

BE SOFTWARE ARCHITECTURE WISE · CHAPTER 6

Architectural Styles II: Distributed Systems

This chapter takes architecture across the network. It explains when distributing a system is worth its cost, the fallacies that make it harder than it looks, and the main distributed styles: service-based architecture, service-oriented architecture, microservices, event-driven, serverless and space-based architecture. It then covers the mechanisms they all depend on: synchronous and asynchronous communication and API design, consistency models and the CAP theorem, and distributed transactions with sagas and the outbox pattern. It is written for developers and architects who are designing or reviewing systems made of many services, and it is precise about failure, delivery guarantees and consistency.

bewiseapps.com/software-architecture-wise/distributed-systems/ · Version 2026-11-03. Educational information only, not professional advice.

AT A GLANCE

You will learn to

- Explain the legitimate reasons to distribute a system and the costs each reason brings.
- Describe the eight fallacies of distributed computing and partial failure, and design responses to them.
- Calculate the availability and latency of a chain of synchronous calls and propose changes that improve them.
- Compare service-based architecture, classic SOA and microservices, and choose between them for a given organisation.
- Decide service granularity by weighing disintegrators against integrators, and recognise the distributed monolith.
- Distinguish events, commands and messages, broker and mediator topologies, and Fowler's four event patterns.
- Explain delivery guarantees, why exactly-once needs careful definition, and how idempotent consumers work.
- Choose communication styles, consistency levels and transaction patterns (sagas, outbox) for distributed workflows.

Before you start

Chapters 0 to 5, especially availability in series and parallel (Chapter 2, Section 2.5), coupling and Conway's law (Chapter 3) and the architecture quantum, modular monolith and bounded contexts (Chapter 5).

Time

About 3½ hours to read; 45 minutes for the test.

Key terms

distributed system · partial failure · fallacies of distributed computing · temporal coupling · service-based architecture · enterprise service bus · microservices · independent deployability · database per service · event-driven architecture · at-least-once delivery · idempotent consumer · CAP theorem · saga

Why this chapter matters

Chapter 5 described styles in which the whole system is deployed as one unit. This chapter takes the same questions across a network. Almost every system of any size is now distributed in some way: a mobile app talks to back-end services, which call payment providers, mapping services and message brokers, and the services themselves run on many machines. Splitting a system can bring independent scaling, fault isolation, independent releases and autonomous teams. It also brings a class of problems that cannot occur inside one process: messages lost or duplicated, services that are slow rather than dead, data that disagrees between copies, and failures in machines you did not know you depended on.

The single most important idea to take away is that **distribution is a trade, not an upgrade**. Each benefit is bought with latency, consistency, observability and operational effort, and a good architect can say which benefit is being bought, at what price, and why a cheaper option inside a monolith will not do. The fallacies of distributed computing and the arithmetic of synchronous call chains in the first section are the tools for pricing that trade before committing to it.

How the sections fit together

The chapter moves from why to how. Section 6.1 sets out the reasons to distribute, the eight fallacies, partial failure and the costs, and shows with a worked example how five services that each meet 99.9% availability give a request path of only about 99.5%. It ends by revisiting the architecture quantum, which explains why synchronous calls bind separately deployed services together.

The next three sections are the main distributed styles, each treated like the styles in Chapter 5: definition and origin, a diagram, how it works, an example, a characteristics table, when to use it and the usual anti-patterns. Section 6.2 covers service-based architecture, a pragmatic middle ground of a few coarse domain services often sharing a database, and the classic service-oriented architecture of enterprise service buses and canonical models, including why it fell out of favour. Section 6.3 covers microservices: services aligned to bounded contexts, independent deployability, database per service, supporting patterns such as API gateways and service meshes, how to decide granularity, and the distributed monolith and other anti-patterns. Section 6.4 covers event-driven architecture: events and commands, broker and mediator topologies, Kafka's partitions and consumer groups, delivery guarantees and idempotent consumers.

The second half turns to the mechanisms every distributed style depends on. Section 6.5 compares synchronous and asynchronous communication and the coupling each creates, and covers REST, gRPC, GraphQL and messaging, API design practices, versioning and contract testing. Section 6.6 is about what readers of replicated data can expect: consistency models from linearisability to eventual consistency, the CAP theorem and what it really says, PACELC and quorums. Section 6.7 explains why two-phase commit is avoided between services and how sagas, compensating transactions and the transactional outbox keep business processes correct without one. Section 6.8 closes with two styles for particular load profiles: serverless functions, which scale to zero and bill per invocation, and space-based architecture, which keeps data in memory to survive extreme spikes such as a ticket on-sale.

Running examples and how to read the chapter

Two examples recur. The food-delivery platform from earlier chapters (customers, restaurants, couriers, payments) shows ordinary distributed design: it appears as five domain services, as a set of microservices, as an `OrderPlaced` event flow and as an order saga. A ticketing system selling seats for a stadium tour shows what changes under extreme, spiky load, where tens of thousands of people want the same few seats at the same second.

