a SQL table for transactional reads, an Elasticsearch index for search, a Redis hash for a dashboard, a data-warehouse table for analytics. Each is shaped for *its* query. No more compromising one schema to serve five access patterns.
- **The link between sides is asynchronous and eventually consistent.** A projection lags the write by the time it takes to process the event — usually milliseconds, but it *can* fall behind (see §6). This lag is the single biggest source of bugs and UX surprises.

- **Read models are disposable.** They're derived data. If a projection is corrupt or you need a new shape, you delete it and **rebuild from the event log** (see §7).

CQRS without event sourcing

You can do CQRS with a plain CRUD write DB that emits change events (via the **outbox pattern** or CDC) into read models. This is the pragmatic 80% case. Conflating “CQRS” with “event sourcing” is the most common conceptual mistake — keep them separate in your head.

Pattern	Write store	Read store	Complexity	Gives you
Plain CRUD	Mutable rows	Same rows	Lowest	Simplicity
CQRS only	Mutable rows	Separate projections	Medium	Read scaling, polyglot reads
Event Sourcing only	Event log	Replay aggregate	High	Audit, time-travel, no read scaling
ES + CQRS	Event log	Many projections	Highest	All of the above; the “full” pattern

4. When it helps — and when it's overkill

Be honest: **most apps should not do event sourcing.** It is a specialist tool with a large, permanent tax.

Use ES+CQRS when...	Avoid it when...
Audit trail is a <i>hard</i> requirement (finance, healthcare, legal) — “why is this number what it is?” must be answerable	A CRUD audit-log table would satisfy compliance
The business <i>thinks in events</i> (ledgers, order lifecycles, insurance claims, inventory movements)	The domain is simple CRUD (a blog, a settings page, a CMS)
You need temporal queries: “what was state on date X?”, replay, what-if analysis	You only ever need “current state”
Multiple read shapes from one write (search + dashboard + reporting)	One read shape suffices
Read and write load differ by orders of magnitude	Read and write load are comparable and low
Collaboration/merge semantics matter (you can derive intent from events)	Last-write-wins is fine

The honest cost list (war-tested):

- Eventual consistency leaks into the UI — “I saved it, why don't I see it?”
- Event versioning is forever — you can never delete an old event shape; you must keep code that reads v1 events written in 2019.
- Debugging requires replaying streams, not reading a row.
- No **UPDATE / DELETE** means GDPR “right to be forgotten” fights an append-only log (see pitfalls).
- Onboarding cost: every new engineer must learn the pattern before touching anything.

- Tooling is thinner; you'll build infrastructure CRUD gives you for free.

Rule of thumb: **event-source the 1–2 aggregates where history is genuinely the product** (the ledger, the order) and leave the rest as boring CRUD. ES is an aggregate-level decision, not an app-level one. For a print-SaaS like this codebase, an order/job lifecycle or an invoice ledger might justify it; the product catalog and CMS absolutely do not.

5. Aggregates & consistency boundaries (DDD)

This is the part interviews probe and incidents expose. An **aggregate** (Eric Evans, *Domain-Driven Design*) is a cluster of objects treated as one unit for data changes, with one **aggregate root** as the entry point. Critically:

An aggregate is your transactional consistency boundary. Everything inside is consistent *immediately* (one stream, one optimistic-concurrency check). Everything *between* aggregates is consistent only *eventually*.

So aggregate design = drawing your strong-vs-eventual consistency lines. Get this wrong and you either (a) make aggregates huge to force consistency, killing concurrency, or (b) put a true invariant across two aggregates and lose it.

Worked example. Invariant: “an account may not be overdrawn.” That invariant is *inside* one account → the account is the aggregate, the rule is enforced synchronously in its command handler on load+append. Now: “transfer \$100 from account A to account B.” That invariant (“don’t lose money”) spans two aggregates → you **cannot** do it in one transaction. You use a **process manager / saga**:

`MoneyWithdrawn(A)` → triggers `DepositMoney(B)` → emits `MoneyDeposited(B)` ; if B's deposit fails, emit a compensating `MoneyRefunded(A)` . (See `12-message-queues-and-the-outbox-pattern.md` for saga/compensation mechanics.)

Command flow (one aggregate):

```
Command --> load aggregate (replay/snapshot) --> decide (validate invariants)
--> produce events --> append(expectedVersion) --> [conflict? reload & retry]
```

Design heuristics: keep aggregates *small* (favor concurrency); reference other aggregates by ID, never by embedding; one command mutates exactly one aggregate; if you “need” a transaction across two aggregates, your boundaries are probably wrong *or* you actually need a saga and eventual consistency.

6. Eventual consistency between write and read

The user submits a command, the API returns `200` , the user navigates to the list view — and their change isn't there, because the projection hasn't caught up. This is the read-your-own-writes problem from `06-replication-and-consistency.md` , now structural.

Mitigations, weakest to strongest:

1. **Return the new state from the write side.** The command handler already has the post-command aggregate state; return it so the UI updates optimistically without waiting for the projection.
2. **Version/correlation tokens.** Write returns the new global position N; the read API can `wait-for(N)` (block until the projection has processed up to N) or the client polls until the projection version $\geq N$.
3. **Read from the write model for the just-acted-on aggregate,** projections for everything else. (You can always load one aggregate by replay.)
4. **Accept and surface the lag** — “Saving...”, then “Saved”. Honest UX beats a lie.

Numbers to internalize: a healthy projection lags **single-digit to low-tens of milliseconds**. Under a backlog (deploy, hot partition, slow downstream) it can hit **seconds to minutes**. *Always* monitor projection lag (current global position – projection’s processed position) and alarm on it; an unmonitored projection that silently falls 6 hours behind is a classic 3 a.m. page.

7. The read-model rebuild problem

Projections are derived, so you *will* rebuild them: a bug shipped wrong data, you need a new read shape, or you migrate stores. Rebuild = drop the projection, replay the entire event log through the projection handler from position 0.

The problem is **scale and side effects**:

- A 2-billion-event log at 50k events/sec replay = **~11 hours**. During that window the read model is stale or absent.
- **Replays must be side-effect-free.** If your projection handler sends an email on `OrderShipped`, a rebuild re-sends ten thousand “your order shipped!” emails to customers from 2021. *This is a real outage class*. Side effects belong in process managers/reactors that track “already handled”, never in idempotent projections.
- **Idempotency:** rebuilds and at-least-once delivery mean a projection may see an event twice. Project with upserts keyed by event/aggregate identity, never blind `INSERT / +=` without dedup.

Strategies to make rebuilds survivable:

Strategy	How	Trade-off
Blue/green projection	Build new version in a parallel table, atomically swap when caught up	Double storage; cleanest zero-downtime
Catch-up subscription	Replay from 0 to “now”, then switch to live tail	Standard; needs a defined cutover point
Snapshot the projection	Periodic checkpoints so partial rebuilds resume	Adds projection-side snapshot machinery
Partition the replay	Parallelize by stream/partition key	Faster; only if per-partition ordering suffices

Kafka users get this semantics for free-ish: reset a consumer group's offset to 0 and reprocess (the log-compaction + offset model is literally designed for replay).

8. Event versioning & upcasting

Events are immutable *and immortal*. The `OrderPlaced` event you wrote in 2019 will still be in the store in 2029, and your 2029 code must still read it. You cannot run a migration to “fix old events” without rewriting history (which defeats the point and breaks hash chains/audits). So you need a deliberate **schema evolution** strategy.

Technique	What it does	When
Tolerant reader	New code ignores unknown fields, defaults missing ones	Most additive changes
Weak schema / additive only	Only ever <i>add</i> optional fields; never rename/remove/retype	Default discipline — follow it religiously
Upcasting	On read, transform old event shape → newest shape before <code>apply</code> sees it	Structural changes (rename, split, type change)
New event type	Stop emitting <code>v1</code> , emit <code>OrderPlacedv2</code> ; keep an upcaster <code>v1→v2</code>	Semantic change too big to upcast cleanly
Copy-and-transform stream	Write a brand-new stream from old events via a one-off migration	Last resort; breaks immutability/audit guarantees

Upcasting is the workhorse. An upcaster is a pure function `oldEvent -> newEvent` applied as events are deserialized, *before* they reach `apply`. Chain them (`v1→v2→v3`) so `apply` only ever knows the latest shape:

```
raw bytes --> deserialize --> [upcast v1->v2] --> [upcast v2->v3] --> apply()
```

This mirrors the **forward/backward compatibility** discipline from `12-message-queues-and-the-outbox-pattern.md` and the schema-registry world (Avro/Protobuf). Same rules: never remove or repurpose a field; adding optional fields is safe; renaming requires an upcaster or a new type. Store an explicit `event_version` in every event from day one — retrofitting it later is misery.

9. Relationship to the outbox pattern

The hardest correctness problem in ES+CQRS is the **dual-write**: you must (a) persist the event as truth *and* (b) publish it so projections/sagas react — atomically. Persist-then-publish risks losing the publish if you crash between them; publish-then-persist risks publishing an event you never durably stored.

The **outbox pattern** solves this (full mechanics in [12-message-queues-and-the-outbox-pattern.md](#)): write the event and an outbox row in **one local transaction**, then a relay/CDC process publishes the outbox to the broker. In event sourcing this is especially clean because **the event store itself is the outbox** — projections are just catch-up subscriptions reading the same ordered log the writes go into. There's no separate publish step to lose; consumers track their own position in the log.

```
+-----+      same log      +-----+
| append event |----->| projection reads |
| (atomic, truth) | (no dual write) | from log position |
+-----+      +-----+
```

When you do CQRS *without* ES (CRUD write DB), you bolt the outbox on: state row + outbox event in one transaction, relay to broker, projections consume. Either way the invariant is the same — **never do two independent writes and pray**. (See [12](#) for at-least-once + idempotent consumers; ES inherits all of it.)

10. Real systems

- **Banking/ledgers:** the canonical fit. [TigerBeetle](#) is a purpose-built double-entry accounting database that is event-sourced/log-structured to the core. Traditional core-banking and brokerage systems are ledgers.
- **EventStoreDB / Kurrent:** the reference event-store database — streams, optimistic concurrency, catch-up subscriptions, projections.
- **Kafka:** “turning the database inside out” (Martin Kleppmann) — the log as the source of truth; Kafka Streams/ksqlDB build projections. Great transport, awkward single-aggregate system-of-record.
- **Postgres:** the pragmatic default event store (unique `(stream, version)` = OCC; `LISTEN/NOTIFY` or logical replication to drive projections). Marten productizes this.
- **Akka Persistence / Axon (JVM), EventFlow / Marten (.NET):** application frameworks implementing aggregates, snapshots, upcasting.
- **Datomic:** a database whose entire model is immutable facts (datoms) with time built in — event sourcing’s ideas at the DB layer.
- **Git:** an event-sourced filesystem — commits are immutable events, the working tree is a projection, `git checkout` is time-travel.

11. Common pitfalls / war stories

- **Side effects in projections → email storms.** A rebuild re-fires every `SendShipmentEmail` since launch. Projections must be pure data transforms; side effects go in tracked reactors. The most expensive ES mistake.
- **Events as CRUD in disguise** (`AccountUpdated {fullStateBlob}`). You’ve kept all the costs and thrown away the value — no intent, no auditability of *why*. Model real domain facts

(`AddressCorrected`), not snapshots.

- **Aggregates too big** to “make things consistent.” A `Company` aggregate holding 10,000 orders means every order change contends on one stream’s version → constant concurrency conflicts and a serialization bottleneck. Smaller aggregates, sagas across them.
- **No `event_version` from day one.** First structural change becomes a heroic migration. Always stamp version + a metadata envelope (correlation id, causation id, timestamp, actor).
- **GDPR vs immutability.** “Right to be forgotten” meets an append-only log. The accepted answer is **crypto-shredding**: encrypt PII in events with a per-subject key; to “delete,” destroy the key, rendering those events unreadable while the log stays intact. Don’t discover this requirement after launch.
- **Unbounded streams.** A 5-year shopping cart with millions of events. Use snapshots, and *close* aggregates whose lifecycle has ended (an order eventually reaches a terminal state).
- **Treating snapshots as truth.** Someone “optimizes” by writing only snapshots and pruning events. Now you can’t rebuild, version, or audit — you’ve reinvented CRUD with extra steps.
- **Forgetting projection idempotency.** At-least-once + replays guarantee double-processing; non-idempotent `balance += amount` projections drift. Upsert by event id.
- **Adopting it everywhere.** Event-sourcing the CMS/settings/catalog buys nothing and taxes everyone. Scope it to the aggregates where history *is* the product.

🦋 Case Study: TigerBeetle — an event-sourced double-entry ledger

The problem. Payment processors, neobanks, exchanges, and energy markets all need to move money correctly, and they were doing it on general-purpose databases (Postgres/MySQL) wrapped in application code. At a real-time payments scale — think tens of thousands of transfers per second, each one a debit on one account and a credit on another that must balance to the cent — that stack falls apart. The TigerBeetle team measured incumbent ledgers handling roughly **1,000–2,000 transfers/sec** before lock contention and row-level coordination on hot accounts (the “settlement account” every transfer touches) became the wall. Worse, money systems carry the *hardest* correctness bar: you can never lose a transfer, never double-apply one, and you must be able to prove, years later, exactly how a balance became what it is. That last requirement is **events as source of truth + an immutable audit log** — the central thesis of this module — promoted to a non-negotiable product feature.

Applying the module’s concepts. TigerBeetle is event sourcing taken to its logical end: the only mutating operation is *append a transfer*. A transfer is an immutable, past-tense fact — `MoneyTransferred {debit_account, credit_account, amount}` — exactly the kind of event from §2. Accounts are not rows you `UPDATE`; an account’s balance is a **projection** (§3) folded from the transfers that touched it — the bank-statement-vs-balance inversion from §1, in production. There is no `UPDATE` / `DELETE` anywhere; a correction is a *new compensating transfer*, preserving the append-only log.

Because the ledger *is* the log, the dual-write problem of §9 disappears: **the event store is the outbox**. Replicas and downstream read models are catch-up subscribers on the same ordered log the leader

commits — no separate publish step to lose.

The write path is one strict total order — a single global stream, not one-stream-per-account — so the “won’t overdraw” invariant (§5) is enforced *synchronously* against the authoritative balance the instant a transfer is appended. This is the §2 optimistic-concurrency idea pushed to a serialized log: instead of compare-and-swap per stream, every transfer is sequenced into one consensus-ordered log, so two concurrent debits on the same account can never both pass the balance check.



The in-memory balances are a **read model** (§3) and a **snapshot** (§2): rather than re-folding the entire transfer history on restart, TigerBeetle periodically checkpoints state so recovery replays only the tail — the §2 **snapshot + tail** load path, where the snapshot is a *cache*, never truth. Wipe every checkpoint and the cluster rebuilds balances from the log alone, satisfying the rule “never let a snapshot become load-bearing.”

The trade-off they accepted. TigerBeetle gives up generality. It is **not** a SQL database — no ad-hoc queries, no arbitrary schema, only accounts and transfers with fixed 128-bit fields. It deliberately rejects the “many polyglot read models” flexibility of full CQRS (§3): the read side is essentially *one* shape (balances), because a ledger only needs to answer ledger questions. It also accepts a **single serialized write order** — sacrificing the per-aggregate concurrency §5 recommends — because for money, a strict total order *is* the feature: it makes double-spends and lost transfers structurally impossible. You trade the freedom to model anything for the guarantee that the one thing you model is always correct. Crucially, the write side here is **strongly consistent, not eventually consistent** — the eventual-consistency lag of §6 is pushed entirely onto downstream analytics/reporting consumers, never the balance check.

Results. By batching thousands of transfers per message, packing the hot data path into cache-friendly fixed-size records, and removing per-row locking in favor of one sequenced log, TigerBeetle reports on the order of **1,000,000 transfers/sec** on commodity hardware — roughly a **1000×** improvement over the few-thousand/sec ceiling of a transfer-on-Postgres design — while replicating across nodes for durability. The immutable log doubles as the audit trail for free: every balance is provably reconstructable from first principles, which is exactly what auditors and regulators demand.

Lessons

- **When history is the product (money, ledgers), event sourcing stops being optional architecture and becomes the spec.** The append-only log isn't overhead — it's the audit trail, the recovery mechanism, and the correctness proof in one.
- **Specialize the read model to the domain.** TigerBeetle skips polyglot CQRS because a ledger asks only ledger questions. Don't build five projections when one is the whole job — and don't reach for a general DB when a purpose-built one fits the invariant.
- **A strict total order can be a feature, not a bottleneck.** Serializing all writes sounds slow, but batching + cache-friendly layout made it *faster* than lock-coordinated row updates — and it made double-spends impossible by construction.
- **Push eventual consistency to where it's harmless.** Keep the invariant (the balance check) strongly consistent on the write log; let lag live only in downstream reporting projections, per §6.

12. Test yourself

1. Why must the `apply(state, event)` function never contain validation or reject an event? (*Hint: the event already happened; rejecting it would mean state can't be rebuilt deterministically. Validation belongs in the command handler before the event is created.*)
 2. You replay the event log to rebuild a read model and customers receive thousands of duplicate "order shipped" emails. What's the design rule that was violated, and where should that code have lived? (*Hint: projections must be side-effect-free; emails belong in a tracked reactor/process manager that records what it already handled.*)
 3. An aggregate has 800,000 events and command latency is 600 ms. What do you add, and what invariant must it never violate? (*Hint: snapshots; they're a cache — you must still be able to rebuild from events alone with snapshots deleted.*)
 4. How does an event store enforce that two concurrent withdrawals don't both succeed past the balance, without locking? (*Hint: optimistic concurrency — `append(expectedVersion)` is a compare-and-swap on the stream tail; the loser retries against fresh state.*)
 5. You must rename a field used in a 2019 event type. Give two valid strategies and one forbidden one. (*Hint: valid = upcaster `oldEvent→newEvent` on read, or emit a new event type with an upcaster; forbidden = mutating historical events in place.*)
 6. Distinguish CQRS from event sourcing, and give a CQRS-without-ES architecture. (*Hint: CQRS = separate read/write models; ES = events as truth. CRUD write DB + outbox/CDC → projections is CQRS with no ES.*)
 7. "Transfer \$100 from A to B" spans two account aggregates. Why can't this be one transaction, and what pattern handles it? (*Hint: aggregates are the transactional boundary; cross-aggregate = eventual consistency via a saga with compensating events.*)
 8. Why is "the event store is the outbox" a cleaner solution to the dual-write problem than a CRUD app bolting on an outbox table? (*Hint: there's no separate publish step to lose — projections are catch-up subscriptions on the same ordered log the write commits to.*)
-

13. Further reading

- **DDIA** (Kleppmann), *Designing Data-Intensive Applications* — Ch. 11 (Stream Processing: event sourcing, change capture, “the database inside out”) and Ch. 5 (Replication / logs).
- Martin Kleppmann, “**Turning the Database Inside Out with Apache Samza**” (talk + essay) — the foundational mental model for logs-as-truth.
- Greg Young — “**CQRS Documents**” and the EventStore talks; originator of the modern CQRS+ES framing.
- Eric Evans, **Domain-Driven Design** — aggregates, aggregate roots, consistency boundaries (Ch. 6).
- Vaughn Vernon, **Implementing Domain-Driven Design** — “Effective Aggregate Design” essays (the small-aggregates argument).
- Martin Fowler — “**Event Sourcing**”, “**CQRS**”, “**Domain Event**” articles (martinfowler.com) — concise canonical definitions.
- **EventStoreDB / Kurrent docs** — streams, optimistic concurrency, catch-up subscriptions, projections.
- **TigerBeetle docs** — a production event-sourced ledger; excellent on why finance needs this.
- Pat Helland, “**Immutability Changes Everything**” (CIDR 2015) — the deep argument for append-only systems.
- Cross-links: [12-message-queues-and-the-outbox-pattern.md](#) (outbox, sagas, idempotency), [06-replication-and-consistency.md](#) (logs, eventual consistency), [09-cap-pacelc-and-consistency-models.md](#) (read-your-writes, consistency models).

14 · Stream Processing & Real-Time Systems

Source: [readme/research/system-design/14-stream-processing-realtime.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains how computer systems can react to information the moment it arrives — instead of waiting until the end of the day to run a report. It matters because things like catching fraud, showing live order counts, or updating a leaderboard require answers in seconds, not hours. Understanding this helps you design (or choose) systems that stay fast and accurate even under heavy load.

The main parts explained simply:

- **Batch vs Stream** — A batch job is like waiting until the end of the day to count all your orders at once. A stream job is like counting every order the moment it comes in, continuously, with no stopping point. Batch is simpler and great for daily reports; streaming is needed when delays cost money.
- **Lambda vs Kappa architecture** — Two ways to build a streaming system. Lambda runs two separate pipelines side by side (one fast but rough, one slow but accurate) — this is a headache because you write the same rules twice and they slowly drift apart. Kappa runs just one pipeline and replays saved history to correct mistakes. Kappa is the recommended approach for new systems.
- **Event time vs processing time** — An event's "event time" is when it actually happened (e.g. a customer clicked at 2:03 PM). "Processing time" is when your server saw it (e.g. 2:07 PM because of a network delay). If you use the wrong clock, replaying old data gives different totals each time. Always use event time for anything where the numbers need to be consistent.
- **Windowing** — Because a stream never ends, you slice it into time buckets to do calculations. A "tumbling" window is non-overlapping fixed buckets (every 1 minute). A "sliding" window overlaps (last 5 minutes, checked every 1 minute). A "session" window closes automatically after a quiet gap (like a browser session). Choosing the wrong window type wastes memory or gives wrong results.
- **Watermarks and late data** — A watermark is the system's best guess that all events up to a certain moment have arrived. When the watermark passes the end of a time bucket, the system publishes that bucket's result. Events that arrive after the watermark are "late." You can drop them, route them aside for separate handling, or update the result — each option trades speed for accuracy.
- **Stateful processing and checkpointing** — Some calculations need to remember things (e.g. running totals per customer). This stored memory is called "state." If a server crashes, the system must restore that memory from a saved snapshot (checkpoint) and re-read events from where it left off — otherwise counts are wrong. Flink (a popular engine) does this without stopping the flow of data.
- **Exactly-once processing** — This means each event is counted exactly once in the final output, even if the network delivers it twice or a server crashes and replays it. Achieving this requires either writing results in a way that duplicates overwrite each other (idempotent), or using database-style transactions.
- **Stream-table duality** — A stream is a list of changes over time ("add 5", "subtract 3"). A table is the current total ("balance = 2"). They are two sides of the same coin. Modern systems like ksqlDB let you write a simple SQL query and the engine keeps the answer table up to date automatically as new events arrive — no scheduled refresh needed.

- **Joins on streams** — Combining two streams (e.g. matching an ad click to a later purchase) is tricky because both streams are infinite. You must set a time window (“only match events within 30 minutes of each other”) to cap how much data the system has to remember. Without a window, memory grows forever and the system crashes.
- **Choosing an engine** — Flink is the most powerful tool for complex, low-latency work. Kafka Streams is simpler and runs inside your existing app with no separate server to manage. Spark Streaming is good if your team already uses Spark. SQL-based tools (ksqlDB, Materialize) are easiest if your team thinks in SQL.

What to do with this:

If you are building anything that needs real-time totals, fraud alerts, or live dashboards — always timestamp events at the source (event time), not at the server (processing time). Start with Kappa (one pipeline, replayable log) rather than Lambda (two pipelines) to avoid maintaining the same logic in two places.

What you'll learn: How to process unbounded data continuously instead of in nightly batches — the architectures (Lambda vs Kappa), the deep semantics of *time* (event time vs processing time, watermarks, late data), how to do windowed aggregations correctly, and how systems like Flink and Kafka Streams deliver *exactly-once* results despite crashes. You'll leave able to reason about a fraud-detection or leaderboard pipeline the way a staff engineer or an interviewer would.

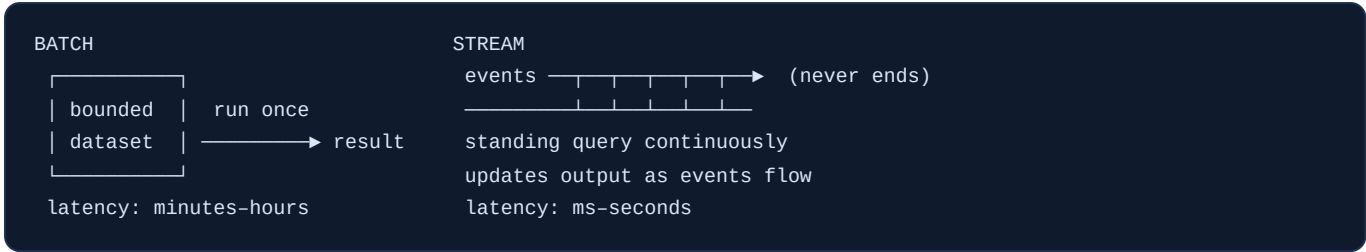
Prerequisites: Read [02-data-models-storage.md](#) (logs, LSM trees), [06-replication-consistency.md](#) (log-based replication), [11-messaging-event-driven.md](#) (Kafka, partitions, offsets, delivery semantics), and ideally [09-cap-pacelc...md](#). Stream processing *is* event-driven architecture with stateful operators bolted on — module 11 is the foundation.

1. Intuition: from snapshots to a river

A **batch** job is a photograph. You collect a bounded chunk of data (yesterday's orders), run a job over the whole thing, and emit a result. The input has a known start and end. MapReduce, Spark, your nightly `php artisan report:generate` cron — all batch.

A **stream** is a river. Data never stops arriving; there is no “end of input.” You can't wait for all the data because there is no “all.” Instead you run a *standing query* that continuously consumes events and continuously updates output. Think of a Laravel queue worker that never drains and maintains running totals in memory — that's the seed of stream processing.

The hard part isn't “process events one at a time” (a `for` loop does that). The hard part is doing **stateful aggregations over time** — “count failed logins per user in any 5-minute window” — *correctly*, when events arrive out of order, late, or duplicated, and when your processing node crashes mid-window.



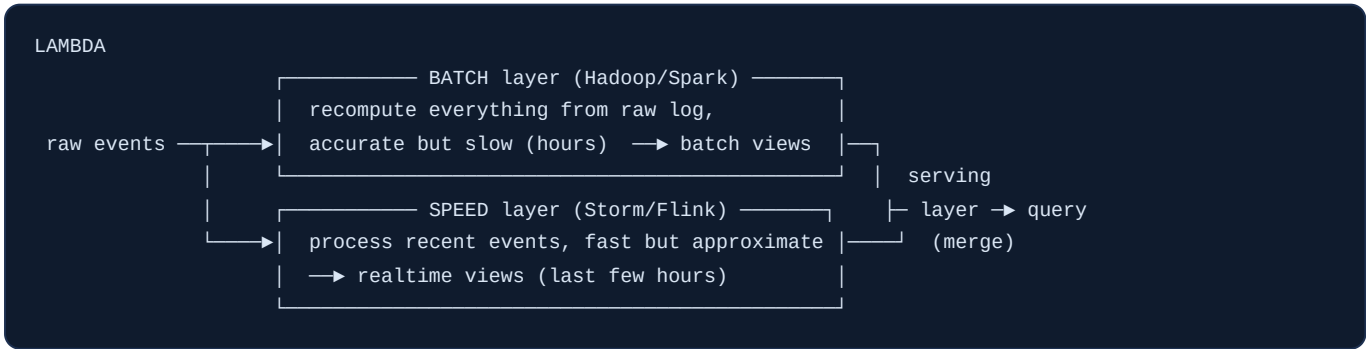
2. Batch vs Stream — the real trade-off

Dimension	Batch	Stream
Input boundary	Bounded (known end)	Unbounded
Latency	Minutes to hours	Milliseconds to seconds
Reprocessing	Trivial — rerun the job	Hard — must replay the log
Correctness model	Easy (you see all data)	Subtle (time, ordering, late data)
Throughput efficiency	Very high (sequential scans)	High but per-event overhead
Failure recovery	Re-run from input	Checkpoint + replay
When to use	Daily reports, ML training, backfills, anything tolerant of hours of lag	Fraud detection, leaderboards, alerting, live dashboards, anything where staleness costs money

Key insight (Tyler Akidau, *The Dataflow Model*): **batch is a special case of streaming** — a bounded stream. A well-designed stream engine can run batch jobs by treating a file as a finite stream. Flink and Beam are built on exactly this premise; this is why “unify batch and streaming” became the industry goal.

3. Lambda vs Kappa architecture

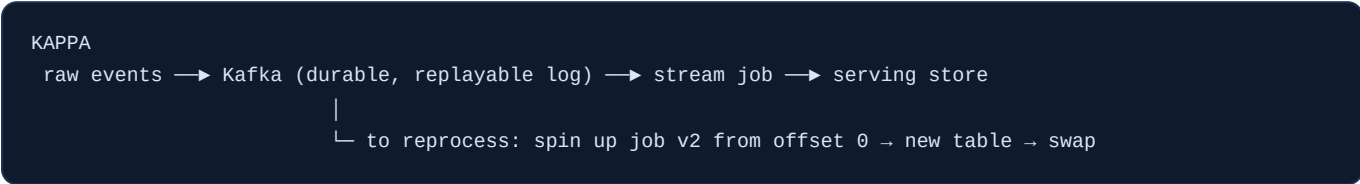
Early streaming engines (Storm, ~2011) were fast but gave only *approximate*, at-least-once results. So Nathan Marz proposed **Lambda**: run two pipelines.



The serving layer merges: “old data from the accurate batch view + recent data from the fast speed view.”
The batch layer periodically overwrites the speed layer’s approximations.

The fatal flaw of Lambda: you write and maintain *the same business logic twice* — once in your batch framework, once in your stream framework, in different languages with different semantics. Bugs diverge. This is operational hell.

Kappa (Jay Kreps, 2014) deletes the batch layer. Insight: if your stream engine is correct (exactly-once) and your log (Kafka) retains enough history, you don’t need batch at all. To “reprocess,” you just **replay the log from offset 0 through a new version of your stream job** into a new output table, then switch over.



	Lambda	Kappa
Codebases	Two (batch + stream)	One (stream)
Reprocessing	Re-run batch	Replay log
Complexity	High (merge logic, dual maintenance)	Lower, but needs long log retention + exactly-once engine
Storage cost	Batch storage	Long Kafka retention (tiered storage helps)
When to use	Legacy, or batch logic genuinely differs from stream	Default for new systems with a modern engine (Flink)

Today, with Flink/Beam giving correct exactly-once streaming and Kafka tiered storage making long retention cheap, **Kappa is the default recommendation**. Lambda survives mostly in legacy stacks.

4. The deepest idea: event time vs processing time

This is *the* concept that separates toy stream code from production-correct pipelines, and the one interviewers probe.

- **Event time** — when the event *actually happened* (timestamp baked into the event by the producer: “user clicked at 14:03:01”).
- **Processing time** — when your operator *observes* the event (wall clock on the worker: “I read this at 14:07:22”).

These diverge because of network delay, queueing, retries, mobile devices going offline, partition rebalancing, and backpressure. A phone in a tunnel buffers events and dumps them 20 minutes later — same event time, wildly later processing time.

```

event time:  |--e1--e2--e3--e4--e5--▶ (true order, true timestamps)
              \   \   \   \
network/queue delays
              ▼   ▼   ▼   ▼
proc. time:  |--e1-----e3--e2----e5--e4▶ (out of order, delayed)

```

Why it matters: “Count purchases between 14:00 and 14:05” must mean *event* time, or your numbers are nondeterministic — rerun the job tomorrow on the same data and you’d get different bucketing because processing time changed. **Processing-time windows are non-reproducible.** Use event time for any correctness-sensitive aggregation; processing time only when you genuinely mean “what I’ve seen so far” (e.g., coarse system-health counters where exactness doesn’t matter).

The price of event time: events arrive **out of order** and **late**, so a window for [14:00, 14:05) may still receive a member at processing-time 14:09. When do you *close* the window and emit a result? Enter watermarks.

5. Windowing

You can’t aggregate an infinite stream; you slice it into finite **windows**.

Window type	Shape	Example use
Tumbling	Fixed size, non-overlapping, contiguous	“Revenue per 1-minute bucket”
Sliding (hopping)	Fixed size, fixed slide < size → overlap	“Failed logins in last 5 min, updated every 1 min”
Session	Dynamic — gap-based; closes after N idle seconds	“User browsing session” (close after 30 min inactivity)
Global	One window, all time; needs a custom trigger	Custom triggering logic

```

TUMBLING (size 5):  [0-5)[5-10)[10-15)      each event in exactly one window

SLIDING (size 10, slide 5):
    [0-10)
    [5-15)
    [10-20)      an event belongs to MULTIPLE windows

SESSION (gap 5):    e e e  ...gap...  e e      two sessions (gap broke them)
                    └─sess A─┘          └─sess B─┘

```

Back-of-envelope on sliding-window cost: a 24-hour sliding window with a 1-minute slide means each event is replicated into **1440** window panes — naive implementations blow up memory. Production engines use **panel/incremental aggregation**: compute per-slide-interval partial aggregates once and combine them, so an event is processed once, not 1440 times.

Session windows are the hardest because their boundaries are data-dependent: a late event can *merge two existing sessions into one*. The engine must support window merging in its state backend.

6. Watermarks & late data

A **watermark** is the engine's assertion: "*I believe I have now seen all events with event-time $\leq T$.*" It's a moving low-water mark on event time that flows through the pipeline as a special record interleaved with data.

When the watermark passes the end of a window, the engine fires (emits) that window's result and (optionally) frees its state.

```
watermark W advances →
event times: 14:00 14:01 14:03 14:02 14:04 | W=14:05 → fire window [14:00,14:05)
               ▲ out of order but still ≤ W, included
later:        14:04(LATE, arrives after W) → dropped or sent to side output
```

The fundamental trade-off — there is no free lunch:

Watermark strategy	Latency	Completeness	Risk
Aggressive (small lag, e.g. $W = \text{max\_seen} - 2s$)	Low — fire fast	Lower — more late events miss the window	Wrong/incomplete results
Conservative (large lag, e.g. $- 5\text{min}$)	High — hold windows open longer	Higher — catch stragglers	Stale results, more state held

You cannot have both low latency and full completeness when data can be arbitrarily late — this is a direct consequence of the unbounded-delay reality. Practical heuristics:

- **Bounded-out-of-orderness:** `watermark = maxEventTime - fixedDelay` (e.g. 30s). Simple, what most teams use.
- **Percentile/adaptive** watermarks: learn the lateness distribution.

Handling late data (events past the watermark):

1. **Drop** — default; fine for dashboards where 0.1% loss is irrelevant.
2. **Side output / dead-letter** — route late events elsewhere for separate handling or backfill.
3. **Allowed lateness + updates** — keep window state alive `allowedLateness` past the watermark; re-fire and *emit a correction* (retraction/update) when a late event lands. This needs a downstream sink that can handle updates (upsert), which connects to stream-table duality (§8). Flink and Beam both support this; it's the gold standard for correctness.

The **trigger** (when to emit) is orthogonal to the window (how to bucket): you can emit early/speculative results before the watermark, then a final result at the watermark, then late updates after — the *Dataflow* model's "what/where/when/how" decomposition (what to compute / where in event time / when to fire / how refinements relate).

7. Stateful processing, checkpointing & exactly-once

A stateless operator (filter, map) is trivial to recover — just reprocess. A **stateful** operator (counts, windows, joins, dedup sets) holds accumulated state that must survive crashes. This is where streaming gets genuinely distributed-systems-hard.

State backend: local, embedded key-value store per operator instance. Flink uses **RocksDB** (an LSM tree — see [02-data-models-storage.md](#)) for state that exceeds RAM, or a heap backend for small state. State is partitioned by key, co-located with the operator processing that key.

The recovery problem: if a worker dies, you must restore its state *and* resume input at a consistent point, so you neither lose nor double-count events.

Checkpointing (Flink — the Chandy-Lamport asynchronous barrier snapshot algorithm):

```
Source injects a "barrier" into the stream every N ms:
—e—e—[BARRIER n]—e—e—[BARRIER n+1]—>
Each operator, on receiving barrier n on ALL its inputs:
  1. snapshots its own state (async, to durable store: S3/HDFS)
  2. forwards barrier n downstream
When all operators + sinks ack barrier n → checkpoint n is COMPLETE & durable.
On crash: restore all state to checkpoint n, rewind source offsets to barrier n.
```

The barrier “cuts” the stream into a consistent global snapshot *without stopping data flow* — that’s the elegance of Chandy-Lamport. Snapshots are written asynchronously so throughput barely dips.

Exactly-once: what it actually means. Networks duplicate; “exactly-once *delivery*” is impossible in general. What systems deliver is **exactly-once *processing semantics / effect*** — the *state and output* reflect each event exactly once, even though the event may be physically delivered multiple times. Achieved by:

- **Checkpointed state** rewound atomically (above) → internal state is exactly-once.
- **For external sinks**, you need one of:
 - **Idempotent writes** — keying by a deterministic id so a replay overwrites rather than duplicates (upsert into Postgres/Cassandra). Easiest, most common.
 - **Transactional / two-phase-commit sink** — pre-commit on checkpoint, commit on checkpoint-complete (Flink’s [TwoPhaseCommitSinkFunction](#) ; Kafka transactions with [read_committed](#)). End-to-end exactly-once, but adds latency and requires a transactional sink.

Delivery semantic	Mechanism	Cost	When
At-most-once	Fire and forget	Cheapest	Lossy-tolerant metrics
At-least-once	Ack + retry, no dedup	Cheap	Idempotent downstream, or dupes OK
Exactly-once (effect)	Checkpoint + idempotent/txn sink	Higher latency, more coordination	Money, billing, fraud, counts that must be right

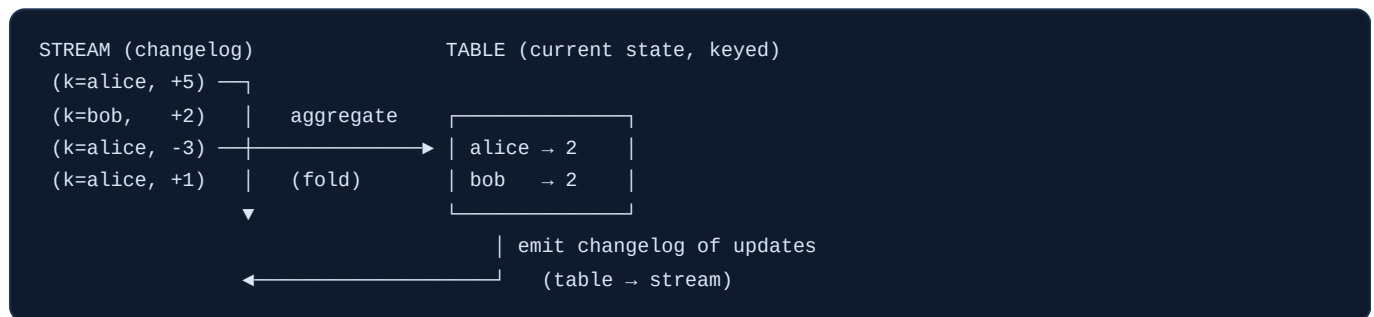
War story baseline: Kafka Streams achieves exactly-once via Kafka transactions across the consume → process → produce loop (`processing.guarantee=exactly_once_v2`). Flink achieves it via barriers + transactional sinks. Spark Structured Streaming uses offset-based checkpointing in a write-ahead log + idempotent sinks (micro-batch, so higher latency).

8. Stream-table duality

The single most important mental model in modern stream processing (Kafka Streams, ksqlDB, Materialize, Flink SQL all rest on it):

- A **stream** is a log of *changes* (an append-only sequence of events: “balance += 5”, “balance -= 3”).
- A **table** is a *snapshot* — the current state, the result of folding all changes up to now (“balance = 2”).

They are two views of the same data — **a changelog** ↔ **a materialized state**:



- **Stream** → **Table**: aggregate/fold the stream by key (a `GROUP BY` that never ends) → a continuously updated table = a **materialized view**.
- **Table** → **Stream**: emit the table’s change events (CDC — change data capture; see `11-messaging-event-driven.md`). This is exactly what Debezium does to your Postgres WAL.

This duality is why “materialized-view databases” (Materialize, RisingWave, ksqlDB) exist: you write a SQL `SELECT ... GROUP BY` and the engine keeps the result table *incrementally* up to date as input streams change — no periodic `REFRESH` . Under the hood it’s a stream job; the SQL is sugar.

9. Joins on streams

Joins are where unbounded data bites hardest, because a SQL join assumes both sides are *finite*. Three flavors:

Join type	What it joins	State held	Real example
Stream-stream (windowed)	Two streams, matched within a time window	Both sides buffered for the window duration → bounded	Match <code>ad_click</code> to <code>purchase</code> within 30 min (attribution)
Stream-table (lookup/enrichment)	Stream event ← current table snapshot	The table (kept fresh via its changelog)	Enrich <code>order</code> event with <code>customer</code> profile
Table-table	Two changelog tables	Both materialized	Keep a denormalized view in sync

Why stream-stream joins are dangerous: without a window, you'd have to remember *every* event from both sides forever to catch a future match — unbounded state, certain OOM. The window bounds the state: “only try to match events within 30 minutes of each other.” Watermarks then let the engine *evict* expired buffered events. Picking the window is a business decision: too small and you miss legitimate late matches; too large and state explodes.

Temporal correctness in stream-table joins: when enriching an order with the customer record, do you want the customer's profile *as it was at the order's event time*, or *as it is now*? Naive joins use “now,” which is wrong for reprocessing (replay the log and the customer's address has since changed → historical orders get rewritten incorrectly). **Temporal/versioned table joins** (Flink's `FOR SYSTEM_TIME AS OF`) join against the table version valid at the event's event time — essential for auditable, replayable pipelines.

10. Use cases mapped to the machinery

Real-time analytics / live dashboards. Tumbling/sliding windows → materialized view → push to dashboard. Acceptable to drop late data (drop strategy) and use bounded-out-of-orderness watermarks. Tools: Flink SQL → Apache Pinot / Apache Druid (sub-second OLAP) → Superset. Latency target: ~1–5s end to end.

Fraud / anomaly detection. Stateful pattern detection over event-time windows: “≥3 transactions > \$500 from ≥2 countries within 5 minutes.” Needs low latency (block the transaction *before* it settles → seconds matter) but *also* correctness, so exactly-once + careful watermarks. Often combines stream-stream joins (login ↔ transaction) with a state machine (CEP — Complex Event Processing; Flink CEP). State per key (per card) can be large → RocksDB backend.

Leaderboards / top-K / counters. Stream → table aggregation by player key, kept in a materialized view, served from Redis (sorted sets, `ZADD` / `ZREVRANGE` give $O(\log n)$ ranked reads). Watch out: a global top-K is a *single hot key* — every event touches it. Partition by region/segment and merge, or use approximate sketches (Count-Min Sketch, HyperLogLog for unique counts) to bound memory: HLL estimates cardinality of billions of items in ~12 KB with ~2% error. Classic space-vs-accuracy trade-off.

11. Common pitfalls / war stories

- **Using processing time and wondering why numbers aren't reproducible.** Backfill the same data and get different per-window counts. Always default to event time for correctness-sensitive aggregates.
 - **No watermark / infinitely conservative watermark.** Windows never close (or hold state forever) → state grows unbounded → RocksDB/heap OOM and the job dies. Always set a finite, sane out-of-orderness bound and TTL old state.
 - **Silently dropping late data without measuring it.** A mobile-heavy product had 4% of events arriving past a 30s watermark; revenue dashboards undercounted and nobody knew until finance reconciled. Emit a *late-records counter* and alert on it.
 - **Assuming "exactly-once" means no duplicate delivery.** It means exactly-once *effect*. If your sink isn't idempotent or transactional, replays after a crash will duplicate writes regardless of what the engine guarantees internally.
 - **Unbounded stream-stream join.** Forgetting the time window → state grows forever. Always window stream-stream joins and verify state size in monitoring.
 - **Hot-key skew.** One viral product / one whale user routes 60% of events to a single partition's operator → that one worker is the bottleneck while others idle. Mitigate with key salting + two-stage aggregation, or pre-aggregate.
 - **Changing the job in an incompatible way and breaking state restore.** Renaming an operator or changing its state schema can make the saved checkpoint unrestorable. Use explicit operator UIDs (Flink) and state migration / savepoints.
 - **Watermark stalls on an idle partition.** Event-time watermark = *min* across all partitions; if one partition goes silent, the global watermark freezes and *nothing* fires. Use idle-source detection to advance past quiet partitions.
 - **Lambda divergence.** Batch and speed layers compute subtly different results (different rounding, different dedup) → the dashboard flickers when the batch view overwrites. The reason Kappa won.
-

12. Choosing an engine

Engine	Model	Latency	Exactly-once	Sweet spot
Kafka Streams	Library (no cluster), reads/writes Kafka	ms	Yes (Kafka txns)	JVM apps already on Kafka; embed in a service, no separate cluster
Apache Flink	True streaming, dedicated cluster	ms, lowest	Yes (barriers + txn sinks)	Complex stateful jobs, CEP, large state, event-time correctness — the power tool
Spark Structured Streaming	Micro-batch (Continuous mode experimental)	~100ms—seconds	Yes (WAL + idempotent)	Teams already on Spark; unified batch+stream; higher latency OK
ksqlDB / Flink SQL / Materialize / RisingWave	SQL over streams (duality)	sub-second—seconds	Yes	Materialized views, analysts who want SQL not Java

Rule of thumb: **Flink** for serious low-latency stateful work; **Kafka Streams** when you want it embedded in a microservice with no cluster to operate; **Spark** when you're already a Spark shop; a **materialized-view DB** when SQL ergonomics beat raw control.

Case Study: Uber's real-time pipeline (AthenaX / Flink)

The problem. Uber's marketplace is a feedback loop measured in seconds: surge pricing, ETA, driver-rider matching, fraud, and the city-ops dashboards that humans use to keep a market healthy all need *fresh* aggregates over *enormous* event volume. By the late 2010s Uber's streaming platform processed on the order of **trillions of messages and petabytes of data per day**, with Kafka clusters handling **tens of millions of messages/sec** at peak. The events come from phones — and phones are the worst possible event source for timing: a driver in a parking garage or a rider on the subway buffers GPS pings and trip events, then dumps them minutes later. Computing “trips completed in this 1-minute bucket in San Francisco” naively on arrival gives numbers that are wrong, jittery, and *non-reproducible* — exactly the failure described in §4.

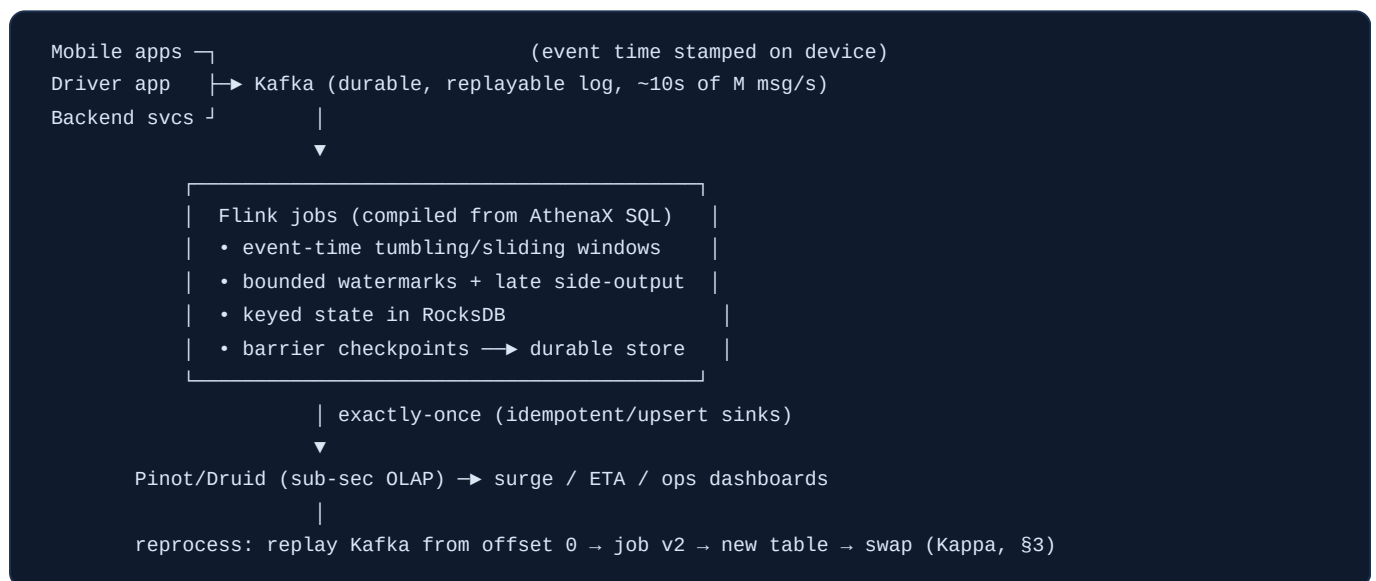
Why event time, not processing time (§4). A trip that *happened* at 14:03 but *arrived* at 14:09 must still land in the 14:03 bucket, or every replay produces different per-window counts and surge pricing reacts to phantom demand. Uber standardized on **event time** — the timestamp the mobile SDK stamps on the event — for all correctness-sensitive aggregates. This is precisely the “processing-time windows are non-reproducible” lesson from the module, applied at city scale.

Windowing + watermarks (§5, §6). Demand/supply and dashboard metrics are computed as **tumbling and sliding event-time windows** (e.g. trips per minute, sliding 5-minute supply/demand ratios per geofenced hex). Because phones are late, Uber configures **bounded-out-of-orderness watermarks** — hold the window open a bounded lag past `maxEventTime` to catch stragglers — and routes events past

the watermark to a **late-data side output** rather than silently dropping them (§6, pitfall in §11). The accepted trade-off is the exact one in the §6 table: a larger watermark lag buys completeness for delayed mobile events at the cost of a few extra seconds of dashboard latency. For a surge signal, “a few seconds late but correct” beats “instant but undercounting trips stuck in a tunnel.”

Stateful processing + checkpointing (§7). Per-geofence, per-product counters and the per-driver/per-card state behind fraud rules are large keyed state, kept in Flink’s **RocksDB state backend** (the LSM-tree backend from §7/§2) so it spills past RAM. Fault tolerance is Flink’s **Chandy-Lamport asynchronous barrier checkpointing** — barriers injected at the Kafka sources, async per-operator snapshots to durable storage, source offsets rewound on recovery — giving **exactly-once processing effect** even when workers die mid-window. Sinks use **idempotent/upsert writes** keyed by a deterministic id so a post-crash replay overwrites rather than double-counts (§7).

Kappa, made usable with SQL (§3, §8). Uber did *not* maintain twin batch+stream codebases. The platform is essentially **Kappa**: Kafka is the durable, replayable log; to reprocess you replay from an earlier offset through a new job version into a new table and swap. To make this accessible to non-streaming engineers, Uber built **AthenaX** — a self-serve layer that compiles **SQL** into managed Flink jobs. That SQL-over-streams ergonomic rests directly on **stream-table duality** (§8): a **GROUP BY** that never ends becomes a continuously updated materialized view. Output lands in OLAP stores (Pinot/Druid-class) for sub-second queries by dashboards.



The key trade-off they accepted. Latency vs completeness (§6), spent in favor of completeness. By holding event-time windows open for a bounded lateness rather than firing the instant an event arrives, Uber gave up a few seconds of freshness to get aggregates that are *correct and reproducible* across replays — the right call when those numbers drive pricing and money. They also accepted the operational cost of long Kafka retention and large RocksDB state (the §3 Kappa cost) to avoid Lambda’s twin-codebase divergence (§11).

Results. AthenaX let hundreds of streaming use cases be authored as SQL instead of bespoke Flink jobs, collapsing time-to-ship from weeks to hours and removing the Lambda dual-maintenance tax. The Flink-on-Kafka platform sustained trillions of daily events with **second-scale end-to-end latency** for analytics

and **sub-second** OLAP query latency on the served tables, with exactly-once correctness surviving worker failures via checkpointing.

Lessons

- **Mobile = late data is the norm, not the exception.** If your producers are phones, design for event time + bounded watermarks + a measured late-records side output from day one; never assume on-time arrival.
- **Pick completeness over raw latency when numbers move money.** A bounded watermark lag is a cheap, explicit way to buy reproducible aggregates — make the lag a tuned knob, not an accident.
- **SQL-over-streams scales an org, not just a cluster.** Stream-table duality (AthenaX) let non-experts ship correct stateful pipelines; the duality is the leverage, not the syntax.
- **Kappa + replay beats Lambda's twin codebases.** A replayable log plus an exactly-once engine removes the divergence bug class entirely — at the price of long retention and large checkpointed state.

Test yourself

1. Why are processing-time windows non-reproducible, and when is that actually acceptable? *Hint: bucketing depends on wall-clock arrival, which varies per run; OK only when “as I saw it” is the intended semantic.*
 2. A window for [10:00, 10:05) — when does the engine emit its result, and what makes an event “late”? *Hint: when the watermark passes 10:05; late = arrives after that watermark.*
 3. You want lower dashboard latency but keep missing stragglers. What's the fundamental trade-off and three ways to handle late data? *Hint: latency vs completeness; drop / side-output / allowed-lateness-with-retractions.*
 4. Explain exactly-once *processing* vs exactly-once *delivery*. Why can your sink still see duplicates under an “exactly-once” engine? *Hint: effect vs physical delivery; sink must be idempotent or transactional.*
 5. How does Flink take a consistent global snapshot without stopping the stream? *Hint: Chandy-Lamport barriers injected at the source, async state snapshot per operator.*
 6. Give the stream-table duality in one sentence each direction, and name a real system for each. *Hint: stream → table = aggregate/materialize (ksqlDB); table → stream = CDC changelog (Debezium).*
 7. Why must a stream-stream join be windowed, and what does the window's size trade off? *Hint: bounds otherwise-infinite state; size = match completeness vs memory.*
 8. You're building a global leaderboard and one operator is pegged at 100% CPU. Diagnose and fix. *Hint: hot key / single top-K key; salt + two-stage aggregate, or approximate sketches.*
-

Further reading

- **DDIA**, Martin Kleppmann — Chapter 11 *Stream Processing* (the canonical text); Chapter 12 for the “unbundled database” / derived-data view.

- **The Dataflow Model**, Akidau et al., VLDB 2015 — the what/where/when/how framework; event time, watermarks, triggers. The foundational paper.
- **Streaming 101 & 102**, Tyler Akidau (O'Reilly Radar blog) — the accessible companion to the paper.
- **Lightweight Asynchronous Snapshots for Distributed Dataflows**, Carbone et al. (Flink checkpointing) + the original **Chandy-Lamport** snapshot paper (1985).
- **Questioning the Lambda Architecture**, Jay Kreps, 2014 — the Kappa argument.
- Apache **Flink** docs: *Event Time, Watermarks, State & Fault Tolerance*. **Kafka Streams** docs: *Streams & Tables*, [exactly_once_v2](#) .
- **Materialize / RisingWave** docs — materialized-view-as-a-stream-job in practice.

Sibling modules: [11-messaging-event-driven.md](#) (Kafka, partitions, delivery semantics, CDC), [02-data-models-storage.md](#) (LSM/RocksDB state backends), [06-replication-consistency.md](#) (log replication), [09-cap-pacelc...md](#) (latency vs consistency framing that mirrors latency vs completeness here).

15 — Probabilistic Data Structures & Algorithms at Scale

Source: [readme/research/system-design/15-probabilistic-structures-and-algorithms.md](#)

IN SHORT — KEY TAKEAWAYS

This document is about clever shortcuts that let big software systems work fast without using too much memory. Instead of tracking every exact detail, these tools keep a tiny, rough summary and accept a small, controlled amount of error in return. Understanding these ideas explains why platforms like Reddit, Google, and Cassandra can handle billions of events without needing a warehouse full of servers.

The main parts explained simply:

- **Bloom / Cuckoo filters (Is this thing already seen?)** — A tiny checklist that can confidently say “definitely not seen” but might occasionally say “yes” when the answer is actually “no.” Used by databases to skip searching files that can’t possibly have what you need, saving huge amounts of disk reading. Cuckoo filters work the same way but also allow removing items.
- **HyperLogLog (How many unique visitors?)** — Counts how many distinct things appeared in a stream (like unique website visitors) using only about 12 KB of memory, no matter if there were a thousand or a billion visitors. Reddit uses this for its “X views” post counter. The trade-off: you get an answer within about 1% accuracy, not an exact number — which is perfectly fine for a display counter.
- **Count-Min Sketch and Top-K (What are the most popular items?)** — A grid of counters that tracks how often each item appears. It can tell you which items are “hot” (trending searches, busy API keys, popular products) using only tens of kilobytes. It always over-counts a little, never under-counts. Pair it with a “Top-K” tracker to find the actual most popular items.
- **t-digest and DDSketch (What is my slowest 1% of requests?)** — Tools for measuring percentiles (like “what speed do 99% of orders complete within?”) across many servers at once, without storing every single measurement. Normal averages don’t work for this — you need these mergeable sketches. Used in monitoring dashboards and performance tracking.
- **Merkle trees (How do two copies of data stay in sync?)** — A tree of checksums that lets two database replicas quickly find which rows are different, without comparing every row. Like comparing chapter summaries before reading pages word for word. Used by Cassandra and DynamoDB to repair out-of-sync copies efficiently.
- **Consistent hashing (How do we spread data evenly across many servers?)** — A method for deciding which server stores which piece of data. When you add or remove a server, only a small fraction of data needs to move — not everything. Used by Cassandra, Memcached, and load balancers. “Virtual nodes” (multiple slots per server) make the load spread even more evenly.
- **Geohash, Quadtrees, S2 (How do we find nearby locations fast?)** — Ways to turn two-dimensional map coordinates into a single value that a normal database index can search. Geohash is simple and widely supported; quadtrees adapt to dense areas; Google’s S2 is the most accurate for planet-scale apps. Used by Redis, Uber, and Google Maps for “find nearby” queries.
- **Skip lists (How do sorted lists stay fast to search and update?)** — A layered linked list with “express lanes” that jumps over many items at once, making search and insert as fast as a balanced tree but simpler to build for concurrent (multi-user) access. Redis sorted sets use this internally.

What to do with this:

When an exact answer is not required (view counts, visitor counts, “is this popular?”), reach for one of these sketches — they use a tiny, fixed amount of memory no matter how big the data grows. Always know which direction the error goes (over-count vs. under-count vs. false positive) before choosing one, so you pick the right tool for your situation.

What you’ll learn: How to trade a tiny, bounded amount of accuracy for orders-of-magnitude less memory and CPU, and the specific data structures that make “good enough, but at billion-scale” possible — Bloom/Cuckoo filters, HyperLogLog, Count-Min Sketch, Top-K, t-digest, Merkle trees, consistent hashing, spatial indexes (geohash/quadtree/S2), and skip lists. For each you’ll get the intuition, the precise mechanics, the error bounds with real math, and the production system that ships it.

Prerequisites: Read [01-foundations-and-estimation.md](#) (back-of-envelope math, the latency numbers), [08-replication-and-consistency.md](#) and [09-cap-pacelc...md](#) (for Merkle anti-entropy), and [10-partitioning-and-sharding.md](#) (consistent hashing is the partitioner). Hashing fundamentals from [02-...](#) help throughout.

0. The space/accuracy trade-off mindset

Most data structures you learned (hash map, B-tree, balanced BST) are **exact and complete**: they store every element and answer every query precisely. That costs $O(n)$ memory. At web scale n is a billion uniques per day, 100M distinct search terms, 50M concurrent geo-points. Exact structures blow past RAM, and RAM is where latency lives (see the latency table in [01](#) : RAM read ≈ 100 ns, SSD ≈ 16 μ s, network round trip ≈ 0.5 ms).

The probabilistic move: **stop storing elements; store a lossy fingerprint of the stream**. You give up exactness in a *controlled, mathematically bounded* way and get:

EXACT structure		PROBABILISTIC structure
memory $\propto n$		memory $\propto 1/\epsilon^2$ or $\log \log n$
answer always correct	$\rightarrow$	answer correct $\pm \epsilon$,
$n=1e9 \rightarrow$ GBs		with prob $1-\delta$
		$n=1e9 \rightarrow$ KBs

The discipline is: **know which way the error points**. Bloom filters never produce false negatives but can produce false positives. HyperLogLog is unbiased (error straddles zero). Count-Min only *over*-counts, never under. An engineer who confuses the direction of error ships a correctness bug, not an approximation.

Three questions to ask of any sketch:

1. **What does it answer?** (membership, cardinality, frequency, quantile, set difference, location)
2. **Which direction is the error, and is it bounded?** (one-sided vs two-sided; ϵ additive vs multiplicative)

3. **Is it mergeable?** (Can two sketches from two shards be combined into the sketch of the union? This is what makes them work in MapReduce/streaming.) Mergeability is the property that separates a toy from a distributed-systems tool.

1. Membership: Bloom filters and Cuckoo filters

Analogy. A nightclub bouncer with a mental list. “Have I seen this face?” If he says *no*, you’ve definitely never been in (no false negatives). If he says *yes*, he might be confusing you with someone (false positive). He keeps no photos — just a fuzzy impression — so the “list” costs almost nothing.

Bloom filter mechanics

A bit array of m bits, all zero, plus k independent hash functions.

- **Insert(x):** set bits $h_1(x) \bmod m, \dots, h_k(x) \bmod m$ to 1.
- **Query(x):** if *all* k bits are 1 $\rightarrow$ “probably present”; if *any* is 0 $\rightarrow$ “definitely absent.”

```
m = 16 bits, k = 3
insert "cat": positions 2, 7, 11  $\rightarrow$  0010000100010000
insert "dog": positions 5, 7, 14  $\rightarrow$  0010010100010010
query "fox"  $\rightarrow$  2,5,14 all set? yes  $\rightarrow$  FALSE POSITIVE (fox never inserted)
query "owl"  $\rightarrow$  1,3,9  $\rightarrow$  bit1=0  $\rightarrow$  DEFINITELY ABSENT
```

False-positive math. After inserting n items into m bits with k hashes, the probability a given bit is still 0 is $(1 - 1/m)^{kn} \approx e^{-kn/m}$. A false positive needs all k chosen bits set:

$$p_{fp} \approx (1 - e^{-kn/m})^k$$

Optimal $k = (m/n) \ln 2$. At that optimum, $p_{fp} = 2^{-k} = (0.6185)^{m/n}$. The headline number to memorize: **~9.6 bits per element gives ~1% FP; ~14.4 bits gives ~0.1%**. Each extra ~4.8 bits per element divides the FP rate by 10. Compare to storing the keys themselves (a 64-bit hash = 64 bits/element, a URL = hundreds of bits): a 1% Bloom is ~6× smaller than even just the hashes.

Back-of-envelope: 1 billion items at 1% FP $\rightarrow 1e9 \times 9.6 \text{ bits} \approx 1.2 \text{ GB}$. Exact set of 64-bit hashes $\rightarrow$ ~8 GB plus hash-table overhead (~16 GB). The Bloom fits in RAM; the exact set spills to disk.

Limitations & variants:

- **No deletion.** Clearing bits would create false negatives (a bit may be shared). Fix: **Counting Bloom filter** — replace bits with small counters (4 bits each, so 4× the space), increment/decrement. Used where churn matters.
- **Must size up front.** m and k are fixed at the FP rate you targeted for the planned n . Overfill and FP rate climbs. **Scalable Bloom filters** chain progressively larger filters.
- Not cache-friendly: k random memory probes per query = k cache misses. **Blocked Bloom filters** confine all k bits to one cache line (64 B), trading a hair of accuracy for ~ k × fewer cache misses — what high-QPS systems actually deploy.

Cuckoo filter

Stores short **fingerprints** (e.g. 8–16 bits) in a cuckoo hash table with two candidate buckets per item, using *partial-key cuckoo hashing* so an item can be relocated knowing only its fingerprint. **Supports deletion** and has better space efficiency than Bloom below ~3% FP.

	Bloom	Counting Bloom	Cuckoo
False negatives	Never	Never	Never
Delete supported	No	Yes	Yes
Space @ 1% FP	~9.6 bits/elem	~38 bits/elem	~12 bits/elem
Lookup cost	k random probes	k random probes	2 cache-local probes
Insert can fail	No	No	Yes (table near full → rebuild)
Use when	static set, read-heavy	need delete, space lax	need delete + low FP + locality

Real systems. Bloom filters guard LSM-tree reads: **RocksDB**, **Cassandra**, **HBase**, **LevelDB** keep a per-SSTable Bloom so a point lookup skips files that can't contain the key — turning an $O(\#levels)$ disk scan into one probe. **Bigtable** popularized this. **CDNs and Chrome's Safe Browsing** historically used Bloom-style filters so a malicious-URL check is a local memory hit, not a network call. **Medium** used Bloom filters to skip already-read posts. Cuckoo/quotient filters appear in newer storage engines and network routers (flow tracking).

2. Cardinality: HyperLogLog

Analogy. Flip coins and remember the longest run of heads you ever saw. If the longest run was 10, you probably flipped a *lot* of coins ($\approx 2^{10}$), because long runs are rare. You count “how many distinct things” without remembering any of them — just the rarest pattern you observed.

Mechanics

Hash each element to a uniform bit string. The bits behave like coin flips. The position of the **leftmost 1-bit** (i.e. number of leading zeros + 1) of any hash is geometrically distributed: seeing a value with p leading zeros suggests you've seen $\sim 2^p$ distinct items.

A single estimate has huge variance, so HLL uses **stochastic averaging**: use the first p bits of the hash to pick one of $m = 2^p$ registers; store the max p of the remaining bits in that register. Combine registers with a (harmonic-mean-based) estimator:

```
m=2^p registers, each holds max leading-zeros seen
hash(x) = 0110 1 0 0 0 1 ...
      ^p^  ^rest^
      |    ^
      |    |
      v    v
register 6  p = leading zeros of rest, store max
estimate ≈ α_m · m^2 / Σ 2^{register_j}    (harmonic mean form)
```

Error bound: standard error $\approx 1.04 / \sqrt{m}$. So $m = 16384$ registers ($p=14$) $\rightarrow$ **~0.81% error using 12 KB** (each register fits in 6 bits $\rightarrow 16384 \times 6/8 \approx 12$ KB). That 12 KB estimates cardinalities up to billions. Exact distinct-count of a billion items needs gigabytes; HLL needs **12 KB, constant, forever**.

The flat memory curve is the whole point: $O(\log \log n)$ — doubling the universe adds essentially nothing. The $\log \log$ is why it's called *HyperLogLog* (refinement of LogLog/Flajolet–Martin).

Practical refinements (HLL++): small-range correction via **linear counting** when most registers are empty; 64-bit hashes to avoid collisions past 2^{32} items; sparse representation that stores only touched registers for small n (a few hundred bytes) then switches to dense.

Mergeable: to union two HLLs, take the element-wise **max** of registers. This is why **BigQuery APPROX_COUNT_DISTINCT**, **Redshift**, **Presto/Athena**, **Druid** can count distinct across shards in parallel — each worker emits an HLL, the coordinator maxes them. **Redis PFADD / PFCOUNT / PFMERGE** is a 12 KB HLL per key (Redis caps it ~0.81% error). Cassandra and many analytics engines use it for `cardinality()`.

Pitfall: HLL gives unions cheaply but **intersections are derived via inclusion–exclusion** $|A \cap B| = |A| + |B| - |A \cup B|$, which subtracts two noisy numbers — error explodes when the intersection is small relative to the sets. Don't compute funnel overlaps from HLLs naively.

3. Frequency & heavy hitters: Count-Min Sketch and Top-K

Analogy. A grid of tally counters. Each item bumps one counter per row, chosen by a hash. To read an item's count, look at all its counters and take the *smallest* — because collisions can only inflate a counter, the minimum is the least-contaminated estimate.

Count-Min Sketch (CMS)

A 2-D array of counters, d rows $\times$ w columns, with d independent hash functions.

- **Add(x, c):** for each row i , `count[i][hi(x)] += c`.
- **Estimate(x):** `min over rows of count[i][hi(x)]`.

```
col→ 0   1   2   3   4
row0 (h0): 12  3  88  1  7
row1 (h1):  5 91  2 14  0    est(x) = min(88, 91, 7) = 7
row2 (h2):  0  7 33  9 60    (true count ≤ estimate, never below)
```

Error bound: with $w = \lceil e/\epsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$, the estimate overshoots the true count by at most $\epsilon \cdot N$ (N = total stream weight) with probability $1-\delta$. Example: $\epsilon = 0.001$, $\delta = 0.001 \rightarrow w \approx 2719$, $d \approx 7 \rightarrow$ ~19,000 counters $\approx$ 76 KB regardless of how many distinct items. Counts a stream of billions of events in tens of KB.

One-sided error: CMS **never underestimates** (the min still includes the item's own additions; collisions only add). Great for “is this hot?”, bad for “is this rare?” — low-frequency items are dominated by collision noise. **Count-Mean-Min** subtracts estimated noise per row to debias.

Top-K / heavy hitters

CMS tells you a *given* item's frequency, but not *which* items are hot. Combine CMS with a **min-heap of size K**: on each update, estimate the item's count and, if it beats the heap minimum, insert/update it. Result: approximate top-K with bounded memory. **Redis** ships `CMS.*` and `TOPK.*` (the latter uses the **HeavyKeeper** algorithm, which probabilistically decays losers so the heap tracks true heavy hitters better than CMS+heap). The classic exact-ish alternative is **Space-Saving / Misra-Gries** (deterministic counters with eviction), used in stream processors.

Need	Structure	Error direction	Memory
Distinct count	HyperLogLog	$\pm 1.04/\sqrt{m}$	12 KB
Is item present?	Bloom/Cuckoo	FP only	~10 bits/elem
Item frequency	Count-Min	over-count only	~tens of KB
Which items are hot?	Top-K / Space-Saving	approximate set	K counters
Percentiles	t-digest / DDSketch	bounded rel. error	~few KB

Real systems. CMS underpins network telemetry (per-flow byte counts in switches), DDoS detection, ad-frequency capping, and database query optimizers' join-size estimates. **Heavy hitters** drive “trending now,” rate-limiting the noisiest API keys, and cache admission (admit only items the sketch says are hot — see **TinyLFU/W-TinyLFU**, the admission policy in Caffeine/RocksDB caches, built on a CMS with aging).

4. Percentiles: t-digest and quantile sketches

The trap: you cannot average percentiles. The mean of per-host p99 is **not** the fleet p99. Storing every latency to sort it is $O(n)$ and won't merge across hosts. You need a **mergeable quantile sketch**.

t-digest (Ted Dunning) clusters sorted values into centroids whose size is *small near the tails and large in the middle*, so p50 is coarse but p99/p999 stay precise — exactly where SLOs care. It's mergeable: combine two digests by merging centroid lists. Memory ~a few KB for thousands of clusters; relative error at the tails is tiny.

DDSketch (Datadog) guarantees **relative-error** quantiles (e.g. answer within $\pm 2\%$ of the true value) with a fully mergeable, bucketed design — preferred when you need a hard accuracy *contract* rather than t-digest's empirically-good-at-tails behavior.

	t-digest	DDSketch	Naive (store all + sort)
Accuracy guarantee	empirical, great at tails	hard relative-error bound	exact
Mergeable	Yes	Yes	only by concatenation
Memory	~KB	~KB (depends on range)	$O(n)$
Use when	metrics/SLO dashboards	strict accuracy SLA	small n, offline

Real systems. Elasticsearch `percentiles` agg uses t-digest; Datadog built and uses DDSketch fleet-wide; Prometheus historically used fixed-bucket histograms (cheap, mergeable, but bucket-bounded error) and now offers native histograms. See `13-observability...md` for how this feeds SLOs.

5. Anti-entropy & sync: Merkle trees

Analogy. Two people each hold a 10,000-page book and want to find which pages differ without reading all 10,000 aloud. They hash each chapter; compare chapter hashes; only descend into chapters that differ; recurse to the page. Mismatches are found in $\log n$ comparisons.

A **Merkle tree** is a binary tree where leaves are hashes of data blocks and each internal node is `hash(left || right)`. The root is a single fingerprint of the whole dataset.

```
      root = H(H12 || H34)
      /      \
  H12=H(H1||H2)  H34=H(H3||H4)
  /  \          /  \
H1=H(d1) H2=H(d2) H3=H(d3) H4=H(d4)
```

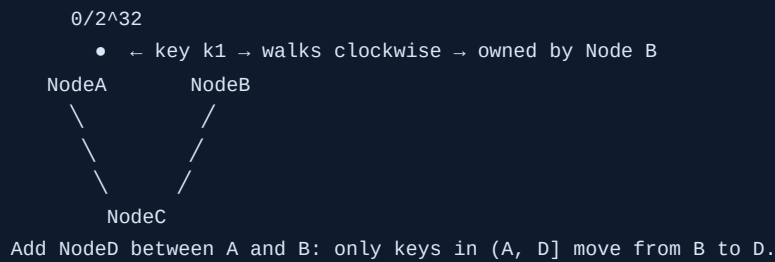
To reconcile two replicas: exchange roots. Equal $\rightarrow$ identical, done in one round trip. Differ $\rightarrow$ recurse only into mismatching subtrees. Cost to find k differing blocks $\approx O(k \log n)$ hashes exchanged instead of shipping all n .

Real systems. DynamoDB and Cassandra use Merkle trees for **anti-entropy repair** between replicas (Cassandra's `nodetool repair`) — the practical answer to the eventual-consistency divergence discussed in `09-cap-pace1c...md`. Git commits/trees, IPFS/blockchains, ZFS scrubbing, and rsync-style block sync all rest on the same Merkle/rolling-hash idea. War story: Cassandra repairs can over-stream if token ranges and tree granularity are misaligned, hashing whole partitions for a one-row diff — tune `repair` ranges.

6. Partitioning: consistent hashing (derived)

Problem with naive `hash(key) mod N`: add or remove one node ($N \rightarrow N \pm 1$) and *almost every* key remaps. With 1M keys and $N=10 \rightarrow 11$, ~91% of keys move — a cache stampede or a full data reshuffle. See `10-partitioning-and-sharding.md`.

Consistent hashing maps both keys *and* nodes onto the same circular hash space `[0, 232)`. A key is owned by the **first node clockwise** from its position.



Key property: adding/removing a node remaps only $\sim K/N$ keys (the arc that node owns), not all of them.

Math: expected fraction of keys moved on a node change is $1/N$.

Virtual nodes (vnodes): one physical node placed once on the ring gives lumpy, unbalanced arcs (load variance up to $\sim O(\log N)$ worse). Place each physical node at **V random positions** (e.g. $V=100-256$). Load standard deviation shrinks $\sim 1/\sqrt{V}$; with $V=100$ you get $\sim 10\%$ imbalance, with $V=256$ a few percent. Vnodes also make **heterogeneous capacity** trivial (a $2\times$ box gets $2\times$ vnodes) and spread a failed node's load across *many* survivors instead of dumping it all on one neighbor.

Scheme	Keys moved on node change	Balance	Use when
hash mod N	$\sim$ all	perfect (static)	N never changes
Consistent hashing (1 point/node)	$\sim K/N$	lumpy	small clusters, simple
+ virtual nodes	$\sim K/N$	even ($\propto 1/\sqrt{V}$)	real distributed stores
Rendezvous (HRW) hashing	$\sim K/N$	even, no ring state	clients need agreement w/o shared ring

Real systems. **Amazon Dynamo** (the paper) introduced consistent hashing + vnodes to the field; **Cassandra**, **Riak**, **ScyllaDB** use it for token assignment; **memcached clients (ketama)** and **Envoy/NGINX upstream hashing** use it for sticky load balancing; **Discord** and CDNs use **rendezvous (Highest-Random-Weight) hashing** when they want each client to independently agree on placement without distributing ring state. Bounded-load consistent hashing (Google/Vimeo) caps any node at $(1+\epsilon) \times \text{average}$.

7. Spatial: geohash, quadtrees, S2

Problem: “find restaurants within 2 km of me” over millions of points. A B-tree on (lat, lng) can't do 2-D range efficiently; you need to map 2-D $\rightarrow$ 1-D so a normal index/range scan works, while keeping nearby points nearby in the 1-D order.

- **Geohash:** interleave bits of lat and lng and base32-encode $\rightarrow$ a string where **shared prefix $\approx$ spatial proximity**. `gbsuv` is near `gbsuw`. A prefix range query = a box. Cheap, string-friendly, indexes in any DB. Used by **Redis GEO commands** (`GEOADD` / `GEOSEARCH`), internally a sorted set scored by 52-bit

geohash) and **Elasticsearch geo**. Pitfall: the **edge problem** — two points across a cell boundary can be physically adjacent but share no prefix, so neighbor queries must check the 8 surrounding cells.

- **Quadtree**: recursively subdivide space into 4 quadrants until each cell holds $\leq$ capacity points. Adapts to density: dense cities subdivide deep, oceans stay coarse. Great for dynamic point sets and range/kNN. Used in game engines, mapping, collision detection.
- **S2 (Google)**: projects the sphere onto a cube, then uses a **Hilbert space-filling curve** to linearize each face into 64-bit cell IDs. Hilbert curves preserve locality far better than geohash's Z-order (fewer big jumps), and S2 handles spherical geometry correctly (no pole/datetime distortion). A region = a small set of S2 cell-ID ranges → indexable as integer ranges. **Uber's H3** is the hex-grid cousin (uniform cell adjacency, used for surge/demand). S2 powers Google Maps, MongoDB geo, and many ride-hailing/geofencing stacks.

	Geohash	Quadtree	S2 / H3
Curve / structure	Z-order (Morton) string	tree	Hilbert curve / hex grid
Locality preserved	OK (Z-order jumps)	good	best (Hilbert)
Dynamic density	fixed precision	adaptive	fixed levels
Edge handling	manual neighbor check	natural	range covering
Use when	quick prefix index in any DB	in-memory dynamic kNN	planet-scale, correct geometry

8. Skip lists

Analogy. An express subway over a local line: extra tracks that skip many stops, so you ride the express then drop to the local near your destination — $O(\log n)$ instead of walking every station.

A skip list is a sorted linked list with **probabilistic express lanes**: each node is promoted to the next level up with probability p (usually 0.5), giving $\sim \log n$ levels. Search starts top-left, moves right until the next node overshoots, then drops a level — expected $O(\log n)$ search/insert/delete with high probability, *without* the rotation bookkeeping of a balanced tree.

```
L3: H _____, 9 →, NIL
L2: H _____, 5 _____, 9 →, NIL
L1: H →, 3 →, 5 →, 7 →, 9 →, NIL
L0: H →1→3→4→5→6→7→8→9→, NIL (full sorted list)
search 7: L3 overshoot-drop, L2 at5 overshoot-drop, L1 5→7 found
```

Why systems pick it over a balanced tree: simpler concurrent implementation (no global rebalancing → fine-grained/lock-free insert), and good cache behavior. **Redis sorted sets (ZSET)** use a skip list + hash map (so **ZRANGEBYSCORE** is range-scan friendly while **ZSCORE** is $O(1)$). **LevelDB/RocksDB memtables**, **Apache Lucene**, and **Java's ConcurrentSkipListMap** use skip lists. Probabilistic balance means a pathological RNG could degrade it, but with $p=0.5$ the chance of being far from $\log n$ is astronomically small.

9. Common pitfalls / war stories

- **Wrong error direction.** Treating a Bloom filter “yes” as authoritative and skipping the real lookup → serving stale/wrong data on false positives. Bloom answers are a *fast path*, always backed by the source of truth.
- **Resizing a Bloom filter under load.** Sizing for `n=1M` then pushing 10M → FP rate silently climbs toward 100% and the filter stops filtering. Monitor fill ratio; use scalable Bloom or rebuild.
- **HLL intersection math.** Computing user-overlap between two segments via inclusion–exclusion when the overlap is tiny → error larger than the answer. Keep exact small sets, or use MinHash for similarity.
- **Averaging percentiles.** Dashboards that `avg(p99)` across hosts report a number that is *neither* the fleet p99 nor meaningful. Aggregate t-digests/histograms, not the percentiles.
- **CMS for rare items.** Using Count-Min to detect *infrequent* events → noise swamps signal because error scales with total stream weight $\epsilon \cdot N$. CMS is for heavy hitters only.
- **Consistent hashing without vnodes.** A 5-node cluster with one ring point each gets 40/10/25/15/10 load splits; the hot node tips over. Always vnode.
- **Geohash edge blindness.** “Nearest store” misses the store 50 m away across a cell boundary. Always query neighbor cells.
- **Non-idempotent sketch merges.** Double-counting the same shard’s HLL/CMS into a rollup (HLL max is idempotent; CMS add is **not** — re-adding inflates counts). Know which merge op is idempotent.
- **Hash quality.** All of these assume *uniform* hashing. A weak hash (or attacker-chosen keys) clusters items, breaking every bound at once. Use murmur/xxhash/SipHash (SipHash for adversarial inputs).

Case Study: Reddit’s unique-pageview counts with HyperLogLog

The problem. Reddit wanted to show a “unique views” number on every one of its millions of posts (the “x views” counter under a submission). At Reddit’s scale this is brutal: hundreds of millions of pageviews per day, tens of millions of distinct posts, and a long tail where popular posts get *millions* of viewers while most posts get a handful. The naive “exact” approach is to keep, per post, the set of every viewer who has seen it (by user id or a hashed visitor id) and report `len(set)`. With ~300M+ events/day fanning into millions of posts, those per-post sets collectively run to **billions of stored ids** — far too much memory to keep hot, and the write path (SADD into a giant per-post set on every pageview) would hammer the datastore. Crucially, the product requirement is a *display* counter: nobody cares whether a post shows 1,200,000 or 1,206,300 views. **Exactness here is unnecessary** — exactly the situation §0 describes.

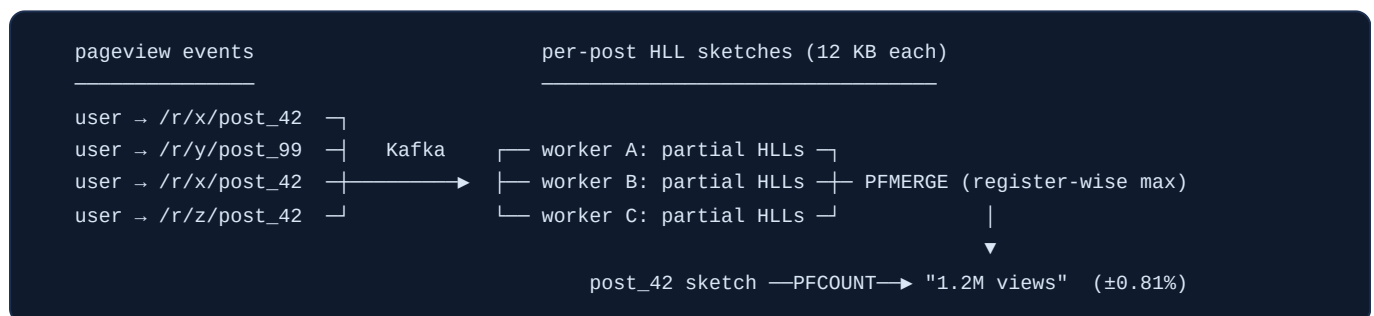
Applying the module. This is the textbook **cardinality estimation** problem from §2, so Reddit reached for **HyperLogLog**. Each post gets one HLL sketch keyed by `post_id`. On a pageview they **PFADD** the viewer’s id into that post’s sketch; to render the counter they **PFCOUNT** it. The set of “who viewed” is never

materialized — only the **lossy fingerprint** of the viewer stream is kept, which is the core probabilistic move from §0: *stop storing elements; store a sketch of the stream*.

The numbers come straight from the §2 mechanics. A Redis-style dense HLL is **~12 KB** per key (`m = 16384` registers at 6 bits each) with a **standard error** $\approx 1.04/\sqrt{m} \approx 0.81\%$. So a post with a million true viewers reports within roughly $\pm 8,000$ — invisible on a rounded “1.2M views” badge. That 12 KB is **constant regardless of whether the post has 10 viewers or 10 million** — the flat `O(log log n)` memory curve that makes HLL viable. Compare to the exact set: a million 64-bit viewer ids is ~8 MB *for one hot post*; HLL is ~700× smaller, and the saving compounds across millions of posts.

Reddit’s real pipeline added two refinements the module calls out:

1. **Mergeability across a streaming pipeline.** Pageview events flow through Kafka into a stream processor. Rather than mutate one Redis key on the hot path per event, workers build **partial HLLs per time-window/per-partition**, and these are combined with the **register-wise max** union (`PFMERGE`) — the idempotent merge from §2. This is what lets the count be computed in parallel across shards and rolled up without coordinating on a single counter.
2. **HLL++ small-range correction.** Most posts are *not* viral; they have tens of viewers. Raw HLL is biased for small `n`, so the **linear-counting / sparse-representation** path from §2 (“HLL++”) keeps the long tail of low-traffic posts accurate and cheap (a few hundred bytes) before switching to the dense 12 KB form only for the posts that actually get big.



The trade-off they accepted. They gave up **exactness and the ability to answer per-viewer questions** in exchange for bounded memory and cheap, mergeable writes. The HLL can tell you *roughly how many* distinct viewers a post had; it can **never** tell you *whether a specific user* viewed it, nor the precise count — and as §2 warns, you can’t cheaply intersect two HLLs (“how many people saw both post A and post B?”) without inclusion–exclusion blowing up when the overlap is small. For a public vanity counter those are non-needs, so the trade is almost free. The accuracy they bought (~0.8%) is far tighter than the display rounding, and the memory they saved is the difference between “fits in Redis” and “doesn’t fit anywhere hot.”

Results. Per-post memory dropped from $O(\text{viewers})$ to a flat **~12 KB**, with error well under 1% — orders of magnitude less RAM than exact sets while keeping the counter accurate enough that no human notices the slack. The write path became a single fixed-cost `PFADD` /merge instead of an ever-growing set insert, so it stays $O(1)$ in space and time even as a post goes viral. The same HLL pattern underlies `APPROX_COUNT_DISTINCT` in BigQuery/Redshift/Presto and `PFCOUNT` in Redis (§2) — Reddit’s case is just the canonical product-facing instance of it.

Lessons

- **Match the structure to the actual requirement, not the literal ask.** “Count unique viewers” *sounds* like it needs a set; the product only needs a rounded number, so a 12 KB sketch beats an 8 MB set with no user-visible loss. Always ask §0’s question: *is exactness actually required here?*
- **Constant memory regardless of scale is the headline feature.** HLL’s $O(\log \log n)$ curve means a 10-viewer post and a 10-million-viewer post cost the same — that flatness is what makes it deployable across millions of keys.
- **Pick structures whose merge op is idempotent for streaming rollups.** Register-wise `max` lets you fan out across Kafka partitions and recombine safely; re-merging the same partition can’t double-count (unlike a Count-Min `add`).
- **Know what the sketch can’t do.** No per-user membership, no cheap intersection — if you later need “did user U see post P?” or accurate overlap, HLL is the wrong tool and you must reach for a Bloom filter or MinHash instead.

10. Test yourself

1. You need to count daily unique visitors across 200 web servers and report a single global number with <1% error. Which structure, how much memory per server, and what’s the merge operation? (*Hint: HLL, 12 KB, register-wise max.*)
2. A Bloom filter sized for 1% FP at 10M items is now holding 30M. What happened to the FP rate and why? (*Hint: $p_{fp} \approx (1 - e^{-kn/m})^k$; n tripled $\rightarrow$ FP soars.*)
3. Why can Count-Min answer “is X a heavy hitter?” but not “is X rare?” Which way does its error point? (*Hint: collisions only add; over-count bounded by $\epsilon \cdot N$.*)
4. Your service computes fleet p99 as `mean(per_host_p99)`. Why is this wrong, and what should you store instead? (*Hint: percentiles don’t average; merge t-digests/histograms.*)
5. Adding the 11th node to a 10-node cluster, how many keys move under `hash mod N` vs consistent hashing with vnodes? (*Hint: ~91% vs $\sim 1/11 \approx 9\%$.*)
6. Two restaurants are 30 m apart but their nearest-neighbor geohash query returns only one. What’s the bug and the fix? (*Hint: cell-boundary edge problem; query the 8 neighbors.*)
7. Why does Redis use a skip list (not a red-black tree) for sorted sets? (*Hint: range scans + simpler concurrent/insert code, same $O(\log n)$.*)
8. Cassandra replicas have diverged after a network partition. What structure finds the differing rows with minimal data transfer, and what’s the cost to locate `k` diffs? (*Hint: Merkle tree, $O(k \log n)$.*)

11. Further reading

- **DDIA** (Kleppmann), *Designing Data-Intensive Applications* — Ch. 5 (Merkle/anti-entropy in replication), Ch. 6 (consistent hashing & partitioning).
- Bloom, “Space/Time Trade-offs in Hash Coding with Allowable Errors” (1970); Fan et al., “**Cuckoo Filter: Practically Better Than Bloom**” (2014).

- Flajolet et al., “**HyperLogLog**” (2007); Heule et al., “**HyperLogLog in Practice**” (Google, 2013, the HLL++ paper).
- Cormode & Muthukrishnan, “**Count-Min Sketch**” (2005); Metwally et al., “**Space-Saving / Efficient Computation of Frequent and Top-k**” (2005); Yang et al., “**HeavyKeeper**” (2018).
- Dunning & Ertl, “**Computing Extremely Accurate Quantiles Using t-digests**”; Masson et al., “**DDSketch**” (Datadog, VLDB 2019).
- Karger et al., “**Consistent Hashing and Random Trees**” (1997); **Amazon Dynamo** paper (2007, vnodes + Merkle).
- **Google S2 Geometry** docs; **Uber H3** docs; Pugh, “**Skip Lists: A Probabilistic Alternative to Balanced Trees**” (1990).
- Official docs: **Redis** (HyperLogLog, GEO, CMS/TOPK, ZSET internals), **RocksDB** Bloom-filter wiki, **BigQuery** `APPROX_*` functions, **Elasticsearch** percentiles aggregation.

See also: `09-cap-pacelc...md` (why anti-entropy exists), `10-partitioning-and-sharding.md` (consistent hashing in context), `13-observability...md` (quantile sketches in metrics).

16 — Rate Limiting, Resiliency & Fault Tolerance

Source: [readme/research/system-design/16-rate-limiting-and-resiliency.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains two things every online system needs: a way to control how much traffic comes in, and a way to stay standing when something inside the system breaks. Without these, one slow database or one misbehaving customer can bring down the whole service for everyone. Understanding these ideas helps you build software that handles problems gracefully instead of collapsing.

The main parts explained simply:

- **Rate limiting (traffic control)** — A “bouncer” that decides how many requests a customer or server can make in a given time. Without it, one bad actor can flood your system and slow it down for everyone else. Especially important when you have many store owners sharing the same platform.
- **Fixed window counter** — The simplest traffic counter: count requests per minute, reset at each minute boundary. Easy to build but has a flaw — a customer can sneak in double the allowed amount right at the boundary between two windows.
- **Sliding window counter** — A smarter version that looks at the recent past more smoothly, avoiding the double-admission problem. Uses very little memory and is accurate enough for most real systems. This is what most professional API services use.
- **Token bucket** — Imagine each customer gets a bucket of tokens that refills steadily. Each request uses one token. When the bucket is empty, requests are refused. Allows short bursts of activity (like a human clicking fast) then settles back to a steady rate. Used by Stripe and AWS.
- **Leaky bucket** — Like token bucket but the output is always a steady drip, no bursts allowed. Useful when the thing receiving the traffic cannot handle sudden spikes at all.
- **Timeouts** — Every call to another service must have a deadline. If there is no deadline and the other service hangs, your system keeps waiting forever, threads pile up, and everything crashes. Setting a sensible wait limit is the most basic protection you have.
- **Retries with jitter** — When a request fails, you try again — but not all at the same moment. Adding a random delay (jitter) before each retry spreads the load out. Without this, all clients retry at the exact same second, making the problem worse instead of better.
- **Circuit breaker** — Like a fuse in your house. When a dependency (database, payment service, etc.) is clearly broken, the circuit breaker “trips open” and stops all calls to it instantly, returning a safe fallback answer. After a short cooling period, it tests the water before resuming normal calls. This stops one broken dependency from dragging down everything else.
- **Bulkhead** — Named after the watertight walls in a ship. Each dependency gets its own small pool of resources. If one dependency hangs and uses up its pool, the rest of the system keeps working with their own pools. One tenant’s traffic spike cannot starve other tenants.
- **Graceful degradation** — When something breaks, show a reduced but still useful experience rather than a blank error page. For example, if the recommendations service is down, show bestsellers instead. The customer can still buy; they just get less personalisation.
- **Load shedding and backpressure** — When the system is already overwhelmed, the right move is to deliberately refuse some requests (lower-priority ones first) so the important requests still get

through. Backpressure means telling the caller to slow down rather than silently dropping work.

- **Cascading failures** — A small problem (one database 100ms slower than usual) can spiral into a full outage because slow calls block threads, blocked threads pile up, and the whole system tips over. Circuit breakers and timeouts exist to cut this chain before it runs away.
- **Retry storms** — If everyone retries at the same time after a problem, the retries themselves become the problem. The system can get stuck in a broken state even after the original cause is gone. Retry budgets and jitter prevent this.
- **Thundering herd** — Many clients hit the same resource at the exact same moment (for example, a popular cache key expires and a thousand users all rush to the database to rebuild it). Solutions include only letting one request rebuild the cache while the rest wait, or refreshing the cache slightly before it expires.
- **Chaos engineering** — Deliberately breaking things in a controlled way (killing a server, cutting a connection) to prove your safety nets actually work before a real incident does it for you.

What to do with this:

Set an explicit timeout on every external call your code makes — this alone prevents the most common cause of full outages. And add a circuit breaker around any dependency that is not under your direct control, so one slow third-party service cannot take down your whole storefront.

What you'll learn: How to throttle traffic precisely (five rate-limiting algorithms, with pseudo-code and the distributed-Redis variants), and how to keep a system standing when its dependencies misbehave — timeouts, retries with jitter and budgets, circuit breakers, bulkheads, load shedding, backpressure, and graceful degradation. You'll also learn to *think in failures*: cascading collapse, retry storms, thundering herds, and how chaos engineering and blast-radius reduction turn “hope” into engineering.

Prerequisites: Read [09-cap-pacelc...md](#) (consistency/availability trade-offs underpin “fail open vs fail closed”), [11-load-balancing...md](#) (where limiters and load shedders live), and [13-caching...md](#) (caches are both a degradation tool and a thundering-herd victim). Familiarity with queues ([12-message-queues...md](#)) helps for backpressure.

1. Why this module exists

Most outages are not caused by a single component dying. They're caused by a small failure that *propagates* — a slow database makes web servers pile up requests, threads exhaust, health checks fail, the load balancer pulls nodes out, the survivors get more load, and the whole tier falls over. The failure was 100ms of extra latency; the outage was the *response* to it.

Rate limiting and resiliency are the two halves of controlling that propagation:

- **Rate limiting** controls load coming *in* (protect yourself from clients and from yourself).
- **Resiliency patterns** control how you behave when something *downstream* breaks (protect yourself from your dependencies).

Both are fundamentally about one idea: **bounded resources, explicitly managed, with a defined behavior at the boundary**. Everything below is a variation on that.

2. Rate limiting

2.1 Intuition

A rate limiter is a bouncer at a club door. The club holds N people. The bouncer decides who gets in *now*, who waits, and who gets turned away. The interesting engineering is entirely in the *policy*: do you count per minute? Do you allow a burst then throttle? Do you smooth everyone to a steady trickle? Each algorithm answers differently.

We rate-limit for: protecting backends from overload, enforcing fairness across tenants (very relevant to a multi-tenant SaaS like this one), monetizing API tiers, and stopping abuse (credential-stuffing, scraping).

2.2 The five algorithms

Fixed Window [--60s--][--60s--] count per bucket	Sliding Log keep every timestamp exact, $O(n)$ memory	Sliding Counter weight prev window by overlap fraction
Token Bucket tokens refill at rate burst up to capacity	Leaky Bucket fixed-rate drain smooths to constant out	

Fixed window counter

Divide time into fixed buckets (e.g. each calendar minute). Keep one counter per bucket.

```
def allow(key, limit, window):
    bucket = key + ":" + floor(now() / window)
    count = INCR(bucket) # atomic
    if count == 1: EXPIRE(bucket, window)
    return count <= limit
```

Dead simple, $O(1)$ memory, one counter. **The flaw: boundary bursting.** A limit of 100/min allows 100 at 00:59 and 100 at 01:00 — 200 requests in a 2-second span across the boundary. You've nominally enforced 100/min but actually allowed $2\times$ the rate over a real window.

Sliding window log

Store the timestamp of every request in a sorted set. To decide, drop everything older than `now - window` and count what remains.

```
def allow(key, limit, window):
    now = now_ms()
    ZREMRANGEBYSCORE(key, 0, now - window) # evict old
    count = ZCARD(key)
    if count < limit:
        ZADD(key, now, unique_id); EXPIRE(key, window)
        return True
    return False
```

Perfectly accurate — no boundary problem. **Cost:** $O(n)$ memory per key (one entry per request in the window). A client doing 10k req/min costs you 10k stored timestamps. Fine for low-limit auth endpoints; ruinous for high-throughput APIs.

Sliding window counter (the practical sweet spot)

Approximate the sliding window using two fixed-window counters, weighting the previous window by how much it still overlaps:

```
estimate = curr_count + prev_count * (1 - elapsed_in_curr / window)
allow if estimate < limit
```

If you're 25% into the current minute, you count 75% of last minute's requests plus all of this minute's. $O(1)$ memory, no hard boundary spike, small approximation error (Cloudflare reported <0.003% error rate in production using exactly this). This is what most production API gateways use.

Token bucket (allows controlled bursts)

A bucket holds up to **capacity** tokens; tokens refill at **rate** per second. Each request consumes one token; if the bucket is empty, reject (or queue).

```
def allow(key, rate, capacity):
    now = now()
    tokens, last = GET(key) # lazy refill - no background timer
    tokens = min(capacity, tokens + (now - last) * rate)
    if tokens >= 1:
        SET(key, tokens - 1, now)
        return True
    SET(key, tokens, now)
    return False
```

The genius is **lazy refill**: you don't run a timer adding tokens; you compute how many *would* have accrued since the last request. Token bucket allows bursts up to **capacity** then settles to **rate** — which matches real traffic (clients are bursty). This is what AWS API Gateway, Stripe, and most cloud APIs use. Stripe famously documented their token-bucket limiter.

Leaky bucket (smooths output to a constant rate)

Requests enter a FIFO queue; they drain at a *fixed* rate. Overflow is dropped. Where token bucket limits the *input* rate but permits bursts to pass through, leaky bucket guarantees a *constant output* — useful when the thing you protect cannot tolerate bursts at all (e.g. a legacy mainframe, a payment processor with a hard TPS cap).

Token bucket: bursts pass through, then throttle → protects rate, allows spikes
Leaky bucket: output is always smooth, queue absorbs → protects the downstream's TPS

2.3 Comparison

Algorithm	Memory	Accuracy	Allows bursts?	When to use
Fixed window	O(1)	Poor (2× boundary)	Accidentally	Crude internal throttle, you don't care about edges
Sliding log	O(n) per key	Exact	No	Low-volume sensitive endpoints (login, password reset)
Sliding counter	O(1)	~exact (<0.01% err)	No	General-purpose API gateway — the default choice
Token bucket	O(1)	Exact	Yes, up to capacity	Public APIs where bursts are normal (Stripe, AWS)
Leaky bucket	O(queue)	Exact	No — smooths	Protecting a fixed-TPS downstream

2.4 Distributed rate limiting

A single counter in app memory breaks the moment you run >1 instance: 10 servers each allowing 100/min = 1000/min actual. You need **shared state**. Options:

1. **Centralized store (Redis)**. Every node does the INCR/ZADD against one Redis. Correct and simple. Cost: a network round-trip per request (+0.2–1ms intra-region) and Redis becomes a hotspot/SPOF. Mitigate with Redis Cluster sharded by key, and a Lua script to make the read-modify-write **atomic** (otherwise two nodes race between GET and SET):

```
-- token bucket as one atomic Redis call; no race window
local tokens = tonumber(redis.call('HGET', KEYS[1], 'tokens') or ARGV[2])
local last   = tonumber(redis.call('HGET', KEYS[1], 'ts') or ARGV[4])
local now, rate, cap = tonumber(ARGV[4]), tonumber(ARGV[1]), tonumber(ARGV[2])
tokens = math.min(cap, tokens + (now - last) * rate)
if tokens >= 1 then
    redis.call('HSET', KEYS[1], 'tokens', tokens - 1, 'ts', now)
    redis.call('EXPIRE', KEYS[1], ARGV[3]); return 1
end
return 0
```

2. **Local + async sync (sloppy counters)**. Each node keeps a local count and periodically reconciles with a central store. Sub-millisecond, no per-request round-trip, but temporarily over-admits during the sync window. This is the “**approximate but cheap**” trade — what large-scale gateways (e.g. envoy's global rate limiting, or per-node buckets reconciled centrally) use when the round-trip cost is unacceptable.
3. **Sticky routing**. Hash the client to one node so its counter is local. Fragile — rebalancing on scale events loses state.

Approach	Latency cost	Accuracy	SPOF risk	Use when
Centralized Redis + Lua	+0.5–1ms/req	Exact	Yes (mitigate w/ cluster)	Correctness matters, moderate QPS
Local + async reconcile	~0	Approximate (over-admits)	No	Ultra-high QPS, slight over-admit OK
Sticky routing	~0	Exact-ish	Per-node	Rare; avoid for elastic fleets

Always tell the client. Return `429 Too Many Requests` with `Retry-After` and `X-RateLimit-Remaining` / `-Reset` headers. A limiter that rejects silently just turns into a retry storm (§4). For a multi-tenant store like this codebase, key by tenant *and* by endpoint class so one noisy tenant can't starve others — that's bulkheading (§3.4) applied to rate limits.

3. Resiliency patterns

Rate limiting protects you from *callers*. These patterns protect you from *calleees* — the dependencies that go slow or down.

3.1 Timeouts — the foundation everything else stands on

A request with no timeout is a resource leak waiting for a bad day.

The default timeout in most HTTP clients is “infinite” or absurdly long (Go's default `http.Client`: no timeout; many connection pools: 30–60s). When a downstream hangs, every caller thread blocks until... never. Threads/connections are finite; they exhaust; new requests can't even get a thread; you're down — caused by a dependency that was merely *slow*, not dead.

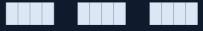
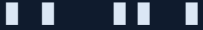
Set timeouts from your **latency distribution, not averages**. If p99 of the call is 200ms, a 2s timeout gives 10× headroom while still bounding the blast. Rule: a service's own timeout budget must be *larger* than the sum of its downstream timeouts and retries, or the outer caller gives up while you're still trying (wasted work). Propagate **deadlines**, not durations — gRPC does this: the original caller sets “this whole tree must finish by T”, and every hop subtracts elapsed time. Without deadline propagation, retries deep in the call tree multiply latency.

3.2 Retries — and the trap inside them

Retries fix *transient* failures (a dropped packet, a brief GC pause, a node restart). They make *persistent* failures catastrophically worse. The math: if a service is overloaded and every client retries 3×, you've **tripled** the load on an already-dying service. This is the **retry storm** — retries are the accelerant in most cascading failures.

Rules for safe retries:

1. **Only retry idempotent operations** (GET, PUT, DELETE) — or use an idempotency key so a retried POST doesn't double-charge a card. (Directly relevant to payment flows: a retried charge without an idempotency key is a duplicate charge.)
2. **Exponential backoff** — wait `base * 2^attempt`, not a fixed interval.
3. **Add jitter**. Without it, all clients that failed at the same instant retry at the same instant — a synchronized wave. AWS's canonical post "Exponential Backoff and Jitter" showed *full jitter* (`sleep = random(0, base * 2^attempt)`) dramatically reduces contention vs naked backoff.
4. **Retry budgets / circuit at the client**. Cap retries to e.g. 10% of total requests (a *token bucket on retries*). When the budget is exhausted, fail fast. Google's SRE book and Envoy both implement retry budgets precisely because per-request retry caps still allow storms.
5. **Don't retry at every layer**. If the DB client retries 3×, the service retries 3×, and the gateway retries 3×, one user request becomes 27 backend calls. Retry at **one** layer, ideally the outermost that can do something useful.

Bad: backoff only  (all clients synchronized)
 Good: backoff + jitter  (spread out, less contention)

3.3 Circuit breaker — stop knocking on a dead door

If a dependency is down, retrying is pointless and harmful. A circuit breaker tracks the failure rate and, once it crosses a threshold, **trips open** — failing fast (or falling back) *without even attempting the call* for a cooldown. Then it tests the water before resuming.



- **Closed**: normal; count failures over a rolling window.
- **Open**: all calls fail instantly (latency → ~0, you stop hammering the sick service, giving it room to recover).
- **Half-open**: after cooldown, allow a *probe* request. Success → close; failure → re-open.

Netflix Hystrix popularized this; modern equivalents: resilience4j (JVM), Polly (.NET), Envoy/Istio outlier detection (at the mesh layer). Key tuning: threshold (e.g. >50% errors over 20+ requests — never trip on a single failure or tiny sample), and cooldown (5–30s). The circuit breaker is the single most important pattern for stopping **cascading failure**, because it converts “slow, blocking, thread-exhausting calls” into “instant failures” the moment a dependency is unhealthy.

3.4 Bulkheads — isolate so one leak doesn't sink the ship

Named after a ship's watertight compartments: flood one, the others keep you afloat. In software, partition resources so one failing dependency can't consume *all* of them.

Concretely: instead of one shared thread pool of 200 for all downstream calls, give the flaky recommendation service its own pool of 20. When recommendations hangs, it exhausts *its* 20 threads — the other 180 keep serving checkout. Without bulkheads, the slow recommendation call eats every thread and takes down the whole app.

Apply per-tenant (one tenant's traffic spike can't starve others — critical for this multi-tenant codebase), per-dependency, or per-endpoint-class. Implementations: separate connection pools, separate thread pools, separate Kubernetes deployments, or even separate cells (cell-based architecture — AWS's approach to blast-radius reduction).

3.5 Fallbacks & graceful degradation

When a call fails (or the breaker is open), what do you *return*? A fallback is the degraded-but-useful answer:

- Cached/stale data ("last known price").
- A default ("recommendations unavailable" → show bestsellers).
- A queued write ("we'll process this shortly") instead of a synchronous one.
- Partial response — render the page without the one broken widget.

Graceful degradation is the system-level version: shed *features*, not the whole service. Amazon's product page is built so that if reviews, recommendations, and "frequently bought together" all fail, you can *still buy the item*. The buy button is the core path; everything else is optional and degrades independently. Decide, per dependency, **fail-open vs fail-closed** (see `09-cap-pacelc...md`): a rate limiter whose Redis is down — do you allow all traffic (fail open, risk overload) or block all (fail closed, risk outage)? A fraud check — fail closed (don't approve when you can't verify). There's no universal answer; choose deliberately and document it.

3.6 Load shedding & backpressure

When you're *already* overloaded, accepting more work makes everything slower and serves *no one* (a queue of 10,000 requests each timing out = 0 successful + 100% CPU). **Load shedding** = deliberately rejecting some requests (429/503) to keep the rest healthy. Shed by priority: drop health-check-able batch/analytics traffic before paying-customer checkout traffic. Better than CPU-based shedding is **queue-latency-based** (used by Facebook/Meta): if the time a request spends *waiting in queue* exceeds a threshold, shed it — that directly measures "am I falling behind?"

Backpressure is the upstream-facing twin: instead of silently dropping, signal the producer to *slow down*. TCP flow control, HTTP/2 flow control, reactive streams (`request(n)`), and bounded queues all implement backpressure. A bounded queue that blocks (or rejects) when full *is* backpressure — it's why you should **never use an unbounded queue** between a fast producer and a slow consumer (unbounded queue = OOM crash deferred to later, plus latency that grows without limit). See `12-message-queues...md`.

Pattern	Protects against	Mechanism	Real systems
Timeout	Hung dependencies	Bound wait time	Every HTTP client; gRPC deadlines
Retry + jitter	Transient blips	Backoff w/ randomness	AWS SDK, Envoy
Retry budget	Retry storms	Token bucket on retries	Google SRE, Envoy
Circuit breaker	Cascading failure	Trip open on high error rate	Hystrix, resilience4j, Istio
Bulkhead	Resource monopoly	Partitioned pools	Hystrix, cell architecture
Fallback	User-facing errors	Degraded answer	Amazon, Netflix
Load shedding	Self-overload	Reject excess by priority	Meta (queue latency)
Backpressure	Producer overrun	Signal slow-down	TCP, HTTP/2, Reactive Streams

4. Failure thinking

4.1 Cascading failures

The canonical sequence: a small perturbation (DB slows 100ms) → callers' threads block longer → thread pools fill → queues grow → latency spikes → health checks time out → load balancer ejects "unhealthy" nodes → survivors get *more* load → they fall over → the LB ejects them too → total outage. The trigger was trivial; the **positive feedback loop** was fatal. Circuit breakers + timeouts + load shedding + bulkheads exist specifically to break these loops.

4.2 Retry storms (metastable failures)

A subtle and dangerous class: the system can get *stuck* in a bad state even after the original trigger is gone. Example: a brief overload causes timeouts; clients retry; retries keep the service overloaded; even after the original spike passes, the retry-induced load *sustains* the overload indefinitely. This is a **metastable failure** (a 2021 research area, HotOS paper "Metastable Failures in Distributed Systems"). Recovery often requires *manual* intervention — drop the retry load, shed aggressively, or even briefly take traffic to zero — because the system won't self-heal. The fix is preventive: retry budgets, jitter, and circuit breakers that *remove* the retry pressure.

4.3 Thundering herd

Many clients wake up and hit the same resource at the same instant:

- **Cache stampede:** a hot cache key expires; 1000 concurrent requests all miss and all hit the DB to recompute the same value. Fixes: (a) *request coalescing / single-flight* — only the first miss recomputes, the rest wait for it; (b) probabilistic early expiration (XFetch) — recompute *before* expiry with rising probability; © locks. See [13-caching...md](#).
- **Synchronized retries** (§4.2) — same root cause, fixed by jitter.

- **Cron / restart herds** — every node's `@daily` job fires at 00:00:00, or a deploy restarts all nodes at once and they all reconnect/warm caches simultaneously. Stagger with jitter.

4.4 Blast-radius reduction

Assume failures *will* happen; minimize how much they take down. Techniques: **cell-based architecture** (partition users into independent cells; a bug or overload affects one cell, not all users), **shuffle sharding** (AWS — assign each customer a random subset of nodes so any two customers rarely share the *same* full set; one customer's "poison" request affects few others), **regional isolation**, and **canary/staged rollouts** (a bad deploy hits 1% before 100%).

4.5 Chaos engineering

You cannot trust resilience you haven't tested under failure. Chaos engineering deliberately injects failures in production-like (or production) environments to verify the system degrades as designed. Netflix's Chaos Monkey (kills random instances), Gremlin, and AWS Fault Injection Service. The discipline: form a **hypothesis** ("if Redis dies, the limiter fails open and latency rises <50ms"), inject the fault on a small blast radius, measure, and fix the surprises. The point isn't breaking things — it's discovering the implicit assumptions ("we assumed the cache was always there") *before* an incident does.

5. Common pitfalls / war stories

- **No timeout = the silent killer.** The single most common root cause of "the whole app went down because one dependency got slow." Audit every outbound call for an explicit, distribution-based timeout.
- **Retrying non-idempotent writes.** A payment retried without an idempotency key = double charge. Customers notice. (See this repo's payment-correctness notes — retried charges are exactly the failure mode to guard.)
- **Retries at every layer.** $3 \times 3 \times 3 = 27$. Pick one layer.
- **Backoff without jitter.** You "added resiliency" and created a synchronized retry wave that's *worse* than no retry.
- **Circuit breaker that trips on one failure / tiny sample.** Now a single blip opens the circuit and you've created an outage from a hiccup. Require a minimum request volume in the window.
- **Unbounded queue as "backpressure."** It's not backpressure; it's deferred OOM with unbounded latency. Bound it.
- **Rate limiter as a SPOF.** If every request must hit one Redis and Redis is down, you must have *already decided* fail-open vs fail-closed — discovering it during the incident is too late.
- **Fixed-window limiter sold as a guarantee.** "100/min" that actually allows 200 at the boundary. Use sliding counter.
- **Load shedding by CPU instead of queue latency.** CPU is a lagging, noisy signal; queue wait time directly measures "am I behind."
- **Health checks that are too strict** (eject nodes during a transient blip → fewer nodes → more load → eject more → cascade). Make health checks tolerant; distinguish "degraded" from "dead."

🌱 Case Study: Stripe rate limiting + Netflix Hystrix circuit breaking

Two of the most-cited engineering stories in this space map almost one-to-one onto the patterns above. Stripe is the canonical “distributed token bucket” story; Netflix is the canonical “circuit breaker + bulkhead” story. Together they cover the whole module.

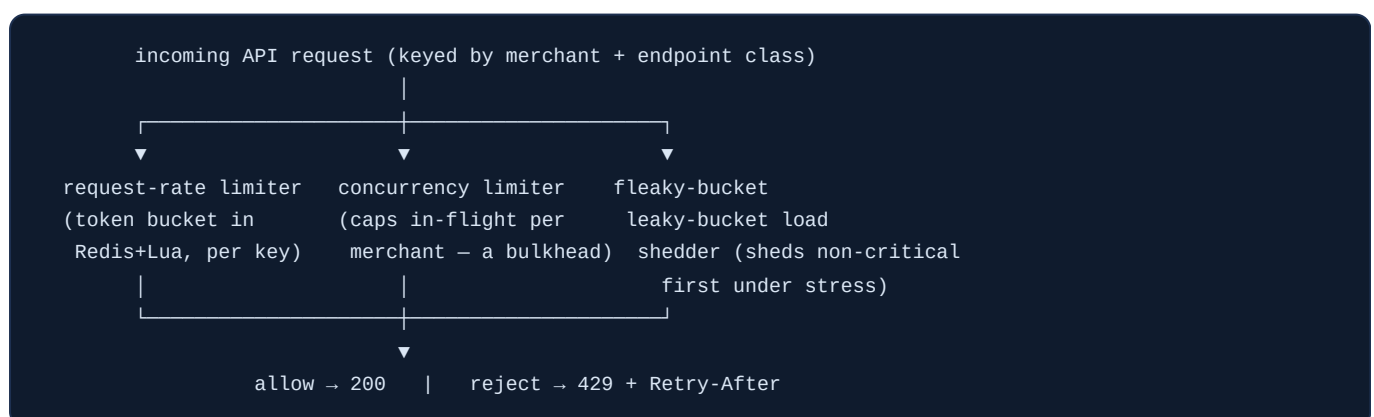
The problem

Stripe runs a payments API where a single bad actor — or one customer’s runaway integration with a buggy retry loop — can flood the API with thousands of requests per second. At Stripe’s scale (millions of API calls per day, billions of dollars in payment volume, every request hitting a shared Postgres/Redis tier), one tenant’s burst must never degrade everyone else’s checkout. They also can’t just drop requests blindly: a rejected `POST /charges` that the client retries without care becomes a *double charge*. So the limiter has to be precise, multi-tenant-fair, distributed across their whole fleet, and paired with idempotency.

Netflix (circa 2011–2012) ran ~1B+ API calls/day through their Edge/API tier, fanning out to dozens of backend services (recommendations, bookmarks, ratings, A/B config, etc.) per device request. A single Netflix home screen touched many dependencies. The math is brutal: at 99.99% per-dependency availability across 30 dependencies, you still see ~0.3% of requests touch a failing dependency — millions of failures/day. One slow service (not even *down*, just slow) would block threads on the API tier, exhaust the pool, and take down the *entire* streaming UI. They needed one slow backend to never sink the ship.

How the module’s concepts were applied

Stripe → **token bucket across N servers via Redis (§2.2, §2.4)**. Stripe documented their limiter as a **token bucket** implemented in **Redis with a Lua script** — exactly the “centralized store + atomic read-modify-write” design from §2.4. Each request runs the bucket math (lazy refill, decrement) as one atomic Redis call so two app servers can’t race between GET and SET. They actually run several limiter *types* layered:



Keying by **merchant + endpoint class** is the bulkhead-applied-to-rate-limits idea from §2.4/§3.4: a noisy merchant exhausts *their* bucket, not the shared backend. The concurrency limiter caps simultaneous in-flight requests per merchant — a bulkhead (§3.4) expressed as a limit. And because retried `POST` s are

dangerous (§3.2 rule 1), Stripe pairs all of this with **idempotency keys**, so a client that retries a 429'd or timed-out charge never double-charges.

Netflix → **circuit breaker + bulkhead + fallback (§3.3–3.5)**. Hystrix wrapped *every* outbound dependency call in a command object that enforced four of this module's patterns at once:

- a **timeout** (§3.1) on every call, set from the dependency's latency distribution;
- a **circuit breaker** (§3.3) with the exact CLOSED → OPEN → HALF-OPEN states from the diagram above — it tripped when the rolling error rate crossed a threshold over a minimum request volume (never on a single failure, the §5 pitfall), failed fast while OPEN, and let a single probe through in HALF-OPEN;
- a **bulkhead** (§3.4) — each dependency got its *own* small thread pool, so a hanging recommendations service exhausted only its ~10–20 threads, never the threads serving playback;
- a **fallback** (§3.5) — when the breaker was open or the call failed, return degraded-but-useful data (a generic non-personalized list instead of personalized recommendations), so the page still rendered.

```
[CLOSED] —errors > 50% over N reqs—> [OPEN] —5s—> [HALF-OPEN]
  ▲ calls real service          | instant fallback | 1 probe
  | (own bulkhead thread pool)  | (no thread held) |
  └──────────────────────────┬──────────────────────────┘
                             probe ok
```

This is precisely the §4.1 cascade broken at three points: the timeout bounds the wait, the bulkhead bounds the resource blast, and the breaker converts “slow blocking call” into “instant fallback” the moment the dependency is unhealthy.

The key trade-off they accepted

Stripe accepted a **Redis round-trip and a Redis SPOF on every API call** (~sub-millisecond, but real) in exchange for *exact, fleet-wide* limiting — they chose the “Centralized Redis + Lua” row of the §2.4 table over the cheaper “local sloppy counter” row, because for payments, over-admitting (charging past a limit, hammering a downstream processor) is worse than a tiny latency cost. They mitigate the SPOF with Redis redundancy and a deliberate fail-open/fail-closed decision per limiter type (§3.5).

Netflix accepted **degraded UX and the overhead of a separate thread pool per dependency** (thread-context-switch cost, more total threads) in exchange for *isolation* — a worse-but-working home screen instead of an all-or-nothing one. They gave up “always perfectly personalized” to get “never fully down.”

Real results

- Stripe's published numbers describe the limiter adding only sub-millisecond overhead while protecting the API; the four-layer design (rate + concurrency + load-shed) kept abusive traffic from affecting well-behaved merchants.
- Hystrix let Netflix survive individual dependency outages with **no visible user impact** beyond reduced personalization, and Netflix open-sourced it precisely because the pattern generalized — it became the template for resilience4j, Polly, and Istio outlier detection (§3.3). It also forced **chaos testing** (§4.5): Chaos Monkey killed instances to *prove* the breakers and fallbacks actually worked.

Lessons

- **A rate limiter is only as good as its keying.** Stripe's win wasn't the algorithm (token bucket is textbook) — it was keying per *merchant* + *endpoint class* so limiting doubled as bulkheading. Decide what a “tenant” is before you pick an algorithm.
- **Wrap dependencies, not requests.** Netflix's leverage came from making *every* outbound call carry a timeout + breaker + bulkhead + fallback by construction, so resilience was the default, not something each engineer remembered to add.
- **Pick your trade-off explicitly and early.** Both teams *chose* — Stripe: latency/SPOF for exactness; Netflix: degraded UX for isolation. The \$5 pitfall is discovering you never decided (fail-open vs fail-closed, over-admit vs round-trip) during the incident.
- **Resilience you haven't broken on purpose is a guess.** Netflix only trusted Hystrix because Chaos Monkey continuously proved the fallbacks fired.

6. Test yourself

1. **Why does a fixed-window limiter allow up to 2× the configured rate, and which algorithm fixes it most cheaply?** (*Hint: bursts straddling the window boundary; sliding-window counter, $O(1)$.*)
 2. **You run 20 app servers and need a global 1000 req/s limit. Name two designs and their trade-offs.** (*Hint: centralized Redis+Lua (accurate, +round-trip, SPOF) vs local sloppy counters (fast, over-admits).*)
 3. **A downstream service is healthy but slow (p99 = 5s, you have no timeout). Walk through how this becomes a full outage.** (*Hint: thread/connection exhaustion → cascade.*)
 4. **Why is “backoff without jitter” sometimes worse than no retry at all?** (*Hint: synchronized retry waves; thundering herd on the recovering service.*)
 5. **Explain the three circuit-breaker states and what a request experiences in each.** (*Hint: closed=normal, open=instant fail/fallback, half-open=single probe.*)
 6. **What is a metastable failure, and why can't the system always self-heal from one?** (*Hint: retry-sustained overload persists after the trigger; positive feedback loop needs external load removal.*)
 7. **Your Redis-backed limiter loses Redis. Fail open or fail closed — and how would the answer differ for a fraud-check vs a public read API?** (*Hint: read API → fail open for availability; fraud → fail closed for safety; per-dependency decision.*)
 8. **Why should you never put an unbounded queue between a fast producer and slow consumer, and what's the correct alternative?** (*Hint: OOM + unbounded latency; bounded queue that applies backpressure/sheds.*)
-

7. Further reading

- **DDIA** (Kleppmann), Ch. 1 (“Reliability”) and Ch. 8 (“The Trouble with Distributed Systems”) — failure thinking foundations.

- **Google SRE Book** — “Handling Overload,” “Addressing Cascading Failures” (the definitive treatment of retry budgets, load shedding, and the cascade loop).
 - **Release It!** (Michael Nygard) — origin of the Circuit Breaker and Bulkhead *as named patterns*; “Stability Patterns/Antipatterns.”
 - **AWS Builders’ Library** — “Exponential Backoff and Jitter,” “Timeouts, retries, and backoff with jitter,” “Workload isolation using shuffle-sharding.”
 - **“Metastable Failures in Distributed Systems”** (Bronson et al., HotOS 2021) — the sustained-feedback failure class.
 - **Stripe Engineering** — “Scaling your API with rate limiters” (token bucket in production).
 - **Cloudflare blog** — “How we built rate limiting capable of scaling to millions of domains” (sliding window counter, error analysis).
 - **Netflix Hystrix wiki** and **resilience4j docs** — circuit breaker / bulkhead implementation details.
 - **Principles of Chaos Engineering** (principlesofchaos.org) — the hypothesis-driven discipline.
-

Sibling modules: [09-cap-pace1c...md](#) (fail-open vs fail-closed), [11-load-balancing...md](#) (where limiters/shedders live), [12-message-queues...md](#) (bounded queues & backpressure), [13-caching...md](#) (cache stampede / thundering herd).

17 · Observability, SRE & Operating Systems at Scale

Source: [readme/research/system-design/17-observability-and-operations.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains how software teams know when their systems are healthy, how they find problems when something goes wrong, and how they release updates without breaking things for customers. It matters because a system that nobody can see inside is impossible to fix quickly — and slow fixes mean unhappy customers and lost sales.

The main parts explained simply:

- **Monitoring vs. Observability** — Monitoring answers questions you already thought to ask (“Is the server slow?”). Observability means you can answer questions you never thought of (“Why does checkout fail only on Android, only for one store, only since 3pm?”) without needing to write new code to investigate. Observability is what you want; monitoring alone leaves you blind to surprises.
- **Metrics, Logs, and Traces (the three pillars)** — These are three different types of data your system collects. Metrics are simple numbers over time (how many orders per minute, how many errors). Logs are detailed records of exactly what happened in one moment. Traces follow a single customer’s request all the way through every part of the system to show exactly where it got slow or broke. You need all three to fully understand problems.
- **RED and USE — what to measure** — RED tells you what customers experience: how many requests per second, how many failed, and how long they took. USE tells you how your servers are coping: how busy they are, how much work is piling up, and whether errors are occurring. Together they give you the full picture without drowning in useless data.
- **Distributed Tracing** — When a customer clicks “Place Order,” that action touches many different services (pricing, payment, inventory). A trace stitches all those steps together into one timeline so you can see exactly which step was slow. Without it, finding the slow step is like searching for one slow runner in a relay race without knowing who dropped the baton.
- **Cardinality — the silent danger** — Cardinality means how many unique combinations your tracking system has to store. If you accidentally track data like individual customer IDs in your metrics system, the number of combinations explodes into the millions and can crash the very tool you rely on to spot problems. Keep metrics simple and broad; save the detailed per-customer data for logs.
- **SLI, SLO, and SLA — making reliability concrete** — An SLI is a measured score (“99.94% of checkouts succeeded this week”). An SLO is your internal target (“we aim for 99.9%”). An SLA is the promise to customers in a contract (“we guarantee 99.5%”). Your internal target is always stricter than the customer promise, so you have room to fix things before customers are affected.
- **Error Budgets — reliability as a currency** — If your goal is 99.9% uptime, you are allowed 0.1% failures — about 43 minutes of problems per month. That “allowance” is your error budget. While budget remains, your team can ship new features and take risks. When the budget runs out, you stop shipping features and only fix reliability until things recover. This stops the endless fight between “ship fast” and “don’t break things.”
- **Alerting on symptoms, not causes** — Alert your team when customers are actually hurting (“checkout success rate dropped”), not when a server metric looks unusual (“CPU is at 80%”). A busy CPU can be completely harmless. Too many false alarms train your team to ignore alerts — then they miss the real emergency.

- **On-call and Runbooks** — When a page (alert) wakes someone up at 3am, a runbook is the step-by-step guide that tells them what to do. Every alert must have one. After any serious problem, the team does a blameless postmortem — they write down what happened and how to prevent it, without punishing anyone, so the system gets better over time.
- **Deployment Strategies — releasing without breaking things** — Rolling updates replace servers one batch at a time. Blue-green keeps the old version running while the new one warms up, then flips the switch instantly. Canary releases send just 1–5% of customers to the new version first, watch for problems, then gradually increase. Feature flags let you ship code that is “off” by default and switch it on later with no new deployment — and switch it off instantly if something goes wrong.
- **Health Checks — liveness vs. readiness** — A liveness check asks “is this server completely stuck?” and restarts it if so. A readiness check asks “is this server ready to take customers right now?” and just removes it from the queue if not. Mixing these up can cause all your servers to restart at the same time during a small database hiccup, turning a minor blip into a full outage.

What to do with this: Set clear reliability targets (SLOs) in plain numbers and build alerts that fire when customers are actually affected — not when server stats look unusual. Make sure every alert has a written playbook so the person woken up at night knows exactly what to do.

What you’ll learn: How to know what a distributed system is doing when you can’t attach a debugger — the three pillars (metrics, logs, traces), how to define what “working” means with SLIs/SLOs/error budgets, how to alert on user pain instead of CPU graphs, and how to ship changes (canary, blue-green, feature flags) and recover (runbooks, on-call) without taking the system down. You’ll leave able to reason about cardinality, sampling, and the failure modes that turn a monitoring stack into a second outage.

Prerequisites: Read [01-fundamentals-and-estimation.md](#) (latency numbers, percentiles), [09-cap-pacelc-and-consistency.md](#) (why partial failure is normal), and [15-microservices-and-decomposition.md](#) (a request now crosses 10 services, which is *why* you need tracing).

1. Monitoring vs. Observability — the core distinction

Monitoring answers *known* questions: “Is CPU above 80%? Is the error rate above 1%?” You decide the questions in advance and build dashboards/alerts for them.

Observability is the property that lets you answer questions you *didn’t* anticipate — “Why are checkouts failing only for tenant `acme`, only on Android, only since 14:03?” — *without shipping new code*. The bar: any novel slice of behaviour should be reconstructable from data you already emit.

Analogy. Monitoring is the dashboard warning lights in your car (oil, temp — predefined). Observability is being able to put the engine on a dyno and probe any sensor for a fault nobody pre-wired a light for. A control system is *observable* (Kalman’s original term) if its internal state can be inferred from its outputs. Same idea: emit rich enough outputs that internal state is recoverable.

The practical consequence: high-cardinality, structured, contextual data beats pre-aggregated counters. A counter `http_errors_total` tells you *something* broke; a structured event with `{tenant, route,`

`status, user_agent, build_sha, region}` tells you *what*.

2. The three pillars: metrics, logs, traces

	+-----+ METRICS +-----+	+-----+ LOGS +-----+	+-----+ TRACES +-----+
question	"is it bad?" (aggregate health)	"what exactly happened on this one event?"	"where in the request did it go wrong?"
shape	numbers over time	timestamped lines	spans w/ parent
cost	cheap, bounded (if cardinality controlled)	cheap to write, pricey to store/ index	expensive, usually sampled
cardinality	LOW (must be)	HIGH (free-form)	HIGH (per request)

Pillar	Best for	Worst for	Real systems
Metrics	Trends, SLOs, alerting, dashboards, capacity	Explaining a single weird request	Prometheus, VictoriaMetrics, Datadog, StatsD
Logs	Forensic detail on one event, audit trails, errors with stack traces	Aggregation at scale, "what's the p99?"	Loki, ELK/OpenSearch, Splunk
Traces	Latency breakdown across services, finding the slow hop	Long-term trends (you sampled most away)	Jaeger, Tempo, Zipkin, AWS X-Ray

These are converging. **OpenTelemetry (OTel)** is the vendor-neutral standard (CNCF, merger of OpenTracing + OpenCensus) that emits all three with one SDK and *correlates* them: a metric anomaly links to exemplar traces, a trace links to the logs of its spans via shared `trace_id`. The end state is **wide structured events** (Honeycomb's model) — one fat event per unit of work with dozens of dimensions — from which metrics and traces are derived views.

Structured logging — non-negotiable

Don't log `"user 42 failed checkout"`. Log:

```
{
  "ts": "2026-06-16T14:03:11Z",
  "level": "error",
  "msg": "checkout_failed",
  "trace_id": "a1b2...",
  "tenant": "acme",
  "user_id": 42,
  "order_total": 129.50,
  "gateway": "stripe",
  "error_code": "card_declined",
  "build": "sha-9f3c"
}
```

Now you can `count by error_code`, `filter tenant=acme`, and jump to the trace. Free-text logs force regex archaeology. Always inject the `trace_id` so logs ↔ traces correlate.

Eyeballing the waterfall, the 140ms DB query inside pricing is the bottleneck — invisible to metrics, painful to find in logs, obvious in a trace.

Context propagation

The magic is passing `trace_id` + `span_id` across process/network boundaries. The standard is **W3C Trace Context**: HTTP headers `traceparent` (`version-traceid-spanid-flags`) and `tracestate` . Every service reads the incoming header, creates a child span, and forwards an updated header downstream. For async/queues (Kafka, SQS), you propagate the same headers as **message attributes** — miss this and your trace breaks at every queue boundary (a very common bug).

Sampling — you cannot trace everything

At 100k req/s, full tracing is petabytes/day. Two strategies:

Strategy	When the decision is made	Pros	Cons
Head-based (e.g. sample 1%)	At ingress, before you know the outcome	Cheap, simple, decided once and propagated via the <code>sampled</code> flag	Drops 99% of traces — including the rare errors you most want
Tail-based	After the trace completes, at a collector	Keep 100% of errors + slow traces, sample the boring fast ones	Needs a buffering collector holding spans until trace ends; memory + complexity

The pragmatic answer (OTel Collector tail sampling, Jaeger adaptive): **head-sample the happy path at ~1%, tail-keep all errors and all traces over your SLO latency**. The “sampled” bit must propagate consistently or you get partial traces (some services log the trace, others don’t).

5. Cardinality — the silent killer

Cardinality = number of unique time series = product of all label-value combinations. In Prometheus, *every unique label combination is a separate stored series* with its own memory and disk.

```
http_requests_total{method, status, route, ...}

method: 5 values
status: 6 values
route: 20 values
    → 5 × 6 × 20 = 600 series    ✓ fine

now add label user_id (1,000,000 users):
    → 600 × 1,000,000 = 600,000,000 series    ✨ OOM
```

This is the **cardinality explosion**, and it has killed countless Prometheus instances (a label with `email` , `request_id` , `timestamp` , `full URL with query string` , or a UUID is the usual culprit). Each series

costs ~1–3 KB of RAM resident; tens of millions of series exhausts memory and the TSDB falls over — your *monitoring* causes an outage during the incident you needed it for.

Rules:

- Metric labels must be **bounded and low-cardinality**: method, status class, route *template* (`/orders/:id` , never `/orders/8675309`), region, tenant-tier (not tenant-id if you have millions).
- High-cardinality data (`user_id` , `request_id` , `trace_id` , exact URL) belongs in **logs and traces**, not metric labels. This is the cleanest division of labour between the pillars.
- Watch out for **unbounded values sneaking into labels**: error messages with embedded IDs, raw paths, customer-supplied input. Validate/template before they become a label.

For Print-Flow-360 specifically: a `tenant_id` label across thousands of stores will blow up Prometheus. Use `tenant_tier` or top-N tenants + an `other` bucket as a label; keep per-tenant detail in logs/traces queried on demand.

6. SLI / SLO / SLA & error budgets

This is the spine of SRE — it turns “is it up?” into an engineering contract.

- **SLI** (Indicator) — a *measured* number, expressed as a ratio of good events to total: `good_requests / total_requests` . E.g. “fraction of checkout requests served <300ms with 2xx/3xx.”
- **SLO** (Objective) — the *internal target* for the SLI: “99.9% of checkouts succeed <300ms over 28 days.”
- **SLA** (Agreement) — the *external, contractual* promise with financial penalties, always looser than the SLO (you alert internally well before you breach a customer contract).

SLI = what you measure

→ 99.94% this week

SLI = your internal target

→ 99.9% (you have headroom)

SLA = customer contract

→ 99.5% (refunds if breached)

SLA < SLO < actual SLI

← the healthy ordering

The “nines” — know the budget in minutes

Availability	Downtime / 30 days	Downtime / year
99%	~7.2 hours	~3.65 days
99.9% (“three nines”)	~43 min	~8.76 hours
99.95%	~21.6 min	~4.38 hours
99.99% (“four nines”)	~4.3 min	~52.6 min
99.999% (“five nines”)	~26 sec	~5.26 min

Five nines means your *entire annual* downtime budget is ~5 minutes — less than a single deploy gone wrong. Each nine is roughly 10× harder and more expensive; pick the cheapest SLO users actually need.

Most products don't need more than 99.9%.

Error budget — the killer feature

`error budget = 1 - SLO` . At 99.9%, you're *allowed* 0.1% failures = ~43 min/month of badness. This budget is a currency:

- **Budget remaining** → **ship fast**, take risks, do that scary migration.
- **Budget exhausted** → **freeze features**, only reliability work ships until the budget recovers.

This dissolves the dev-vs-ops war: reliability isn't "max uptime at all costs," it's "stay within budget." 100% is the wrong target — it forbids all change and costs infinitely. Google's SRE book is built on this. **Burn-rate alerting** is the modern form: alert when you're consuming budget too fast (e.g. "at this rate the 28-day budget is gone in 6 hours") with multi-window (fast 5m + slow 1h) to be both responsive and resistant to flapping.

7. Alerting — on symptoms, not causes

The cardinal rule: **alert on user-visible symptoms (SLO violations), not on internal causes.**

"High CPU" is *not* an alert-worthy symptom — a box can run at 95% CPU all day serving users perfectly. "Checkout success rate dropped below 99.9%" *is*. CPU is a *diagnostic* you consult *after* the symptom page fires.

Anti-pattern alert	Why it's bad	Symptom-based replacement
CPU > 80%	Often harmless; pages at 3am for nothing	Latency p99 > SLO threshold
Disk 70% full	Not urgent (yet); a <i>ticket</i> , not a <i>page</i>	Disk will fill in <4h (predictive)
"Service X restarted"	Restarts can be normal (autoscaling)	Error rate up <i>for users</i>
One node down	Redundancy should absorb it	Capacity/quorum at risk

Every page must be actionable, urgent, and real. If a human can't do something *now*, it's a ticket or a dashboard, not a page. The failure mode here is **alert fatigue**: when 90% of pages are noise, on-call learns to ignore them — and sleeps through the real one. Track your alert → action ratio; ruthlessly delete alerts nobody acts on.

Tune for **precision vs. recall**: too sensitive → false pages → fatigue; too lax → missed outages. SLO burn-rate alerts (above) are the principled way to balance both.

8. On-call & runbooks

A page is a contract with a human who's about to lose sleep. Discipline:

- **Runbooks:** every alert links to a runbook — “what this means, how to diagnose, how to mitigate, how to escalate.” A page with no runbook is a research project at 3am. Runbooks should bias toward **mitigation first** (roll back, shed load, failover) and root-cause later.
- **Sustainable load:** cap pages/shift (Google’s guidance: ≤ 2 incidents per 12h shift). More than that means you’re under-investing in reliability and burning out humans.
- **Blameless postmortems:** every significant incident gets a written postmortem focused on *systemic* causes and action items, not who fumbled. Blame → people hide incidents → you stop learning. The output is a prioritized list of fixes, tracked to completion.
- **Error budget governs the postmortem teeth:** repeated budget burn forces the freeze, which forces the fixes to actually get prioritized over features.

9. Deployment strategies — changing a running plane’s engine

Most outages are *caused by changes*. Deploy strategy is how you de-risk change.

ROLLING

v1 v1 v1 v1 → v2 v1 v1 v1 → v2 v2 v1 v1 → v2 v2 v2 v2
(replace instances batch by batch; both versions live briefly)

BLUE-GREEN

[BLUE v1] ← 100% traffic [GREEN v2] idle, warmed
flip router → [GREEN v2] ← 100% [BLUE v1] kept for instant rollback

CANARY

v1 95%
v2 5% → watch SLIs → 25% → 50% → 100% (or roll back)

Strategy	Rollback speed	Extra cost	Blast radius of a bad deploy	Schema/state risk	Use when
Rolling	Slow (roll forward batch by batch)	None (in-place)	Grows as rollout proceeds	Both versions must coexist	Default k8s deploy; stateless services
Blue-green	Instant (flip back)	2× infra during cutover	All-or-nothing per flip	DB must serve both; hard with migrations	You need instant rollback & can pay 2× briefly
Canary	Fast (route 0% to canary)	~1 extra version's worth	Limited to the canary %	Both versions coexist	High-traffic services where you can measure SLIs on the slice
Feature flags	Instant (toggle, no deploy)	Negligible	Per-flag, per-cohort	Decouples deploy from release	Risky features, gradual exposure, A/B, kill-switch

Feature flags decouple deploy from release — code ships dark, then you turn it on for 1% → 100%, and *kill it instantly* without a redeploy when it misbehaves. This is exactly Print-Flow-360’s `config/pdf_service.php` “all flags off by default” pattern (per [CLAUDE.md](#)): the new Node.js path is

deployed but dormant, flipped per-feature, instantly revertable. The cost is **flag debt** — stale flags become dead conditional spaghetti; budget time to remove them.

The hard part is always state. Code rolls back in seconds; a database migration does not. The discipline is **expand/contract (parallel-change)**: (1) add new column/table — backward compatible; (2) deploy code that writes both old+new; (3) backfill; (4) switch reads to new; (5) stop writing old; (6) drop old — *later*. Never make a deploy and its schema change mutually dependent, or you can't roll back the code without a destructive DB revert. (This mirrors the API backward-compat rule in [CLAUDE.md](#): add new field alongside old, migrate consumers, remove later.)

10. Health checks: liveness vs. readiness vs. startup

A subtle distinction that, gotten wrong, *amplifies* outages.

Probe	Question	If it fails...	Gotcha
Liveness	"Is the process wedged/deadlocked?"	Restart the container	Must be <i>shallow</i> — don't check the DB here
Readiness	"Can this instance serve traffic <i>right now</i> ?"	Remove from load balancer (don't kill)	Check deps that gate serving (pool warmed, caches primed)
Startup	"Has a slow-booting app finished initializing?"	Hold off liveness until done	Prevents killing apps that boot slowly

```
request → LB → [ready? yes → route] [ready? no → skip, try another]
kubelet → liveness fail → kill+restart pod
```

The classic cascading-failure trap: you put a DB ping in your *liveness* probe. The DB has a transient blip → liveness fails on *every* instance simultaneously → Kubernetes restarts *all* of them at once → thundering-herd of cold instances hammer the recovering DB → total outage. The DB hiccup should have failed **readiness** (drop from LB, keep the process alive, recover when DB returns). **Rule: liveness checks the process; readiness checks dependencies.** Never let a shared dependency fail all your liveness probes in lockstep.

11. Common pitfalls / war stories

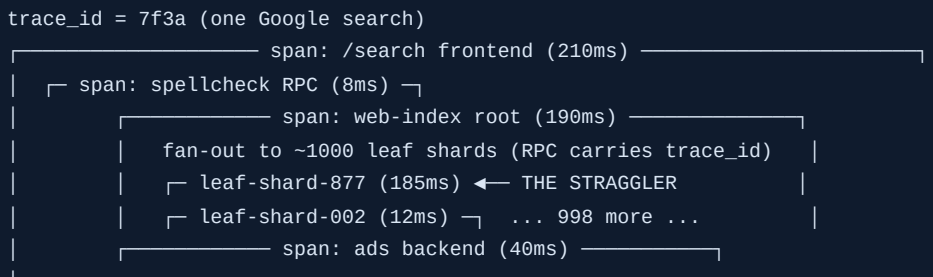
- Averages hide the tail.** "Avg latency 40ms" while p99 is 3s and 1% of users rage-quit. Always histograms. Worse: averaging percentiles across hosts is mathematically meaningless — aggregate the histograms, then compute the percentile.
- Cardinality bomb during an incident.** Someone adds `user_id` (or a raw URL, or an error string with an ID) as a Prometheus label "just to debug." Series count explodes, Prometheus OOMs, and you lose *all* monitoring mid-incident. High cardinality → logs/traces, never metric labels.

- **The trace that ends at the queue.** Tracing works beautifully until a request hits Kafka/SQS and nobody propagated `traceparent` in message headers — the trace silently truncates and the slow async consumer is invisible. Propagate context across *every* boundary, sync and async.
- **Liveness checks a dependency → synchronized mass-restart.** (See §10.) The single most damaging health-check mistake.
- **Alert fatigue.** 200 alerts, 5 meaningful. On-call mutes the channel; the one real page is missed. Delete noisy alerts aggressively; measure actionability.
- **100% uptime as the goal.** Forbids all deploys, costs infinitely, and there's no budget to take the risks that let you move fast. Pick the SLO users need and *spend* the error budget.
- **Head-sampling away your errors.** 1% head sampling keeps 1% of *errors* too — the exact traces you need are gone. Tail-sample to keep all errors/slow traces.
- **Logging PII / secrets in structured logs.** Now they're in your log index, replicated, retained, and discoverable. Scrub at the emission layer (relevant to the plaintext-CVV findings noted in this project's memory).
- **Dashboards as the alerting strategy.** Nobody's staring at Grafana at 3am. Dashboards are for *diagnosis after a page*; SLO alerts do the waking.

✱ Case Study: Google Dapper + the SRE SLO/error-budget model

The problem. By the mid-2000s a single Google web search no longer hit one server — it fanned out to *thousands* of internal services (spell-check, ads, universal-search backends, ranking shards). When a search came back slow, an engineer staring at metrics could see “search p99 regressed” but had *no way* to know **which** of the thousands of downstream calls caused it. Logs from individual services were uncorrelated — there was no thread tying one user's request together across machines. At Google scale (billions of queries/day, tens of thousands of machines, requests fanning out 1:1000+), the classic “ssh in and read the log” approach was dead. They needed observability across a system no human could hold in their head.

Dapper: trace context propagation, applied. Dapper (the system Jaeger, Zipkin and OpenTelemetry trace later copied) is the **distributed tracing** pillar from §2 and §4 made real. Every request gets a `trace_id` at ingress. Each unit of work is a **span** with a `span_id` and `parent_span_id`, exactly the span tree from §4. The crucial trick is **context propagation** (§4): the trace context rides along inside Google's RPC framework, so *every* RPC automatically carries the trace IDs to the next hop — this is the W3C-Trace-Context idea before the standard existed. Because it was baked into the shared RPC library, instrumentation was nearly free per-team: you got tracing without touching app code.



The waterfall makes the one slow leaf shard (out of ~1000) instantly visible — the **tail-latency amplification** problem from §3 ($1 - p^N$), now *diagnosable* because the trace stitches the fan-out back together.

Sampling — the key trade-off. Tracing every one of billions of requests would cost more storage and network than the service being traced. Dapper accepted a hard trade-off: aggressive **head-based sampling** (§4). In practice it traced a *tiny* fraction of requests (on high-traffic services, well under 0.1%; early versions sampled ~1/1024). They **gave up completeness to get affordability** — and the bet held, because at high QPS even a small sample contains thousands of examples of any recurring latency pattern. The cost they paid is the §4/§11 pitfall: rare one-off errors can be sampled away. (Modern OTel tail-sampling — “keep all errors + slow traces” — is the industry’s later correction to exactly this gap. Dapper chose simplicity and constant overhead over error-completeness.) The sampling decision is made once at ingress and *propagated* via the sampled flag, so a trace is either fully captured or not — no partial traces.

SLOs and error budgets — the operating contract. Tracing tells you *where* it broke; the SRE model from §6–8 tells you *whether you should care and what to do*. Google defines an **SLI** as $\text{good events} / \text{total events}$ (e.g. fraction of requests served correctly under a latency threshold), sets an internal **SLO** (commonly 99.9–99.99% depending on the service), and derives an **error budget** = $1 - \text{SLO}$. That budget is the currency from §6: while budget remains, teams ship features and run risky migrations; when it’s burned, a **feature freeze** kicks in and only reliability work ships. This is what dissolved Google’s dev-vs-ops tension — reliability became a *number with a budget*, not an argument.

Critically, Google alerts on **symptoms, not causes** (§7): pages fire on **SLO burn rate** (“at this rate the 28-day budget is gone in hours”), using multi-window alerts (a fast 5-minute window plus a slow 1-hour window) to be both responsive and resistant to flapping — not on “CPU > 80%.” And new versions roll out as a **canary** (§9): route a small slice of traffic to v2, watch its SLIs against the baseline, and auto-roll-back if the canary’s error/latency SLIs degrade — spending error budget deliberately and in a bounded blast radius.

Results. Dapper ran at Google with single-digit-percent (often <1%) request-latency overhead thanks to sampling, while still surfacing latency regressions across thousand-way fan-outs that were previously undebuggable. It became the template for an entire industry (Zipkin, Jaeger, OpenTelemetry). The SLO/error-budget model, published in the SRE book, is now the default reliability framework across the industry. Together they let Google operate services at availability targets (three-to-four-plus nines) while still deploying constantly — the apparent contradiction §6 resolves.

Lessons

- **Bake context propagation into shared infrastructure, not app code.** Dapper got near-universal coverage because tracing lived in the RPC library — teams opted in for free. Retrofitting per-service instrumentation never reaches full coverage.
- **Sampling is a deliberate trade, and head-sampling's blind spot is errors.** Constant-overhead head sampling buys affordability at scale; if you must keep rare errors, layer tail-sampling on top — know which one you chose and why.
- **Tie the whole loop together: traces diagnose, SLOs decide, error budgets govern, burn-rate alerts wake you, canaries gate change.** Observability is wasted if it isn't connected to an operating contract that turns signals into decisions.
- **100% is the wrong target.** Google's success came from *spending* the error budget to move fast, not hoarding it — the SLO is set to what users actually need, and headroom is fuel for change.

12. Test yourself

1. **Why do metric labels need low cardinality but trace attributes don't?** *Hint: each unique metric label-combo is a separately stored time series (RAM/disk per series); traces are per-event and queried, not pre-aggregated.*
 2. **Your service has p99 = 100ms but a user request fans out to 20 of them. Roughly what's the chance a request hits at least one p99-slow hop?** *Hint: $1 - 0.99^{20} \approx 18\%$. Tail-latency amplification — your p99 is the user's ~p82.*
 3. **At 99.9% SLO, how much monthly downtime is your error budget, and what should happen when it's exhausted?** *Hint: ~43 min; freeze feature work, ship only reliability fixes until budget recovers.*
 4. **Why should a DB connectivity check live in the readiness probe, not the liveness probe?** *Hint: liveness fail = restart; a shared DB blip would restart all instances simultaneously → thundering herd → cascading outage. Readiness just removes from the LB.*
 5. **Head-based vs. tail-based sampling — which keeps your error traces, and what does it cost?** *Hint: tail-based keeps errors/slow traces; cost is a buffering collector holding all spans until the trace completes (memory + complexity).*
 6. **You must deploy v2 plus a column rename. Why can't you do both in one shot, and what's the pattern?** *Hint: code rolls back in seconds, destructive schema doesn't. Expand/contract: add column → dual-write → backfill → switch reads → drop old later.*
 7. **Why is "CPU > 80%" a bad alert and "checkout success < 99.9%" a good one?** *Hint: alert on user-visible symptoms, not causes; high CPU can be harmless, success rate is what users feel. CPU is a post-page diagnostic.*
 8. **What's the difference between an SLO and an SLA, and why is SLA < SLO?** *Hint: SLO is internal target, SLA is contractual w/ penalties; you keep headroom so you breach the internal alert long before the customer contract.*
-

13. Further reading

- **Google SRE Book** — *Site Reliability Engineering* (free at sre.google/books). Read: Ch. 4 (SLOs), Ch. 5 (eliminating toil), Ch. 6 (monitoring distributed systems), Ch. 10 (alerting on SLOs), Ch. 15 (postmortem culture). The canonical source for everything in §6–8.
- **Google SRE Workbook** — practical SLO/error-budget implementation, multi-window burn-rate alerting.
- **DDIA (Kleinmann), *Designing Data-Intensive Applications*** — Ch. 1 (reliability/percentiles/tail latency).
- **OpenTelemetry docs** (opentelemetry.io) — spec, context propagation, Collector tail sampling.
- **W3C Trace Context spec** (w3.org/TR/trace-context) — the `traceparent` header format.
- **Prometheus docs** — instrumentation best practices, histograms, *and* the cardinality warnings; PromQL `rate()` / `histogram_quantile()`.
- **Brendan Gregg** — “The USE Method” (brendangregg.com/usemethod.html).
- **Tom Wilkie** — “The RED Method” (Weaveworks/Grafana blog).
- **Charity Majors et al., *Observability Engineering (Honeycomb)*** — wide structured events, high-cardinality debugging, the “unknown unknowns” framing.
- **Cindy Sridharan, *Distributed Systems Observability*** (free O’Reilly report) — the three pillars deep dive.

Cross-links: percentiles & latency math → `01-fundamentals-and-estimation.md` ; partial failure & consistency → `09-cap-pacelc-and-consistency.md` ; why a request crosses many services (and needs tracing) → `15-microservices-and-decomposition.md` ; resilience patterns that observability validates (circuit breakers, load shedding) → `16-resilience-and-fault-tolerance.md` .

Module 18 — Architecture Patterns: Monolith → Microservices → Cells

Source: [readme/research/system-design/18-architecture-patterns-microservices.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains how to decide whether to split a software system into smaller, separate pieces — and how to do it safely if you do. It matters because splitting too early wastes time and money, while splitting at the right time (and in the right way) lets a growing team ship changes faster without breaking things for customers.

The main parts explained simply:

- **Monolith vs Modular Monolith vs Microservices** — A monolith is one big app doing everything together. Microservices break it into many small apps that talk over a network. The middle option — a modular monolith — keeps one app but draws strict internal fences between areas. For most teams this middle option is the right answer and avoids most of the headaches.
- **Don't split too early** — Splitting a small system into microservices before you need to creates a lot of extra work (networking, debugging, deployments) without solving any real problem. Most companies should wait until they have 15–20+ engineers or a part of the system that needs to scale or fail independently.
- **Bounded contexts — split by business area, not by technology** — The safest place to draw a dividing line is where the same word means two different things. “Order” means something different to the sales team versus the shipping team — that natural difference is where to split, not along technical lines like “a separate database service.”
- **Strangler Fig — how to migrate without a risky rewrite** — Instead of throwing away the old system and building a new one from scratch (very risky), you grow the new pieces around the old one, move traffic piece by piece, and slowly retire the old parts. The hardest part is moving the data, not the code.
- **API Gateway and BFF — the front door** — An API gateway is a single entry point that handles security and routing for all clients. A BFF (Backend-for-Frontend) is a tailored version of that for a specific screen (mobile vs web), so each screen gets exactly the data shape it needs.
- **Service Mesh — networking on autopilot** — A service mesh (tools like Istio) handles retries, timeouts, encryption between services, and traffic routing automatically, without changing your app code. It is only worth the complexity when you have many services; for small systems a good HTTP library is enough.
- **Service Discovery — finding services automatically** — When services start and stop constantly, you cannot use fixed addresses. Discovery tools (like Kubernetes) give each service a stable name and automatically route traffic to healthy copies.
- **Database-per-service — each service owns its own data** — No two services should share the same database. If they do, changing one service's data structure forces changes in the other — they become secretly coupled even if deployed separately. Keeping data separate is what makes services truly independent.
- **Sagas — multi-step actions without a shared transaction** — When an action spans multiple services (reserve stock, charge card, create shipment), you cannot rely on a single database rollback if something fails. A saga runs each step separately and has a “undo” step ready for each one if something goes wrong later.

- **Distributed Monolith — the costly mistake** — This is when a team splits into microservices but the services still all depend on each other synchronously (one calls the next, which calls the next). You pay all the costs of separate services but get none of the benefits. Signs: services must deploy together, they share a database, one slow service makes everything slow.
- **Cell-based architecture — limiting damage when things go wrong** — Instead of one big shared system serving all customers, you run many identical small copies (cells) and pin each customer to one. If a bug or bad request hits cell 3, only cell 3's customers are affected — not everyone.
- **Shuffle Sharding — making damage even smaller** — Instead of one fixed cell per customer, each customer gets a random small set of servers from a shared pool. The chance two customers share all the same servers is very small, so a problem with one customer rarely fully affects another. Amazon uses this for Route 53 to protect customers from each other.

What to do with this:

Start with a modular monolith — strict internal boundaries, one deployment — and only extract a separate service when one part has a clearly different scaling or failure need (like a heavy file-processing job). When you do split, draw the line where business meaning changes naturally, move data carefully using the strangler approach, and always ask: “can I deploy this one piece alone right now?” If the answer is no, you have not really split yet.

What you'll learn: How to decide *whether and how* to split a system into services, where the real boundaries live (DDD bounded contexts, not org charts), how to migrate without a rewrite (strangler-fig), and the infra that makes it survivable (gateways, BFFs, service mesh, discovery). Then the advanced layer almost nobody gets right: avoiding the *distributed monolith*, and cell-based / shuffle-sharding architectures that cap blast radius in multi-region active-active systems.

Prerequisites: Read [09-cap-pacelc...md](#) (consistency trade-offs), [10-data-replication...md](#) (replication & per-service data), [14-messaging-and-queues...md](#) (async integration), and [16-observability...md](#) (you cannot run distributed systems blind). This module assumes you can already reason about partial failure and eventual consistency.

1. The core intuition: a restaurant kitchen

A **monolith** is one chef cooking every dish at one big station. Fast and simple when there's one chef — no coordination, everything's within arm's reach. But you can't add a second chef to the *same* station without them colliding; to go faster you clone the entire chef (and their whole station) even if only desserts are busy.

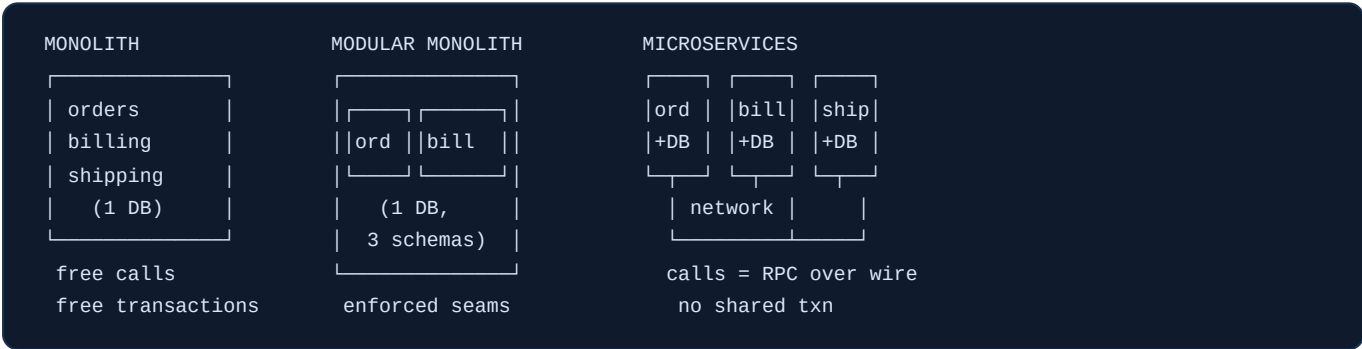
Microservices are specialized stations — grill, salad, pastry — each with its own tools and its own fridge, communicating by passing tickets. You can staff the busy stations independently. But now a dish that touches three stations needs the tickets to flow correctly, and if the pastry fridge dies, anyone who needed cream is stuck.

Cells are the realization that one giant kitchen serving 10,000 covers is a single point of catastrophe. So you build many identical *small* kitchens, and route each table to exactly one. A fire in kitchen #7 ruins dinner for 1/Nth of guests, not everyone.

The entire progression is a trade of **simplicity for independence and isolation** — and you pay in coordination, latency, and operational surface area. The senior skill is knowing how little of that trade you actually need.

2. Monolith vs Modular Monolith vs Microservices

The crucial middle option that junior engineers skip: the **modular monolith**. One deployable, one database, but with hard *internal* module boundaries (separate schemas, no cross-module SQL joins, communication via in-process interfaces only). It gets you most of the design discipline of microservices with none of the network.



Dimension	Monolith	Modular Monolith	Microservices
Deploy unit	1	1	N independent
Cross-boundary call	function call (~ns)	function call (~ns)	network RPC (~0.5–5 ms LAN)
Transactions	ACID, free	ACID, free	sagas / eventual consistency
Team scaling	contention on 1 codebase	per-module ownership	full autonomy
Independent scaling	no (clone whole app)	no	yes (scale hot service)
Failure isolation	none	none (shared process)	per-service
Operational cost	low	low	high (mesh, CI/CD, on-call ×N)
Debugging	stack trace	stack trace	distributed tracing required
When to use	startup/MVP, <5 eng	most companies, scaling teams before scaling load	independent scaling/failure needs, large org, proven boundaries

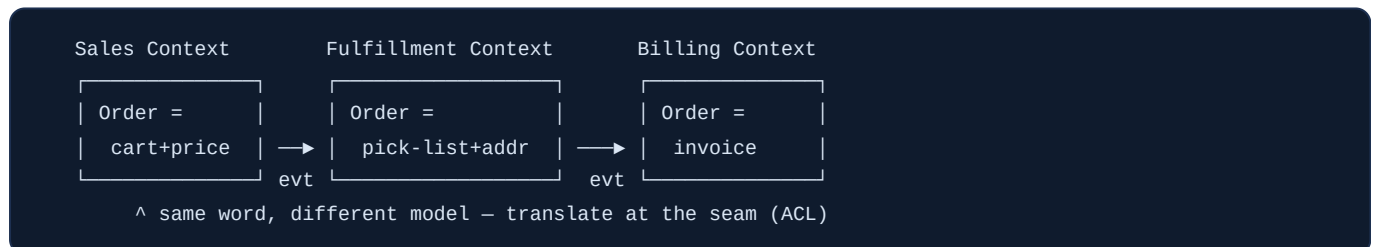
Opinionated take — do NOT split early. Microservices solve an *organizational* and *isolation* problem, not a “clean code” problem. Below ~15–20 engineers you almost never need them; you need module discipline. Shopify, GitHub, and Stack Overflow ran enormous monoliths for years. Shopify’s “modular monolith” (the *Componentization* effort) is the canonical proof that you can get boundary discipline without distribution tax. Premature microservices give you all the costs (network failure, eventual consistency, deploy orchestration) before you have the problem they solve.

Back-of-envelope on the network tax: A request that touches 4 modules in a monolith is 4 function calls — microseconds total. The same 4 as microservices is 4 RPCs. At p50 ~1 ms each that’s ~4 ms; but tail latency compounds: if each service is p99 = 20 ms, a *serial* fan-out of 4 has a p99 closer to ~40–60 ms because you’re rolling the dice four times. **Tail amplification is the silent killer** of naive microservices (see [13-latency...md](#) on the “tail at scale”, Dean & Barroso).

3. Finding boundaries: DDD bounded contexts

The #1 cause of failed microservice migrations is splitting on the wrong axis. **Do not split by technical layer** (a “database service”, an “auth service” that everyone calls synchronously) or by org chart. Split by **bounded context** — a region of the domain where a term has one unambiguous meaning.

The litmus test: the word “Order” means different things to different parts of the business. In *Sales* an Order is a cart with pricing. In *Fulfillment* it’s a pick-list with a shipping address. In *Billing* it’s an invoice line. Each meaning is a bounded context; the seam between them is a natural service boundary.



Heuristics for good boundaries:

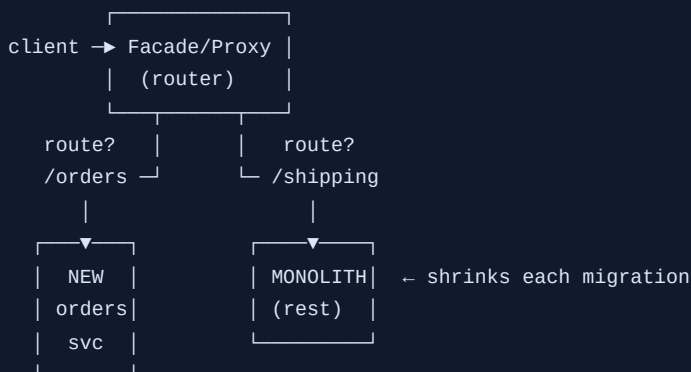
- **High cohesion, low coupling.** Things that change *together* belong together. If every feature touches two services, they’re one service wearing a costume.
- **Data ownership.** A service owns its data exclusively. If two services need to write the same table, you drew the line wrong.
- **Async over sync at the seam.** Contexts integrate by *events* (“OrderPlaced”) not synchronous queries where possible — this decouples availability.
- **Anti-Corruption Layer (ACL).** When a context consumes another’s model, translate it at the edge so upstream changes don’t leak in.

In Print-Flow-360 terms: a *bounded context* maps cleanly to the modular-monolith idea — [orders](#) , [pricing](#) , [pdf-service](#) , [designer](#) are already context-shaped. Notice [pdf-service](#) is the one piece extracted as a real separate service, precisely because it’s a *different scaling and failure profile*

(heavy CPU/IO file work). That's the right reason to extract: different operational characteristics, not "it felt cleaner."

4. The Strangler-Fig migration

Named after the fig vine that grows around a host tree until the tree dies and only the fig's hollow shell remains — you grow the new system around the monolith and route traffic over piece by piece. **Never do a big-bang rewrite** (see the war stories; Netscape, the canonical cautionary tale).



Steps: (1) put a **facade/proxy** in front of the monolith so the boundary is movable; (2) carve out *one* bounded context behind it; (3) route that path to the new service; (4) **dual-write or replicate data** during transition; (5) backfill + verify; (6) cut reads over; (7) delete the dead code in the monolith. Repeat. Each step is independently shippable and reversible.

The hard part is data, not code. You'll run a period where both old and new own a slice. Patterns: *change-data-capture* (Debezium tailing the monolith's WAL to feed the new service), *dual writes* (risky — partial-failure inconsistency; prefer CDC or outbox), and the **outbox pattern** (write business row + event row in one local transaction, relay publishes the event — atomic, no distributed transaction). See [14-messaging...md](#) for outbox details.

5. Edge & inter-service infra

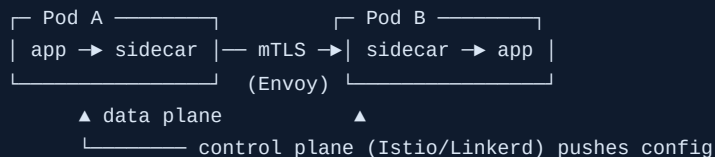
API Gateway vs BFF

	API Gateway	Backend-for-Frontend (BFF)
Purpose	single front door: auth, rate-limit, routing, TLS termination	per-client API tailored to one frontend (web/iOS/Android)
Count	usually 1 (or 1/region)	1 per client type
Aggregates calls?	sometimes	yes — its whole point
Owner	platform team	the frontend team that consumes it
Risk	becomes a “god gateway” with business logic	duplication across BFFs
Real systems	Kong, AWS API Gateway, Envoy, Zuul	Netflix coined BFF; SoundCloud popularized it

The BFF exists because a mobile screen and a web dashboard need *different* shapes of the same data, and you don’t want every backend service to know about UI concerns. Keep the gateway dumb (cross-cutting concerns only) and push aggregation into BFFs. The anti-pattern is a gateway that grows business logic until it’s a monolith you can’t deploy independently.

Service Mesh (sidecars, mTLS, traffic shaping)

A mesh moves networking concerns *out of your app* into a **sidecar proxy** (Envoy) deployed next to each instance. Your code makes a plain `localhost` call; the sidecar handles retries, timeouts, mTLS, load balancing, circuit breaking, and telemetry.



What it buys you: **mTLS everywhere** (zero-trust between services), **traffic shaping** (canary 5% → 50% → 100%, blue/green, mirroring), **resilience** (per-route timeouts, retries with budgets, circuit breakers) *without recompiling apps*, and uniform L7 metrics. Real: **Istio** (Envoy + heavy control plane), **Linkerd** (lighter, Rust micro-proxy), AWS App Mesh.

Opinionated: a mesh is *overkill* below ~10–15 services. Sidecars add ~0.5–1 ms per hop and real memory/CPU overhead per pod, plus a notoriously complex control plane. Start with a good HTTP client library (timeouts + retries + circuit breaker, e.g. resilience4j/Polly) and graduate to a mesh when polyglot services or compliance-grade mTLS force your hand.

Service Discovery

Services come and go (autoscaling, redeploys), so hardcoded IPs are dead on arrival. Two models:

Model	How	Examples	Trade-off
Client-side	client queries registry, picks an instance, calls it	Netflix Eureka + Ribbon, Consul	client is smarter; LB logic duplicated per language
Server-side	client hits a stable VIP/LB which routes	Kubernetes Service + kube-dns, AWS ELB	simpler clients; extra hop

Kubernetes makes this mostly invisible: a `Service` gives a stable DNS name and virtual IP; kube-proxy/iptables (or the mesh) load-balances to healthy pods via readiness probes. The deep gotcha: **health checks must reflect real readiness** (can it serve traffic + are its deps up?), or discovery will happily route to a zombie.

6. Database-per-service — the rule and the pain

The rule: each service owns its database; no other service touches it directly. This is what makes services independently deployable — you can change `orders` ' schema without coordinating a global migration. Violate it (shared database) and you have a *distributed monolith*: separate deploys but coupled at the data layer, so nothing is actually independent.

The pain it creates — and how the field solves it:

- **No cross-service JOINS.** You can't `JOIN orders.customer_id = customers.id` across a network. Solutions: **API composition** (fetch from each, join in memory — simple, but N calls and no DB-level filtering), or **CQRS** (maintain a read-optimized materialized view fed by events — fast reads, eventual consistency, more moving parts).
- **No distributed ACID transaction.** “Reserve inventory AND charge card AND create shipment” spans 3 services. You can't 2PC at scale (blocking, coordinator is a SPOF). Use a **Saga**: a sequence of local transactions, each with a **compensating action** to undo on failure.

```
SAGA (orchestrated)           on failure → compensate backward
Order → reserve inventory → charge card → create shipment
      ▲ release stock ← refund ← (rollback chain)
```

Saga style	How	Pro	Con
Choreography	services react to each other's events	no central coordinator; loose	logic smeared across services; hard to follow
Orchestration	a coordinator drives the steps	explicit, debuggable	coordinator is a new component to run

Sagas trade **atomicity for availability** and expose *intermediate* states to the world (an order can be “payment pending”). You must design for that (idempotent steps, since retries will re-deliver; see `14-messaging...md` on exactly-once-illusion). Real systems: **AWS Step Functions**, **Temporal**, **Camunda** for orchestration.

7. The Distributed Monolith — the anti-pattern that eats teams

This is the failure mode that gives microservices a bad name: you paid the full distribution tax and got *none* of the independence. Symptoms:

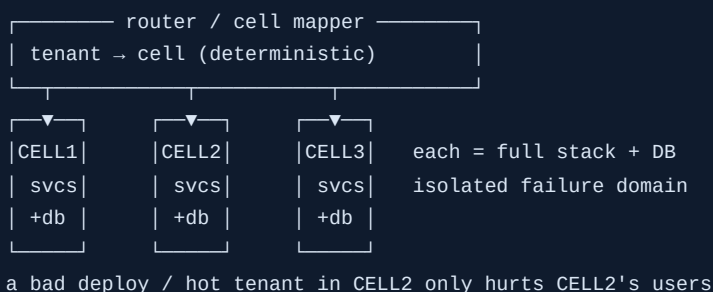
- **Synchronous call chains.** $A \rightarrow B \rightarrow C \rightarrow D$ where every request blocks on the next. Availability multiplies *down*: four services at 99.9% in series $\approx 99.6\%$ (≈ 35 hrs/yr down vs 8.7). And tail latency compounds.
- **Shared database** behind multiple services.
- **Lock-step deploys.** “You can’t release `orders` until `pricing` ships v2.” If services must deploy together, they’re one service.
- **Shared library hell.** A change to a common DTO/proto forces a coordinated rollout of everyone.
- **Chatty interfaces.** One user action = 30 internal calls.

DISTRIBUTED MONOLITH (bad)	TRUE MICROSERVICES (good)
$A \rightarrow B \rightarrow C \rightarrow D$ (sync chain)	$A \xrightarrow{\text{evt}} [\text{bus}] \rightarrow B$
availability multiplies down	services react async, deploy alone
one deploy needs all of them	no temporal coupling

Fixes: integrate via **async events** not sync RPC at the seam (temporal decoupling); enforce **independent deployability** as a hard gate (“can I deploy this service alone, right now?”); version contracts and never break consumers (see [CLAUDE.md](#)’s own backward-compat rule — same principle); collapse chatty chains by *moving the boundary* (if A always calls B, maybe A and B are one context).

8. Cell-based architecture & shuffle sharding (blast-radius isolation)

The advanced frontier. Even with perfect microservices, a *poison-pill* request, a bad deploy, or a hot tenant can take down a shared service for **everyone**. Cells fix this by partitioning the *entire stack* into independent, identical copies and pinning each customer to one.



Blast radius math: with 1 shared stack, a fatal bug = 100% impact. With 10 equal cells, a cell-local failure = $\sim 10\%$. The router (cell mapper) must be dead-simple and ultra-reliable — it’s the only shared component, so keep it thin (a lookup table, ideally cached/static), not a smart service.

Shuffle sharding — the elegant upgrade

Plain sharding into cells still means *all* the (say) 100 tenants in cell #2 share a fate. **Shuffle sharding** (AWS Route 53 / popularized by Colm MacCárthaigh) assigns each customer a *random small subset* of nodes instead of one cell, so two customers rarely share their *full* set.

Toy math: 8 worker nodes, give each customer a random pair (2 nodes). There are $C(8,2)=28$ possible pairs. A poison customer melting *both* their nodes only fully knocks out the *handful* of other customers sharing the *exact same pair*. The probability another customer shares both your nodes is tiny — you get isolation approaching “private infrastructure per customer” at a fraction of the cost.

```
Customer A → {n1, n4}
Customer B → {n2, n7} ← shares 0 nodes with A
Customer C → {n1, n7} ← shares only 1 node with A (still served by n4)
A node dies → most customers still have a surviving node
```

Isolation strategy	Blast radius	Cost	When
Single shared stack	100%	lowest	small, low-stakes
Cells (N)	~1/N	N× infra-ish	multi-tenant SaaS, noisy-neighbor risk
Shuffle sharding	~combinatorially tiny	shared pool, low	many customers, want near-private isolation cheaply

Real systems: **AWS** runs cell-based architecture across S3, DynamoDB, Lambda; **Slack** publicly moved to cells; **Route 53** uses shuffle sharding. The trade-off: cross-cell operations are forbidden by design (a customer lives in one cell), and you need rock-solid per-cell deploy automation (you deploy to cells one at a time — automatic canary across the fleet).

Multi-region & active-active

Cells generalize to regions. **Active-passive** (one region serves, another on warm standby) is simple but wastes capacity and has failover lag + risk of cold standby. **Active-active** (all regions serve writes) maximizes availability and locality but forces you to confront the hardest problem in the field: **multi-region data consistency** (see [09-cap-pacelc...md](#) and [10-data-replication...md](#)).

Topology	RTO/RPO	Write consistency	Complexity	When
Single region	bad (region = SPOF)	trivial	low	most apps, honestly
Active-passive	minutes RTO, ~0 RPO if sync replica	one writer	medium	compliance/DR needs
Active-active	~0 RTO	hard (conflicts, CRDTs, or globally-ordered)	high	global, latency-sensitive, 99.99%+

Active-active needs either **partitioned writes** (region owns a tenant’s data — easiest, maps perfectly onto cells), **CRDTs / last-writer-wins** (eventual, accept conflicts — DynamoDB Global Tables), or a **globally consistent store** (Google **Spanner** / **CockroachDB** using TrueTime/Raft to give external consistency

across regions — at a latency cost: cross-region commits pay the speed of light, ~30–150 ms round trips). You cannot have low-latency global writes *and* strong consistency *and* partition tolerance — physics and CAP both say no.

9. Common pitfalls / war stories

- **The premature split.** A 6-person startup splits into 12 microservices “to be ready to scale.” They now spend 80% of their time on CI/CD, service discovery, and distributed debugging instead of features. They had zero scaling problem. *Fix: modular monolith first; extract a service only when it has a distinct scaling or failure profile (like `pdf-service`).*
 - **Splitting by layer, not domain.** A “database access service” everyone calls synchronously — congratulations, you built a network-attached bottleneck and a SPOF. Boundaries are *vertical* (domain), not horizontal (tech layer).
 - **The shared database that “we’ll fix later.”** Two services on one DB = distributed monolith forever. The schema becomes a global contract nobody can change. *Fix it during the strangler migration, not after.*
 - **Big-bang rewrite.** The mythical “v2 from scratch while v1 runs” — by the time v2 catches up, requirements moved and you have two systems. (Joel Spolsky’s “Things You Should Never Do” / the Netscape 6 collapse.) *Strangle, don’t rewrite.*
 - **Sync fan-out tail latency.** A page that synchronously calls 20 services has a p99 dominated by the *slowest of 20* on every request. *Fix: parallelize, set aggressive timeouts + fallbacks, cache, or precompute via events.*
 - **Retry storms / no budgets.** Service B slows down, everyone retries, the retries amplify load, B falls over harder — a *metastable failure*. *Fix: retry budgets, exponential backoff + jitter, circuit breakers, load shedding.*
 - **Distributed transaction via dual writes.** “Write to DB then publish to Kafka” — the process crashes between the two and you’ve lost the event silently. *Fix: outbox pattern (one local txn) + CDC relay.*
 - **Mesh as a cargo cult.** Installing Istio for 5 services and then spending a quarter debugging Envoy config and sidecar OOMs. *Earn the mesh.*
 - **No idempotency in sagas.** At-least-once delivery means every saga step runs more than once eventually. A non-idempotent “charge card” step double-charges. *Every step needs an idempotency key.*
 - **One cell that’s special.** A cell-based system where “the big customer gets their own special config” — now you have a snowflake and your per-cell deploy automation lies to you. *Cells must be identical.*
-

🧩 Case Study: Amazon — the service-oriented mandate + AWS cell-based architecture

The problem (early 2000s). [Amazon.com](#) began life as a single C++/Perl monolith called **Obidos**, fronting one giant Oracle database. By the early 2000s the site was serving tens of millions of customers and the monolith had become a *deployment chokepoint*: every team's change had to be coordinated into one release train, a single bad commit could brick the whole storefront, and the shared database was a contention point nobody could evolve without negotiating with everyone else. This is exactly the **distributed monolith trap from §7**, except it was still a **literal monolith** — lock-step deploys and a shared database, just without the network in between. Growth was throttled not by hardware but by *coordination*.

The fix, step one — Bezos's service mandate (~2002). The (now legendary) internal memo decreed: all teams expose their data and functionality *only* through service interfaces over the network; no team may read another team's database directly; there is no other inter-team communication allowed. This is the **database-per-service rule from §6 enforced as org policy**, and it simultaneously forced the **bounded-context discipline from §3**: each team had to define a clean contract for its slice of the domain (catalog, cart, ordering, payments, fulfillment) and own its data exclusively. Crucially they did *not* big-bang rewrite — they strangled Obidos service by service (**§4 strangler-fig**), routing each capability out from behind the monolith over time.

BEFORE (Obidos monolith)

```
| catalog/cart/order |  
| payment/fulfillment |  
| (one Oracle DB) |
```

one deploy train, shared schema

AFTER (service mandate)

catalog —┐

cart —┐

order —┐

payment —┐

fulfil —┐

independent deploy per service

each owns its DB,

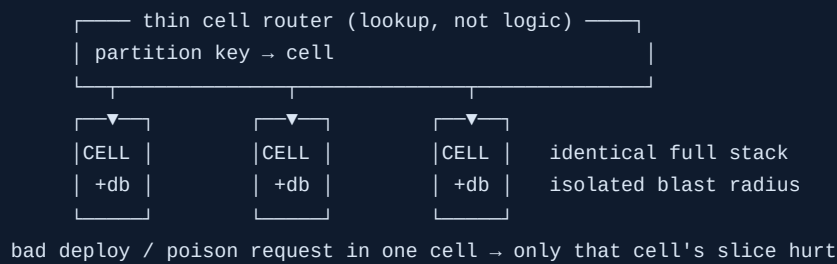
talks only via API

no shared tables

no lock-step deploy

The payoff was organizational throughput: hundreds, then thousands of **two-pizza teams** deploying independently — Amazon later reported deploying to production on the order of **once every ~11.6 seconds** across the fleet. That cadence is *impossible* with a shared database or lock-step releases; it is the direct dividend of the §6 ownership rule plus the §7 independent-deployability gate.

The fix, step two — cell-based architecture inside AWS. The same teams went on to build AWS, and there they hit the next ceiling from §8: even with clean services, a single shared service plane means a poison-pill request, a hot tenant, or a bad config push can hurt *every* customer at once. So AWS services (DynamoDB, S3, Lambda, SQS, and many control planes) are partitioned into **cells** — full, identical stacks (compute + storage + their own data) — with each customer/partition deterministically mapped to one cell by a deliberately *thin, dumb router* (the **cell mapper from §8**, kept simple because it's the one shared component).



With N cells a cell-local failure caps impact at $\sim 1/N$ instead of 100% — the **blast-radius math from §8**. Deploys go cell-by-cell as an automatic canary across the fleet, so a regression is caught in one small cell before it reaches the rest.

Shuffle sharding on top (§8). For services where even per-cell fate-sharing was too coarse, AWS (notably **Route 53** and the **AWS Shield/Builders' Library** work by Colm MacCárthaigh) applies **shuffle sharding**: each customer gets a *random small subset* of nodes from a shared pool rather than one fixed cell. With a pool of nodes and, say, a per-customer set of a few, the chance any two customers share their *entire* set is combinatorially tiny — so a single abusive or poisoned customer can fully knock out only the handful sharing its exact set, while everyone else keeps a surviving node. This buys near-“private infrastructure per customer” isolation from a *shared* pool, which is the whole point: isolation without $N\times$ the cost.

The key trade-off they accepted. Cells (and the service mandate before them) **forbid cheap cross-boundary operations**. Once a customer lives in one cell, you cannot trivially run a query or transaction that spans cells — cross-cell joins and global transactions are *designed out*. Internally that meant giving up convenient distributed ACID in favor of **API composition and sagas (§6)** and embracing eventual consistency at the seams (an order can sit in “pending” between services). They traded **developer convenience and strong global consistency for independent deployability and bounded blast radius** — and at Amazon’s scale, *availability and isolation were worth far more than the convenience of a JOIN*.

Results. Independent per-team deploys (the ~ 11 s cadence above) replaced the monolith’s coordinated release train; cell isolation turned what would have been platform-wide AWS outages into incidents scoped to a fraction of customers; shuffle sharding pushed effective blast radius down to a tiny combinatorial fraction. The architecture is now the canonical reference for multi-tenant SaaS reliability.

Lessons

- **Enforce boundaries as policy, not aspiration.** The §6 database-per-service rule only stuck because Amazon made “no direct DB access, talk via API” a *mandate* — discipline you merely hope for erodes; discipline the platform enforces survives.
- **Independent deployability is the real prize.** The ~ 11 -second deploy cadence is the measurable payoff of §6 + §7 — if you can’t deploy one service alone, you don’t have microservices, you have a distributed monolith.
- **Cap blast radius before you chase five-nines.** Cells (§8) accept the cost of forbidding cross-cell operations in exchange for turning 100%-impact failures into $1/N$ -impact failures; shuffle sharding sharpens that to a combinatorial sliver — isolation, not raw uptime, is what bounds the worst day.

- **Keep the one shared component dumb.** The cell router/mapper is the single point everyone depends on, so it must be a thin, boring lookup — putting logic there recreates the very SPOF cells exist to eliminate.

Test yourself

1. A team wants to split a 4-service synchronous call chain ($A \rightarrow B \rightarrow C \rightarrow D$, each 99.9% available) for “reliability.” What’s the combined availability, and why might splitting *hurt* reliability here? (*Hint: $0.999^4 \approx 99.6\%$; serial sync chains multiply availability down and compound tail latency.*)
 2. You’re migrating a monolith and must run both old and new `orders` for a month. How do you keep their data consistent without distributed transactions? (*Hint: CDC/outbox + dual-read verification, not dual writes.*)
 3. Why is “split by bounded context” better than “split by org chart” or “split by technical layer”? (*Hint: cohesion/coupling and data ownership; layer-splits create chatty SPOFs.*)
 4. When is a service mesh *not* worth it, and what do you use instead? (*Hint: $< \sim 10$ services / homogeneous stack; resilient HTTP client libraries.*)
 5. Give the math for why 10 cells reduces blast radius vs a shared stack, and how shuffle sharding improves on plain cells. (*Hint: $1/N$ vs combinatorial $C(n,k)$ overlap probability.*)
 6. Name three ways to do multi-region active-active writes and the consistency cost of each. (*Hint: partitioned ownership; CRDT/LWW eventual; Spanner-style globally-consistent at cross-region latency.*)
 7. What three symptoms tell you you’ve built a *distributed monolith* rather than microservices? (*Hint: shared DB, lock-step deploys, sync chains / chatty interfaces.*)
 8. Why must every saga step be idempotent, and how do you enforce it? (*Hint: at-least-once delivery $\rightarrow$ replays; idempotency keys / dedup store.*)
-

11. Further reading

- **DDIA** (Kleppmann), *Designing Data-Intensive Applications* — Ch. 1 (maintainability/scalability), Ch. 7–9 (transactions, distributed consistency) underpin saga/CAP reasoning.
- Sam Newman, *Building Microservices* (2nd ed.) and *Monolith to Microservices* — the canonical strangler-fig and decomposition references.
- Eric Evans, *Domain-Driven Design* + Vaughn Vernon, *Implementing DDD* — bounded contexts, context maps, ACL.
- Chris Richardson, microservices.io + *Microservices Patterns* — saga, CQRS, API composition, outbox (definitive pattern catalog).
- Martin Fowler — “MonolithFirst”, “StranglerFigApplication”, “BoundedContext”, “MicroservicePremium” essays.
- Colm MacCárthaigh / AWS — “Shuffle Sharding” (AWS Builders’ Library) and “Workload isolation using shuffle-sharding”.

- AWS Builders' Library — “Avoiding fallback in distributed systems”, cell-based architecture write-ups; AWS Well-Architected reliability pillar.
- Dean & Barroso, “*The Tail at Scale*” (CACM 2013) — why fan-out kills tail latency.
- Corbett et al., “*Spanner: Google’s Globally-Distributed Database*” (OSDI 2012) — TrueTime + globally consistent writes.
- Joel Spolsky, “*Things You Should Never Do, Part I*” — why not to big-bang rewrite.
- Istio / Linkerd / Envoy official docs; Kubernetes Services & readiness probes docs.

Next: see [14-messaging-and-queues...md](#) (outbox, idempotency, async seams) and [09-cap-pace1c...md](#) (the consistency cost you inherit the moment data crosses a service boundary).

Module 19 — Specialized Systems: Search, Geospatial, Time-Series & Analytics

Source: [readme/research/system-design/19-specialized-systems-search-geo-timeseries.md](#)

IN SHORT — KEY TAKEAWAYS

This document explains why a standard database like PostgreSQL cannot handle every type of question efficiently, and which specialist tool to use instead. It covers the four most common “hard” workloads — searching text, finding nearby places, recording measurements over time, and totaling huge amounts of data — and shows how each specialist tool is built to be dramatically faster.

The main parts explained simply:

- **Match the tool to the question** — A regular database works well for fetching rows by ID. But text search, proximity, time-series tracking, and billion-row analytics each need a different purpose-built store. The key skill is recognizing which category your problem falls into, then choosing the right tool rather than forcing one tool to do everything.
- **Text search (inverted index)** — Search engines like Elasticsearch build an index like the one at the back of a textbook: word → list of documents that contain it. Searching is a fast lookup, not a scan of every document. Results are ranked by how often the word appears and how rare it is across all documents (the BM25 formula handles this automatically, and it is the default in Elasticsearch).
- **Finding nearby places (geospatial indexes)** — “Find the 50 nearest print shops” is a two-direction problem — latitude and longitude together. Tools like PostGIS and Redis GEO flatten two dimensions into a single sortable key so nearby places always cluster together. They also fix the “border seam” trap: always search the target cell plus all surrounding cells, then measure real distance, so shops right on a boundary are never missed.
- **Tracking measurements over time (time-series databases)** — Tools like Prometheus and TimescaleDB store metrics (error rates, order counts, server load) with extreme compression — about 1–2 bytes per data point instead of 16. Old fine-grained data is automatically rolled up into hourly summaries and the detail is dropped to save space. One critical rule: never add a unique per-user or per-request identifier as a label — it creates millions of separate data streams and crashes the system.
- **Summing billions of rows (columnar analytics)** — Databases like ClickHouse or BigQuery store each column separately on disk. A “total sales by region” query reads only the two relevant columns out of two hundred — 10–100× less data than a regular database which must read every column of every row to reach the two it needs. Data is typically arranged in a “star” layout: one large events table in the center, surrounded by small descriptor tables (products, dates, customers).
- **Data lakehouse** — The modern approach: keep data as cheap files on cloud storage, but add a layer on top that gives you reliable saves, rollbacks, and SQL queries. One copy of the data can serve both spreadsheet-style reports and machine-learning workloads without expensive copying between separate systems.
- **Polyglot persistence (using multiple stores together)** — The mature approach is to keep your main database as the reliable source of truth and feed a derived copy into specialist stores for fast reads. Orders live in PostgreSQL; a search index is built from them for text queries; a columnar store gets a copy for analytics. Each read goes to the right tool; all writes start at the source.

What to do with this: When a feature needs text search, map proximity, time-based charts, or big-data reporting, reach for the specialist tool — do not try to squeeze it into a general-purpose database query. Keep your main database as the single reliable source and treat specialist stores as fast, rebuildable copies.

What you'll learn: When a relational table (`SELECT ... WHERE col LIKE '%foo%'`) is the wrong tool, and which purpose-built store replaces it. We cover the four most common “my Postgres can't do this efficiently” workloads — full-text search, geospatial proximity, time-series telemetry, and analytical aggregation — down to the index data structures, ranking math, and failure modes that surface in production and in interviews.

Prerequisites: Read `04-storage-engines-btree-lsm.md` (B-tree vs LSM, since these stores reuse both), `05-indexing-and-query-execution.md`, and `13-replication-and-partitioning.md` (sharding shows up everywhere here). For the OLAP section, `09-cap-pacelc-consistency-models.md` helps frame why analytical stores relax consistency.

0. The core mental model: indexes are shaped like queries

A row-oriented OLTP database (MySQL, Postgres) is optimized for one access pattern: *fetch/modify a few rows by primary key or a selective index*. Its B-tree answers “give me the row(s) where `id = x` or `created_at BETWEEN a AND b`”. Every specialized system in this module exists because some query *shape* doesn't fit that:

Query shape	Wrong tool (why)	Right tool (index used)
"docs containing 'red business cards' ranked by relevance"	LIKE '%...%' = full scan, no ranking	Inverted index (Lucene)
"50 nearest print shops to (lat, lng)"	B-tree is 1-D; 2-D range needs 2 indexes intersected	R-tree / geohash / S2
"avg CPU per minute over 30 days, 1M series"	billions of tiny rows, B-tree write amplification	Columnar TSDB w/ downsampling
"revenue by region by month across 2B order rows"	reads 100% of columns to aggregate 3 of them	Columnar OLAP (star schema)

The skill being taught: **recognize the shape, name the structure, justify the trade-off.**

1. Search — the inverted index

Intuition

A book's index at the back maps *word* → *page numbers*. That's an inverted index. The “forward” direction is `document → words`; you *invert* it to `word → list of documents`. Now “find docs containing X” is a hash lookup, not a scan.

Mechanics

Three stages turn text into a queryable index:

```
"Red Business Cards!"
| 1. Tokenization   -> ["Red", "Business", "Cards", "!"]
| 2. Filtering      -> lowercase, strip punctuation, stem ("cards"->"card"),
|                   drop stopwords ("the", "a")
| 3. Indexing       -> postings lists
▼
TERM      POSTINGS (docID:freq:[positions])
"red"     -> [12:1:[0], 88:2:[3,9], 204:1:[5]]
"busi"    -> [12:1:[1], 47:1:[0]]
"card"    -> [12:2:[2,2], 88:1:[7]]
```

The pipeline (Lucene calls it an **analyzer**) is the single most important and most misused part. *The same analyzer must run at index time and query time.* If you stem at index time but not query time, “running” never matches “run”. Components:

- **Tokenizer** — splits on whitespace/punctuation, or uses n-grams (for CJK languages with no spaces), or edge n-grams (for typeahead).
- **Token filters** — lowercasing, stemming (Porter/Snowball), synonym expansion, ASCII-folding (café → cafe), stopwords removal.

A **postings list** stores, per term, the documents containing it plus per-doc frequency and positions (positions enable phrase queries — “red card” as an ordered pair, not just both words anywhere). Postings are stored compressed (delta-encoded doc IDs + variable-byte/FOR encoding) so a term appearing in 10M docs isn’t 80MB.

Ranking: TF-IDF → BM25

You don’t just want matches, you want them *ranked*. Two classic intuitions:

- **Term Frequency (TF)**: a doc mentioning “card” 5× is more about cards than one mentioning it once.
- **Inverse Document Frequency (IDF)**: “card” appearing in every doc is uninformative; “lenticular” appearing in 3 docs is highly discriminating. Rare term = high weight.

TF-IDF = $TF \times \log(N / df)$. **BM25** (Lucene/Elasticsearch default since ES 5.0) fixes two TF-IDF flaws:

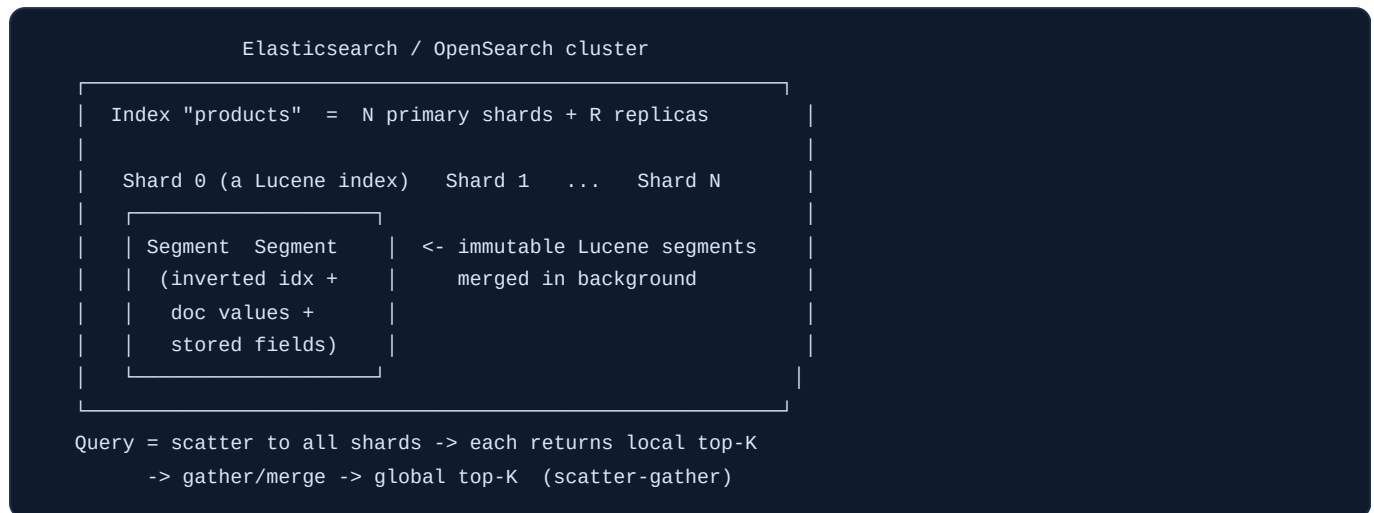
1. **TF saturation** — the 20th occurrence of a word shouldn’t add as much as the 2nd. BM25 has a saturation curve controlled by **k1** (default 1.2).
2. **Length normalization** — a 10,000-word doc naturally contains more terms; penalize length via **b** (default 0.75) relative to average doc length.

$$BM25 = \sum_{terms} \frac{IDF(term) \cdot TF(term, doc) \cdot (k1 + 1)}{TF + k1 \cdot (1 - b + b \cdot |doc| / avgdoclen)}$$

Back-of-envelope: a single-shard BM25 query over 10M docs returns top-10 in **~5–50 ms** because it walks compressed postings, not raw text. The cost driver is *number of shards fanned out × postings*

length, not corpus size on disk.

Architecture: Lucene → Elasticsearch/OpenSearch



Key architectural facts that bite people:

- **Segments are immutable.** A document update = mark old doc deleted (tombstone) + write new doc to a new segment. Deletes reclaim space only on **merge**. This is an LSM-style design (see [04-...](#)).
- **Near-real-time, not real-time.** A write goes to an in-memory buffer + translog; it's not searchable until a **refresh** (default 1s). So ES is eventually consistent for reads-after-write. Don't build "user posts comment, immediately re-query ES to show it" — read your own write from the source DB.
- **Doc values** are a columnar, on-disk structure (separate from the inverted index) used for sorting, aggregations, and faceting — because the inverted index is term → doc, but sorting needs doc → value.
- **Shard count is fixed at creation.** Over-shard and you get tiny-segment overhead + scatter-gather fan-out cost; under-shard and you can't scale or rebalance. Rule of thumb: target **10–50 GB per shard**.

Fuzzy / typeahead / did-you-mean

Technique	How	When to use
Edit distance (Levenshtein)	Match terms within N character edits; Lucene uses a Levenshtein automaton, not pairwise comparison	Typo tolerance ("buisness" → "business"); cap at distance 1–2
Edge n-grams	Index <code>c</code> , <code>ca</code> , <code>car</code> , <code>card</code> at index time	As-you-type prefix autocomplete; fast but inflates index
Completion suggester (FST)	A finite-state transducer in memory	Sub-ms typeahead; can't do mid-word matching
n-gram (non-edge)	Index all substrings	Infix/"contains" matching; very large index
Phonetic (Soundex/Metaphone)	Index by how it sounds	Name search ("Smith"/"Smyth")

War-story trade-off: edge n-grams blow up index size and force you to reindex to change min/max gram size. The completion suggester is faster but lives in heap memory — large suggestion sets cause GC pressure.

2. Geospatial — making 2-D queries fast

Intuition

A B-tree sorts on one dimension. “Find shops within 5 km of me” is inherently 2-D: a point close in latitude can be far in longitude. You can’t sort 2-D space linearly... unless you **flatten 2-D into 1-D while preserving locality** (so nearby points get nearby keys). That’s the whole game.

The four data structures

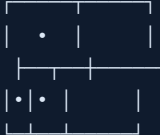
GEOHASH (space-filling curve)

Divide world into a grid, interleave lat/lng bits into one base32 string. "9q8yy" ~ ±0.6km cell.

Prefix match = same region.

△ adjacent cells can differ in prefix (the "edge problem").

QUADTREE (recursive subdivision)



split a cell into 4 when it has too many points; dense areas get deeper trees.

R-TREE (bounding boxes)

Group nearby objects in minimum bounding rectangles, nested in a tree. Great for polygons/extents, not just points. (PostGIS GiST uses R-tree.)

S2 (Google) / **H3** (Uber)

Project sphere onto a cube, use a Hilbert curve per face. Fixes geohash's poles/edge distortion; S2=squares, H3=hexagons.

Structure	Indexes	Strength	Weakness	Real systems
Geohash	points	Dead simple, prefix = region, works in any KV store	Edge/seam problem; rectangular cells distort near poles	Redis GEO (geohash on a sorted set), early Elasticsearch
Quadtree	points	Adapts to density; great for “load tiles in viewport”	Unbalanced under skew; rebalancing cost	Many in-memory map engines
R-tree (GiST)	points + polygons	Handles extents/shapes, true distance, rich predicates	More complex; insert/update heavier	PostGIS, SQLite R*Tree, MySQL spatial
S2 / H3	points + regions	Uniform cells, hierarchical, no seam distortion, set ops on regions	Library, not a DB; learning curve	Google Maps, Uber (H3 for surge zones), Foursquare

Why the edge problem matters (the classic interview trap)

With pure geohash prefix matching, two points 10 m apart can sit in cells with different prefixes (you’re on a cell boundary). A naive “same prefix = nearby” query *misses neighbors across the seam*. Fix: query your

cell **plus its 8 neighbors**, then post-filter by true Haversine distance. Redis GEO does this internally (`GEOSEARCH`).

PostGIS — the workhorse

PostGIS extends Postgres with a `geography` / `geometry` type and a **GiST index** (R-tree variant). A nearby query:

```
SELECT id, name FROM shops
WHERE ST_DWithin(location, ST_MakePoint(:lng,:lat)::geography, 5000) -- 5 km
ORDER BY location <-> ST_MakePoint(:lng,:lat)::geography           -- KNN, index-assisted
LIMIT 50;
```

The `<->` KNN operator uses the GiST index to walk the tree in nearest-first order — it does **not** scan all rows. `ST_DWithin` uses a bounding-box pre-filter (cheap) then exact distance (expensive) only on candidates. Use `geography` (treats Earth as a spheroid, meters) for real distances; `geometry` (planar, faster) only if your data is small-area and projected.

Numbers: GiST KNN over ~10M points returns 50 nearest in **single-digit ms**. The cost is tree height (~log of node count). PostGIS scales to tens of millions of points on one node before you need to shard by region (and region-sharding reintroduces the edge problem at shard boundaries).

Redis GEO

Built on a sorted set where the score *is* the 52-bit geohash. `GEOADD` / `GEOSEARCH BYRADIUS` . Blazing fast, in-memory, but it's points-only, no polygons, and you inherit Redis durability/memory trade-offs. Use it for “live driver positions” leaderboards, not for authoritative spatial data.

3. Time-Series Databases (TSDB)

Intuition

Metrics/IoT/telemetry are append-only, timestamp-ordered, and queried by *range over time + grouped by series*. You write 1M points/sec, never update them, and care about “last 5 min” far more than “this point from 90 days ago”. A general DB treats each point as a precious mutable row with a B-tree — catastrophic write amplification. A TSDB treats time as the primary axis.

What makes a TSDB different

```
A "series" = metric name + label set, identified by a series key:  
http_requests_total{method="POST", route="/checkout", status="500"}  
                |_____| the cardinality bomb _____|
```

Storage layout (conceptual):

```
series_id | _____ time-ordered, delta+columnar compressed _____>  
1001      | t0:v t1:v t2:v t3:v ... (one tightly packed column per series)  
1002      | t0:v t1:v t2:v ...
```

Core techniques every TSDB shares:

1. **Columnar / delta-of-delta compression.** Timestamps are near-uniform, so store deltas-of-deltas (often 1 bit each); values use XOR float compression (Facebook **Gorilla** paper — ~1.37 bytes/point vs 16). 10×+ compression is normal.
2. **Downsampling / rollups.** Raw 10 s data → 1 min avg/max → 1 h avg. Old high-resolution data is aggregated and discarded.
3. **Retention policies.** Drop data older than N days *by dropping whole time-partitioned chunks*, not row-by-row **DELETE** (which would be O(rows) and fragment the heap).
4. **Time-partitioned chunks.** Data lands in the “hot” recent chunk (in memory/WAL), older chunks become immutable and compressed.

The #1 failure mode: cardinality explosion

Cardinality = number of distinct series = product of label-value combinations. Add a label like **user_id** or **request_id** and you go from 1,000 series to 100,000,000. Each series needs an index entry and an open in-memory chunk. **This OOM-kills Prometheus** and is the single most common TSDB incident.

```
labels: {method, status, route} -> 4 × 5 × 50 = 1,000 series    ✓  
add label: user_id (1M users)   -> 1,000 × 1M = 1 billion      ✨ OOM
```

Rule: labels must be **bounded, low-cardinality dimensions**. High-cardinality IDs belong in logs/traces, not metric labels.

The big three compared

	Prometheus	InfluxDB	TimescaleDB
Model	Pull (scrapes targets)	Push	Push (it's Postgres)
Storage	Custom TSDB (local)	TSM (LSM-like)	Postgres + hypertable chunks
Query lang	PromQL	Flux / InfluxQL	SQL + time functions
HA / scale	Single-node; federate or use Thanos/Cortex/Mimir for long-term + global	Clustering in enterprise	Postgres replication; sharding harder
Best for	Infra/k8s monitoring, alerting	General IoT/metrics, push pipelines	Teams wanting SQL + joins to relational data
Gotcha	Local storage = not durable long-term; cardinality limits	Storage engine churn across versions	Inherits Postgres write ceiling; cardinality via chunk bloat

Prometheus deliberately keeps each node **simple and ephemeral** — it scrapes, stores ~weeks locally, and you bolt on Thanos/Mimir for global query + object-storage long-term retention. This is a CAP/operational choice: monitoring must keep working even when the rest of the world is on fire, so it avoids distributed dependencies.

Numbers: a single Prometheus node ingests **~1M samples/sec** and holds millions of active series in ~GBs of RAM (each series ~constant overhead). The wall you hit is RAM $\propto$ active series, not disk $\propto$ samples.

4. Analytics / OLAP

OLTP vs OLAP — the foundational split

	OLTP (transactional)	OLAP (analytical)
Query	Few rows by key, read+write	Aggregate many rows, read-mostly
Rows touched	1s–100s	millions–billions
Columns touched	most/all of a row	a few of many
Latency target	ms	seconds–minutes ok
Storage	row-oriented	column-oriented
Example	MySQL, Postgres	BigQuery, Snowflake, ClickHouse, Redshift

Why columnar wins for analytics

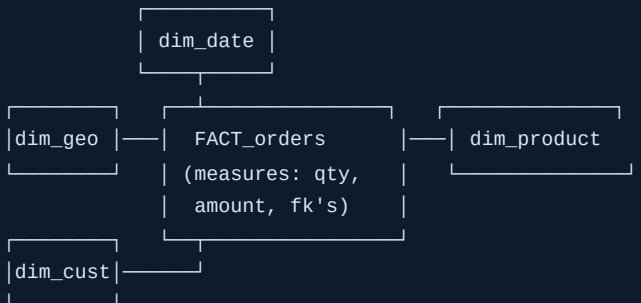
ROW STORE (on disk):	COLUMN STORE (on disk):
[id name region amount date]	[id,id,id,...]
[id name region amount date]	[name,name,...]
[id name region amount date]	[region,region,...] <- read only these
	[amount,amount,...] <- two columns for
"SUM(amount) GROUP BY region"	[date,date,...] SUM..GROUP BY
reads EVERY column of EVERY row.	reads 2 of 5 columns.

Three compounding wins:

1. **I/O reduction** — read only the columns the query needs. A 5-column-aggregate over a 200-column table reads ~2.5% of the data.
2. **Compression** — a column is one data type with low entropy → run-length encoding, dictionary encoding, bit-packing give 10×+ ratios. Better compression = even less I/O.
3. **Vectorized / SIMD execution** — process a column as a contiguous array in CPU cache, one SIMD instruction over many values. (C-Store/Vertica and MonetDB pioneered this.)

The catch: **columnar is terrible at OLTP**. Inserting one row touches every column file; point updates are expensive. So OLAP stores are append/bulk-load oriented, often immutable+merge (ClickHouse **MergeTree**, an LSM cousin). Don't point a transactional app at a warehouse.

Schema: star vs snowflake



- **Fact table** — the events/measures (one row per order line), huge, numeric, foreign keys to dims.
- **Dimension tables** — the descriptive context (product, customer, date, geography), small, wide.
- **Star** = dims are denormalized/flat (fast, fewer joins). **Snowflake** = dims normalized into sub-dims (less redundancy, more joins). Warehouses overwhelmingly prefer **star** — storage is cheap, joins are the cost.
- **Slowly Changing Dimensions (SCD Type 2)**: when a customer moves region, you don't overwrite — you add a new dim row with validity dates, so historical facts join to the region that was true *at the time*. Forgetting this silently corrupts historical reports (a real war story).

Warehouse vs Lake vs Lakehouse

	Data Warehouse	Data Lake	Lakehouse
Storage	Proprietary columnar, schema-on-write	Object store (S3), raw files, schema-on-read	Object store + open table format (Iceberg/Delta/Hudi)
Data	Structured, modeled	Anything (JSON, parquet, images, logs)	Structured + semi-structured, governed
Strength	Fast SQL, governance, BI	Cheap, flexible, ML/raw data	Warehouse SQL+ACID <i>on</i> lake storage
Weakness	Expensive, rigid, hard for ML	"Data swamp", no ACID, slow ad-hoc	Newer, operational complexity
Examples	Snowflake, BigQuery, Redshift	S3 + Hive/Athena	Databricks (Delta), Iceberg + Trino

The lakehouse is the current convergence: keep data in cheap S3 **Parquet** files, but add a metadata/transaction layer (**Apache Iceberg / Delta Lake / Hudi**) that gives ACID commits, schema evolution, time-travel, and **MERGE** — so one copy of data serves both BI SQL engines and ML, no copy-into-warehouse step.

5. Decision guide — pick the right store

```
Is the query...
├ "match/rank text, fuzzy, faceted"      -> Search engine (Elasticsearch/Lucene)
├ "near a point / inside a region"       -> Geospatial (PostGIS, Redis GEO, S2/H3)
├ "metrics over time, downsampled"       -> TSDB (Prometheus/Timescale/Influx)
├ "aggregate billions of rows, few cols" -> Columnar OLAP (ClickHouse/BigQuery)
└ "fetch/update a few rows by key, ACID" -> stay on Postgres/MySQL ✓
```

The senior move is usually **polyglot persistence with the relational DB as source of truth + a derived index**. E.g. orders live in Postgres; a CDC stream (Debezium → Kafka) materializes them into Elasticsearch for search and ClickHouse for analytics. This keeps writes ACID and reads specialized — at the cost of eventual consistency between the source and the derived stores (see **09-...** and the CDC discussion in **15-...**).

6. Common pitfalls / war stories

- **Analyzer mismatch.** Index with stemming + synonyms, query without them → “running shoes” returns nothing. *Always* use the same (or a compatible search) analyzer at query time. Symptom: exact-match works, “obvious” matches don’t.
- **Using Elasticsearch as a primary database.** It’s near-real-time (1 s refresh), has no real transactions, and silently drops/merges on segment merges. People lose data when ES is the only store. Treat it as a

derived index, reindex-able from a durable source.

- **Mapping explosion / dynamic mapping.** Letting ES auto-create a field per unique JSON key (e.g. `attributes.<sku>`) explodes the mapping and the cluster state. Use `flattened` type or explicit mappings.
- **Geohash edge bug.** Querying only the matching cell misses neighbors across the seam. Always include the 8 adjacent cells + post-filter by Haversine.
- **geometry vs geography in PostGIS.** Computing distance in `geometry` (planar, degrees) gives nonsense kilometers. Use `geography` for Earth-surface meters.
- **Cardinality bomb in metrics.** Adding `user_id / trace_id / url-with-querystring` as a Prometheus label = millions of series = OOM. Keep labels bounded; push IDs to logs/traces.
- **Deleting old TSDB data with DELETE.** Row-by-row deletes fragment and choke the engine. Use retention/chunk-drop. Same lesson in Postgres partitioning: `DROP PARTITION`, don't `DELETE`.
- **Running analytics on the OLTP replica.** A `GROUP BY` over the orders table on a row store scans everything, blows the buffer cache, and tanks transactional latency. ETL/CDC into a columnar store instead.
- **Over-sharding Elasticsearch.** “More shards = faster” is wrong; each shard is a Lucene index with fixed overhead, and scatter-gather fan-out grows. Size shards to 10–50 GB.
- **Forgetting SCD Type 2.** Overwriting a dimension corrupts every historical report that depended on the old value. Version dimension rows.

Case Study: Uber's H3 Geospatial Index

The problem

Uber's marketplace is one continuous spatial-matching engine: at any instant it tracks **millions of live drivers**, and at peak handles on the order of **tens of millions of trips per day** — well over a million ride requests per hour globally. Every one of those requests is the geospatial query from §2: “*of all the drivers near this rider right now, which are the closest?*” — answered in tens of milliseconds, continuously, for every city on Earth.

Two harder layers sit on top of plain proximity. **Surge pricing** needs to aggregate supply (drivers) and demand (open requests) into pricing *zones* and recompute multipliers every few minutes. **Marketplace analysis** needs to compare “how busy is this area” across cities — but a fixed lat/lng grid has cells of wildly different real-world area near the poles vs. the equator, so a zone in London isn't comparable to one in Lima. They needed a spatial unit that is (a) fast to look up, (b) hierarchical for roll-ups, and © **uniform in area** so cross-city aggregation is honest.

Why a relational DB fails here

This is the exact §2 “wrong tool” story. A row in Postgres lives in a B-tree, which is one-dimensional — it sorts on a single key. A nearby query is irreducibly 2-D: a driver close in latitude can be far in longitude. The naive fixes all break at Uber's scale:

- **Two B-tree indexes (one on lat, one on lng) intersected** — the §0 table’s “2-D range needs 2 indexes intersected” trap. Each index is unselective (a thin latitude band still spans the whole planet’s longitude), so the intersection scans enormous candidate sets.
- **PostGIS GiST/R-tree on one node** — genuinely good (the §2 workhorse), but §2’s own number is the ceiling: “tens of millions of points on one node before you need to shard by region — *and region-sharding reintroduces the edge problem at shard boundaries.*” Uber’s write rate (every driver pinging GPS every few seconds = millions of *updates* per second) and global footprint blow past a single spatial node.
- **SELECT ... WHERE distance < 5km** scanning every driver — a full scan per request, millions of times an hour. Dead on arrival.

So Uber did the §5 “recognize the shape, name the structure, justify the trade-off” move and reached for a **specialized spatial index** instead of a general DB.

Applying the module’s concepts: H3

H3 is precisely the “**S2/H3**” row of the §2 data-structures table — “project sphere onto a polyhedron, use a hierarchical curve, fixes geohash’s poles/edge distortion.” The key implementation choices map straight onto §2:

- **Flatten 2-D into 1-D while preserving locality** (§2 intuition). H3 projects the globe onto an icosahedron (20 triangular faces) and tiles each face with **hexagons**, addressed by a 64-bit **H3Index**. `latLngToCell(lat, lng, res)` turns any coordinate into one integer key — the same trick as geohash, but on a better surface.
- **Hexagons over squares** — H3’s deliberate divergence from geohash/S2 squares. A hexagon has only **one class of neighbor** (all 6 share an edge, all roughly equidistant from center), whereas a square has edge-neighbors *and* corner-neighbors at different distances. This makes “expand the search ring” and flow/movement modeling clean.
- **Hierarchy for roll-ups** — 16 resolutions (res 0 ≈ continent, res 9 ≈ ~0.1 km² city block). `cellToParent` / `cellToChildren` let surge aggregate fine cells into coarse zones, the spatial analogue of TSDB downsampling (§3).
- **The edge problem, handled by design** (§2’s “classic interview trap”). A nearby search isn’t “same prefix”; it’s `gridDisk(cell, k)` — return this hex plus all hexes within `k` rings — then post-filter by true Haversine distance. This is §2’s “query your cell **plus its neighbors**, then post-filter by true distance,” made first-class.

Data flow for “find nearby drivers”:

```
Driver GPS ping → latLngToCell(lat, lng, res9) → H3Index (uint64)
                                     |
                                     | (sharded by cell)
in-memory map: H3Index → {driverIds...}
                                     ▼

Rider request → riderCell = latLngToCell(rider)
               candidateCells = gridDisk(riderCell, k=2)  ← cell + rings
               drivers = union(map[c] for c in candidateCells)
               return top-N by Haversine(rider, driver)    ← exact post-filter
```

Because the key is just a `uint64`, the index is **not a database** (the §2 caveat: “library, not a DB”) — Uber shards driver-location maps by H3 cell across an in-memory fleet, exactly the §5 “specialized index, relational DB as source of truth” polyglot pattern. Trips and payments stay in the durable store; the hot spatial layer is derived and rebuildable.

The trade-off they accepted

H3’s headline win — **uniform-area cells** for honest cross-city comparison — is bought with a **small, permanent geometric lie**: you *cannot* perfectly tile a sphere with hexagons. Every H3 grid contains exactly **12 pentagons** (inherited from the icosahedron’s 12 vertices), and parent/child hexagons don’t nest perfectly (a child isn’t fully contained by one parent — “containment” is approximate). Uber accepted approximate hierarchy and 12 special-case cells (placed over oceans, where almost no rides happen) in exchange for near-uniform area and clean neighbor math. For a marketplace that cares about *relative* density and fast neighbor expansion, approximate nesting is a fine price; for legal/cadastral boundary work it would be the wrong choice (use PostGIS polygons, §2).

Results

- Nearby-driver lookups become a `uint64` map hit + a bounded `gridDisk` expansion — **single-digit-millisecond** matching, holding up under millions of location updates per second.
- One uniform grid lets surge and analytics compare any two areas worldwide without per-city grid tuning. H3 is open-sourced and now used well beyond Uber.

Lessons

- **Choose the index to fit the query shape, not the DB you already run.** A 2-D proximity query on a 1-D B-tree is the canonical mismatch; reach for a spatial index even if it means leaving your relational comfort zone.
- **The grid’s geometry is a design parameter.** Hexagons vs. squares, uniform vs. variable area, perfect vs. approximate nesting — each is a real trade. Uber traded perfect hierarchy for uniform, comparable cells.
- **Edge/seam handling is not optional.** “Same cell” never means “all neighbors”; always expand by neighbor rings (`gridDisk`) and post-filter by true distance.
- **Keep the spatial layer derived.** The fast in-memory H3 index is rebuildable from the durable source of truth — the §5 polyglot-persistence pattern, so you get specialized reads without risking ACID writes.

7. Test yourself

1. Why does the same analyzer have to run at both index and query time? Give a concrete failure if it doesn’t. *Hint: stemming/case/synonyms must align or the query term never matches the indexed term.*
2. Two BM25 improvements over TF-IDF — name them and the parameter that controls each. *Hint: `k1` (TF saturation), `b` (length normalization).*
3. A user posts data, then immediately re-queries Elasticsearch and doesn’t see it. Why, and what’s the fix? *Hint: refresh interval (~1 s, near-real-time); read-your-write from the source DB, don’t lower refresh*

globally.

4. Explain the geohash “edge problem” and the standard query fix. *Hint: boundary points get different prefixes; query the cell + 8 neighbors, post-filter by true distance.*
5. You add a `request_id` label to a Prometheus metric and the server OOMs. What happened and what’s the rule? *Hint: cardinality = product of label values; high-cardinality IDs don’t belong in labels.*
6. Why is a column store 10–40× faster for `SUM(amount) GROUP BY region` but unusable for the order-checkout write path? *Hint: reads only needed columns + compression + SIMD; but one row insert touches every column file.*
7. When would you choose a lakehouse (Iceberg/Delta) over a classic warehouse? *Hint: cheap S3 storage + ACID + one copy serving both BI SQL and ML, with schema evolution/time-travel.*
8. What’s wrong with overwriting a `region` value in a dimension table, and what’s the pattern? *Hint: historical facts re-attribute to the new region; SCD Type 2 with validity dates.*

8. Further reading

- **DDIA (Kleppmann)** — Ch. 3 “Storage and Retrieval” (B-tree vs LSM, column-oriented storage, data warehousing, star/snowflake schemas). The single best grounding for this module.
- **“Finding the Needle”** — Lucene/inverted index internals; and the *Elasticsearch: The Definitive Guide* sections on analysis, relevance, and shards.
- **Robertson & Zaragoza, “The Probabilistic Relevance Framework: BM25 and Beyond.”**
- **Pelkonen et al., “Gorilla: A Fast, Scalable, In-Memory Time Series Database”** (Facebook, VLDB 2015) — delta-of-delta + XOR float compression.
- **Stonebraker et al., “C-Store: A Column-oriented DBMS”** (VLDB 2005) — the columnar OLAP foundation.
- **Google S2 Geometry library docs** and **Uber H3 docs** — hierarchical spherical indexing.
- **PostGIS official docs** — `ST_DWithin`, GiST indexing, `geography` vs `geometry`, KNN `<->`.
- **Prometheus docs** — “Storage” and “Naming/labels (cardinality)”; **Thanos/Cortex/Mimir** for long-term/global.
- **Apache Iceberg / Delta Lake** specifications — open table formats and the lakehouse pattern.

20 — Case Studies & the System Design Interview Framework

Source: [readme/research/system-design/20-case-studies-and-interview-framework.md](#)

IN SHORT — KEY TAKEAWAYS

This document is a study guide for answering “design a big system” questions in a software engineering interview. It gives you a step-by-step method to follow, then walks through ten real-world examples — like how WhatsApp, Uber, Netflix, and Stripe work under the hood — so you can practise applying the method. It matters because interviews at top tech companies almost always include one of these problems, and most people fail by jumping to answers before understanding the question.

The main parts explained simply:

- **The 7-step method** — A fixed checklist to run through every time: first agree on what the system must do, then estimate how big it will be, define the rules (API), pick a data store, draw the big picture, dive deep into two or three tricky parts, and finally name what will break and why. Following this order stops you from jumping to solutions too soon.
- **Doing the maths** — Before drawing anything, you work out how many requests per second the system will get and how much data it will store. This one step tells you whether you need one server or a thousand.
- **URL shortener (like bit.ly)** — How to turn a long web address into a short one and handle millions of clicks per day without a single bottleneck.
- **Rate limiter** — A door-bouncer for APIs that counts how many requests a user has made and turns them away if they go too fast. The tricky part is making this work across many servers at once.
- **News feed (like Twitter/Instagram)** — When someone posts, do you push a copy to every follower immediately, or wait until followers ask? The right answer is: both, depending on how popular the account is.
- **Chat (like WhatsApp)** — How messages travel in real time, get stored for offline users, and arrive in the right order without duplicates.
- **Nearby drivers (like Uber)** — How to find the closest driver in under a second when millions of drivers are moving and sending location updates every few seconds.
- **File sync (like Dropbox)** — Splitting files into small chunks so only the changed part re-uploads, and handling conflicts when two devices edit the same file offline.
- **Video streaming (like YouTube/Netflix)** — Storing video in many quality levels, serving it from servers close to the viewer, and letting the player automatically switch quality based on internet speed.
- **Notifications (push/email/SMS)** — Sending alerts through many channels at scale without sending the same message twice or losing important ones.
- **Distributed cache** — A very fast memory store that sits in front of a database. The key lesson is how to split the cache across many machines without causing a crash when you add or remove one.
- **Payments** — Never double-charge. Every payment is a series of small steps recorded in a permanent log, and every retry must be safe to run again without charging twice.
- **Common mistakes to avoid** — Picking a technology before doing the maths, ignoring skew (one user or one piece of data gets far more traffic than others), and not planning for what happens when

any part of the system goes down.

What to do with this: Memorise the 7-step checklist and practise talking through it out loud. Then pick two or three of the case studies and sketch them from memory — without looking — until you can explain the one or two most important design decisions for each and why those choices were made.

What you'll learn: A reusable, repeatable framework for attacking any system-design problem (whiteboard or production), and then ten fully worked compact designs — URL shortener, distributed rate limiter, news feed, chat, ride-sharing geo, file sync, video streaming, notifications, distributed cache, and payments — each reduced to its requirements, 2–3 load-bearing decisions, the failure modes, and what an interviewer (or an incident) will probe. By the end you should be able to walk into a 45-minute design round and *control the conversation* instead of free-associating.

Prerequisites: This is the capstone. Read the earlier modules first — especially [09-cap-pacelc...md](#) (consistency trade-offs), [10-replication-and-partitioning...md](#) (sharding), [12-caching...md](#), [14-message-queues-and-streaming...md](#) (Kafka/log), [15-consensus...md](#) (Raft/leader election), and [17-rate-limiting-and-load-shedding...md](#). The case studies *compose* those primitives; this module assumes you know each one.

1. The framework: a 7-step loop you run every time

Most candidates fail not from ignorance but from **lack of structure** — they jump to “I’ll use Kafka” before anyone agreed what the system does. The fix is a fixed loop. Treat the interviewer as a product manager *and* a co-architect: drive, but check in.

```
| 1. Requirements  → functional + non-functional + constraints |
| 2. Estimation    → QPS, storage, bandwidth (back-of-envelope) |
| 3. API           → the contract (endpoints, params, returns) |
| 4. Data model    → entities, access patterns, choose store |
| 5. High-level    → boxes + arrows, request path end-to-end |
| 6. Deep dives    → 2-3 components the interviewer cares about |
| 7. Bottlenecks   → scale, failures, trade-offs, what breaks |
```

↑ loop back whenever a constraint changes the design ↑

Time budget for a 45-min round: ~5 min requirements, ~5 min estimation, ~5 min API + data model, ~10 min high-level, ~15 min deep dives, ~5 min wrap-up on bottlenecks. The single biggest mistake is spending 25 minutes on requirements/estimation and never drawing a deep dive.

Step 1 — Requirements

Separate **functional** (“user can shorten a URL”) from **non-functional** (“99.9% availability, p99 < 50ms read”). The non-functional ones *are* the design — read-heavy vs write-heavy, consistency vs availability, latency budget, and durability requirements pick your architecture far more than the features do. Always ask: read:write ratio, expected scale (DAU), latency SLO, and consistency tolerance.

Step 2 — Estimation (the numbers you must memorize)

Quantity	Value	Use
Seconds/day	~86,400 (~10 ⁵)	DAU × actions ÷ 10 ⁵ = avg QPS
Peak factor	2–10× average	size for peak, not average
L1 cache ref	~1 ns	—
Main memory ref	~100 ns	RAM is ~100× cache
SSD random read	~16–100 µs	—
Network round trip (same DC)	~0.5 ms	—
Network RTT (cross-continent)	~80–150 ms	speed of light floor
Disk seek (HDD)	~10 ms	avoid in hot path
1 KB over 1 Gbps	~10 µs	bandwidth math

Worked example (always show the math): 100M DAU, each posts 2× and reads 100× → writes = 200M/day ÷ 86,400 ≈ **2,300 write QPS** (peak ×5 ≈ 11.5K). Reads = 10B/day ÷ 86,400 ≈ **115K read QPS**. That 50:1 read skew immediately says: cache aggressively, replicate reads, optimize the read path. Storage: 200M writes × 1 KB × 365 × 5 yr ≈ **365 TB** — needs sharding, not one box.

Steps 3–4 — API and data model

Keep the API boringly RESTful unless there's a reason (gRPC for internal high-throughput, WebSocket for push). The data model's job is to **match the access pattern**: if you query by key, a KV store beats a relational join. State the access pattern *out loud*, then pick the store (see the per-case tables below).

Steps 5–7

Draw the request path as boxes and arrows; then the interviewer will pull on a thread (“what if a shard is hot?”). Your deep dives and bottleneck discussion are where senior signal lives — show you know *what breaks* and *the trade-off you chose*.

Senior tell: name the trade-off explicitly. “I’ll use fan-out-on-write for the common case but fall back to fan-out-on-read for celebrities — hybrid — because pure write-fanout blows up at 100M followers.” That sentence signals staff-level reasoning.

2. URL shortener (TinyURL / bit.ly)

Requirements: create short → long mapping; redirect; ~100:1 read:write; low-latency redirect (p99 < 50ms); links rarely deleted. Optional: custom aliases, analytics, expiry.

Estimation: 100M new URLs/day → ~1,160 write QPS, ~116K read QPS. $100M \times 365 \times 5yr \times 500 B \approx 91$ TB.

Critical decisions:

1. **Short-code generation.** Three options:

Approach	Pro	Con	When
Hash(long URL) + truncate	stateless, dedups identical URLs	collisions need retry; not random	dedup matters
Random base62 (7 chars)	$62^7 \approx 3.5T$ space, simple	collision check (read before write)	default
Counter + base62 encode	no collisions, monotonic	needs distributed ID (see below)	high write rate

2. **Distributed ID without a single bottleneck.** A central auto-increment is a SPOF. Use **ranged allocation**: each app server grabs a block of 1,000 IDs from a coordinator (ZooKeeper/etcd or a `counters` row with `SELECT FOR UPDATE`), then hands them out locally. Or Twitter **Snowflake** (timestamp+machine+seq). Base62-encode the ID → short code.

3. **Read path = cache.** Redis/Memcached in front of the KV store (DynamoDB/Cassandra). 100:1 read skew → ~95%+ hit rate. Redirect with **301 (cached by browser, kills your analytics)** vs **302 (every hit reaches you, enables analytics)** — classic trade-off interviewers love.

Bottlenecks / what they probe: collision handling under concurrency; cache stampede on a viral link (use request coalescing); hot-key for one mega-popular short code (replicate that key); 301 vs 302 reasoning.

3. Distributed rate limiter

Requirements: limit N requests / window / key (user/IP/API-key); low added latency (<1ms ideal); accurate enough; works across a fleet of stateless servers. (Deep treatment in `17-rate-limiting-and-load-shedding...md`.)

Critical decisions:

1. **Algorithm.**

Algorithm	Accuracy	Memory	Burst behavior
Fixed window counter	edge bursts (2× at boundary)	tiny	allows 2N at window edge
Sliding window log	exact	O(N) per key — expensive	precise
Sliding window counter	good approx	tiny	smooths edges
Token bucket	allows controlled bursts	tiny	best for APIs
Leaky bucket	smooths to constant rate	tiny	shaping/queueing

Default to **token bucket** (refill rate + capacity) — it's what Stripe/AWS API Gateway use. 2. **Where does the counter live?** Per-node local counters are fast but each node only sees 1/N of traffic → the global limit

is $N \times$ too loose. Centralize in **Redis** (atomic `INCR + EXPIRE` , or a Lua script for token bucket to make it atomic). Trade-off: Redis is now in your hot path (~0.5ms RTT) and a SPOF — mitigate with local approximate counters + periodic sync, accepting slight over-admission.

Bottlenecks / probes: the boundary-burst bug of fixed windows; race conditions (must be atomic — Lua/ `INCRBY` not read-modify-write); Redis failure mode (fail-open vs fail-closed — usually **fail-open** so a Redis outage doesn't take down your whole API); clock skew across nodes; distributed accuracy vs latency.

4. News feed (Twitter/Instagram timeline)

Requirements: users follow others; home timeline = recent posts from followees, newest-first; read-heavy; low feed-load latency; eventual consistency OK (a post appearing 2s late is fine).

Critical decision — fan-out on write vs read:

FAN-OUT ON WRITE (push)	FAN-OUT ON READ (pull)
post → write to each follower's precomputed feed cache	post → store once
read → $O(1)$ read own feed	read → gather from all followees at query time
fast reads, costly writes	cheap writes, costly reads
celebrity = 100M cache writes!	fine for celebrities

	Fan-out on write	Fan-out on read	Hybrid (real answer)
Read latency	very low	high (scatter-gather)	low
Write amplification	huge for big accounts	none	bounded
Best for	most users	celebrities	everyone

The senior answer is hybrid: push for normal users; for accounts with millions of followers, *don't* fan out — pull their recent posts at read time and merge into the precomputed feed. Twitter's "timeline" famously evolved exactly this way.

Mechanics: feeds live in Redis lists/sorted-sets (by timestamp/score), capped at ~800 entries. Posts in Cassandra/sharded store. Write path enqueues fan-out jobs to Kafka → workers update follower feeds asynchronously.

Bottlenecks / probes: celebrity fan-out (the headline question — *always* raise it); ranking (chronological vs ML-ranked changes everything); thundering-herd on a viral post; feed staleness; storage of duplicated feed entries.

5. Chat / messaging (WhatsApp)

Requirements: 1:1 and group messaging; real-time delivery; delivery + read receipts; online presence; ordering within a conversation; offline message storage; ~billions of messages/day.

Critical decisions:

1. **Connection model.** Need server → client *push*, so **persistent WebSocket** (or MQTT, which WhatsApp uses — low overhead, great on mobile). One server holds millions of long-lived connections. A **session/connection registry** (Redis: `user_id → gateway_server`) lets sender's gateway route to recipient's gateway.

```
A —ws→ Gateway1 → [route via Redis: B is on Gateway2]
                        |
                        v
B ←ws← Gateway2 ← Message Service (persist → ack)
```

2. **Delivery + ordering.** Persist before ack ("stored"). Per-conversation **sequence numbers** (not wall-clock — clocks skew) give ordering. Offline → store and push via APNs/FCM when the device reconnects. Receipts are just more messages (sent/delivered/read state).
3. **Group fan-out.** Small groups: write to each member. Large groups (WhatsApp caps ~1024) bounds the blast radius — that cap *is* the design decision.

Bottlenecks / probes: connection state at scale (graceful failover when a gateway dies — clients reconnect, re-register); exactly-once vs at-least-once delivery (you get at-least-once + idempotent message IDs → client dedups); ordering guarantees; end-to-end encryption (Signal protocol — server can't read, so no server-side search); presence at billions (heartbeats are expensive — debounce/approximate).

6. Ride-sharing — nearby drivers (Uber/Lyft)

Requirements: drivers send location every few seconds; rider requests → find nearby available drivers fast; match; low latency on "find nearby" (<200ms); millions of location updates/sec.

Critical decision — geospatial indexing: you can't `WHERE distance < r` over millions of points per query. Encode location into a **searchable cell**:

Technique	Idea	Used by
Geohash	recursively bisect lat/lon → string prefix = proximity	Redis <code>GEO*</code> , many
Quadtree	tree of subdivided quadrants, denser where busier	classic
Google S2	sphere → Hilbert curve cells (handles poles, no edge distortion)	Google, Uber early
Uber H3	hexagonal grid (uniform neighbor distance)	Uber today

Nearby query = “find all drivers whose cell $\in$ {my cell + neighbors}.” Hexagons (H3) win because all 6 neighbors are equidistant, unlike squares (diagonal vs edge).

Second decision — the write storm. Millions of drivers $\times$ update every 4s = millions of writes/sec. Don’t hammer a DB. Keep the live location index **in memory** (sharded by region), updated continuously; persist a sampled trail separately. Shard the geo index **by region**, not by driver ID, so a nearby query hits one shard.

Bottlenecks / probes: hotspot cities (downtown SF at rush hour — one cell is overloaded; subdivide adaptively / quadtree depth); update frequency vs accuracy vs cost; matching consistency (two riders shouldn’t get the same driver — short-lived lock / dispatch queue); cell-boundary edge case (driver just across a cell line is “nearby” but in a different cell — must query neighbors).

7. File storage & sync (Dropbox / Google Drive)

Requirements: upload/download files; sync across devices; share; versioning; handle large files; bandwidth-efficient sync; ~strong-ish consistency on metadata.

Critical decisions:

1. **Chunking + dedup.** Split files into ~4 MB **content-addressed chunks** (hash = ID). Benefits: resumable uploads, parallel transfer, and **dedup** (same chunk stored once globally — huge savings). Edit a 1 GB file $\rightarrow$ only changed chunks re-upload (delta sync).
2. **Split metadata from blobs.** Metadata (file tree, chunk list, versions, permissions) in a **strongly-consistent** store (sharded SQL / Spanner). Chunks in **object storage** (S3) behind a CDN. The metadata service is the brain; blob storage is the muscle.

```
Client -chunks-> Block servers -> S3 (content-addressed)
|
└─metadata-> Metadata DB (file tree, version vector)
      |
      other device ← notification (poll/long-poll/push) → pull new chunks
```

3. **Sync protocol.** Client watches local FS, computes diffs, pushes changed chunks + new metadata version. Other devices get notified and pull. **Conflict resolution** when two devices edit offline: version vectors detect the conflict; resolve by keeping both (“file (conflicted copy)”) — Dropbox’s actual choice, because silent auto-merge of binaries is dangerous.

Bottlenecks / probes: dedup hashing cost; small-file overhead (chunking wastes on tiny files); conflict resolution semantics; consistency of metadata vs eventually-consistent blobs; bandwidth (delta sync, compression); large-file resumability.

8. Video streaming (YouTube / Netflix)

Requirements: upload + transcode; stream to any device/bandwidth; massive read scale; low startup latency + no rebuffering; store petabytes.

Critical decisions:

1. **Transcoding pipeline (write path).** Upload → object storage → **fan-out transcode jobs** (queue + worker fleet) producing many resolutions/codecs (240p...4K, H.264/VP9/AV1) and **segmenting** into ~2–10s chunks for adaptive streaming. Transcoding is embarrassingly parallel and the expensive part — split each video into segments and transcode segments in parallel across the fleet.
2. **Adaptive Bitrate Streaming (read path). HLS/DASH:** the player fetches a manifest listing segments at multiple bitrates, then *picks the bitrate per segment* based on measured bandwidth — drop to 480p mid-stream rather than rebuffer. This client-side logic is the heart of “no buffering.”
3. **CDN is the whole game.** 99%+ of bytes served from edge caches near users (or ISP-embedded boxes — **Netflix Open Connect**). Origin only sees cache misses. This is what makes petabyte-scale, low-latency streaming economical.

```
Upload → S3 → [transcode workers] → segments+manifests → S3 (origin)
                                     |
Viewer player ← ABR (HLS/DASH) ← CDN edge ← origin (on miss)
```

Bottlenecks / probes: transcode throughput & cost; CDN cache hit ratio and warming for a brand-new viral video (origin shielding / tiered cache); ABR algorithm trade-offs (aggressive quality vs rebuffer risk); storage tiering (cold archive for the long tail); thumbnail/preview generation.

9. Notification system (push / email / SMS)

Requirements: send across channels (push/email/SMS/in-app); high throughput, bursty; user preferences & opt-outs; retries; no duplicate spam; some priority tiers.

Critical decisions:

1. **Queue-decoupled pipeline.** Producers publish events → a **message queue** (Kafka/SQS) → channel-specific workers call providers (APNs/FCM, SES/SendGrid, Twilio). Decoupling absorbs bursts and isolates a slow provider from the rest.

```
Services → Notification API → Kafka → [push worker] → APNs/FCM
                                     |→ [email worker] → SES
                                     |→ [sms worker] → Twilio
preferences + dedup + rate-limit check
```

2. **Idempotency + dedup.** At-least-once queues mean retries → duplicates. Attach an idempotency/notification key; check a dedup store (Redis with TTL) before sending. Critical: an SMS billed twice or a duplicate “your code is X” is a real incident.

3. **Preferences, throttling, DLQ.** Respect opt-outs and per-user rate caps *before* sending. Failed sends → retry with backoff → **dead-letter queue** after max attempts (never silently drop a payment receipt).

Bottlenecks / probes: third-party provider failures/rate limits (circuit breaker + failover provider); dedup correctness; priority (a 2FA code must jump the marketing queue — separate high-priority topic); fan-out for “notify all users” (batch + throttle, don’t melt the providers).

10. Distributed cache (Memcached / Redis Cluster)

Requirements: sub-ms reads; scale beyond one machine’s RAM; high hit ratio; tolerate node failures; (see `12-caching...md`).

Critical decisions:

1. **Sharding via consistent hashing.** Naive `hash(key) % N` remaps *almost everything* when N changes → mass cache miss → DB meltdown. **Consistent hashing** (hash ring + virtual nodes) moves only $\sim 1/N$ keys when a node joins/leaves. This is *the* answer interviewers want.

```
Ring: ...-[vnode A]-key1-[vnode B]-key2-[vnode C]-... (wrap)
key → first vnode clockwise. Add node D → only keys
between its predecessor and D move. Virtual nodes even out load.
```

2. **Eviction + invalidation.** LRU/LFU eviction; write policy (write-through vs write-back vs **cache-aside**, the default). Invalidation is “one of the two hard problems” — prefer short TTL + cache-aside over trying to keep cache perfectly in sync.
3. **Failure & stampede protection.** Replicas for availability (Redis Cluster: primary+replica per shard). **Cache stampede** when a hot key expires → thousands of concurrent DB reads; fix with request coalescing (single-flight), staggered TTL (jitter), or background refresh.

Bottlenecks / probes: hot key (one key > one node’s capacity — replicate the key or client-side cache it); rebalancing cost; consistency between cache and source of truth; thundering herd; memory fragmentation; cold-start warming.

11. Payment system

Requirements: charge/refund; **correctness over availability** (never double-charge, never lose money); idempotency; audit trail; integrate external gateways; reconciliation; PCI scope.

Critical decisions:

1. **Idempotency is non-negotiable.** Network retries must not double-charge. Client sends an **idempotency key**; server records (key → result) and returns the *same* result on replay. Stripe’s API is built around this — interviewers expect you to raise it unprompted.

- **Ignoring the “what if it fails” question.** Every box you draw can crash. If you can’t say what happens when each one dies, you haven’t finished the design.
-

13. Test yourself

1. You compute 50:1 read:write at 100M DAU. Name three concrete design moves that ratio dictates. (*Hint: cache, read replicas, optimize/precompute the read path.*)
 2. Why does a URL shortener that uses `hash(url) % num_shards` fall over when you add a shard, and what fixes it? (*Hint: mass remap → consistent hashing.*)
 3. When is fan-out-on-read *better* than fan-out-on-write, and what’s the hybrid rule? (*Hint: celebrities; push for normal users, pull for huge accounts.*)
 4. A token-bucket rate limiter uses Redis `GET` then `SET`. What’s the bug and the fix? (*Hint: read-modify-write race → atomic Lua/ `INCR`.*)
 5. Why split metadata from blobs in Dropbox, and which gets the strongly-consistent store? (*Hint: brain vs muscle; metadata = SQL/Spanner, chunks = S3.*)
 6. Your payment service writes to its DB then calls the gateway; the process crashes in between. What pattern prevents the resulting inconsistency? (*Hint: transactional outbox / saga + reconciliation.*)
 7. Why does Uber prefer H3 hexagons over a square geohash grid for nearby-driver queries? (*Hint: equidistant neighbors; uniform distance vs square’s diagonal-vs-edge problem.*)
 8. A hot cache key gets more traffic than a single node can serve. Consistent hashing doesn’t help — why, and what does? (*Hint: it’s one key, not key distribution; replicate the key / client-side cache.*)
-

14. Further reading

- **Alex Xu, *System Design Interview Vol. 1 & 2*** — the canonical compact walkthroughs for most of these case studies.
- **Martin Kleppmann, *Designing Data-Intensive Applications (DDIA)*** — Ch. 5–6 (replication/partitioning), Ch. 7 (transactions), Ch. 9 (consistency/consensus), Ch. 11 (stream processing/outbox). The theory under every case here.
- **DynamoDB paper** (Amazon, 2007) — consistent hashing, quorums, eventual consistency.
- **Google Spanner paper** — strong consistency + TrueTime, relevant to payments/metadata.
- **Twitter “Timelines at Scale” talks** — fan-out hybrid in production.
- **Uber Engineering blog: H3** — hexagonal geospatial indexing.
- **Stripe API docs: Idempotent Requests** — the reference idempotency-key design.
- **Netflix Open Connect docs** — CDN/edge strategy for video at scale.
- **“Jeff Dean — Numbers Everyone Should Know”** — the latency table to memorize.
- Sibling modules: `09-cap-pacelc...md`, `10-replication-and-partitioning...md`, `12-caching...md`, `14-message-queues-and-streaming...md`, `15-consensus...md`, `17-rate-limiting-`

and-load-shedding...md .