The chapter is rated intermediate to advanced. It assumes Chapters 0 to 5, in particular availability in series and parallel (Chapter 2), coupling and Conway's law (Chapter 3), and the architecture quantum, modular monolith and bounded contexts (Chapter 5). Beginners can read each section's opening summary and the 'If you are new to this' boxes first. Experienced engineers will find the precision where it matters: what at-least-once delivery implies for every consumer, what 'exactly once' does and does not mean, and why eventual consistency must be designed for rather than tolerated.

What to carry forward

Three habits run through every section. First, count synchronous dependencies, not services: the number of things that must all be working at the same moment for a user's request to succeed decides availability and latency. Second, assume messages are lost, delayed, duplicated and reordered, and design receivers to be idempotent. Third, decide who owns each piece of data, because ownership decides where transactions are possible and where consistency must be managed across services. Chapter 7 develops that third habit into a full treatment of data architecture.

SECTION 6.1 OF 6.8 · INTERMEDIATE · 26 MIN READ · LAST REVIEWED NOVEMBER 2026

Why distribute, and the fallacies of distributed computing

BEFORE YOU READ THIS

This section builds on:

- 0.4 How computers talk
- 2.5 Availability and reliability
- 5.1 Styles and patterns

Terms it uses from earlier: Architecture quantum, Latency, Availability, Conway's law, Modular monolith

THE IDEA IN BRIEF

Imagine a small restaurant where one cook does everything in one kitchen. It is simple: if the cook is ill, the restaurant is closed, but nothing gets lost between people. Now imagine a chain with a central bakery, a sauce kitchen and many branches linked by delivery vans. It can serve far more customers, and one kitchen can close without closing the others. But vans get stuck in traffic, orders go missing, and when a delivery does not arrive nobody knows whether the bakery never received the order or the van broke down. Distributed software works like the chain.

WHAT YOU NEED TO KNOW

- A distributed system is a set of programs on different machines that cooperate by sending messages over a network to provide what users see as one service.
- Legitimate reasons to distribute are scale, fault isolation, independent deployability, team autonomy, geography and data residency, different technology needs and security isolation; each has a cheaper partial substitute inside a monolith that should be considered first.
- Partial failure is the defining problem: after a timeout the caller cannot tell whether the request was lost, the callee failed, or only the reply was lost, so retried operations must be idempotent.
- The eight fallacies of distributed computing (Deutsch and colleagues at Sun, with Gosling's eighth) are false assumptions about reliability, latency, bandwidth, security, topology, administration, transport cost and homogeneity.
- Distribution costs latency, observability, consistency and operational effort, and those costs grow roughly with the number of deployable units.

- In a synchronous chain availabilities multiply and latencies add: five services at 99.9% give about 99.5%, roughly 3.6 hours of downtime a month.
- Synchronous calls bind services together at run time (temporal coupling), so the number of synchronous hops in key journeys matters more than the number of services.

What makes a system distributed

A distributed system is a set of programs running on different computers that cooperate, by sending messages over a network, to provide what users see as one service. Your food-delivery app is one: the phone, the company's order and payment services, the restaurant's tablet, the courier's phone and a card processor all play a part in a single order. Chapter 5 described styles deployed as one unit (monoliths); this chapter is about styles whose parts are deployed separately and talk across the network.

Leslie Lamport, who did more than anyone to put the field on a mathematical footing, captured its essential difficulty in an internal email of May 1987: a distributed system is one in which “the failure of a computer you didn't even know existed” can make your own computer unusable. The joke works because it is true. Once a system depends on machines it does not control and cannot see, a whole class of failures appears that simply cannot happen inside one process.

That is why distribution is a decision with a price, not a sign of maturity. The rest of this section sets out the reasons that justify the price, the false assumptions that make people underestimate it, and a way of calculating part of it before you pay.

Why distribute at all

There are a handful of legitimate reasons to split a system across processes and machines. Each is a quality attribute from Chapter 2 that a single deployment unit struggles to deliver beyond a certain size. Each also has a cheaper partial substitute inside a monolith, and a good architect checks the substitute first.

Reasons to distribute, and what a well-built monolith can already do

Reason	What distribution offers	What a single deployment can already do
Scale	Parts with very different load can be scaled separately: the courier-location feed receives thousands of updates a second, the restaurant-onboarding screens a few an hour.	Run many identical replicas behind a load balancer, add read replicas and caches. This is enough for most systems, and far cheaper.
Fault isolation	A memory leak in recommendations cannot crash checkout, because they run in different processes on different machines.	Separate thread pools and bulkheads inside one process give partial isolation; a crash still takes everything down.
Independent deployability	One team can release its part twenty times a day without coordinating a release with anyone else.	A modular monolith with good automation can also release many times a day, but every release ships everything.
Team autonomy	Teams own a service end to end, choose their pace and are on call for it; Conway's law works for you rather than against you.	Module ownership works well up to a few teams; beyond that, a shared build and release become a queue.
Geography and data residency	Run parts near users for lower latency, or keep personal data in one jurisdiction because a law or a contract requires it.	Deploy a complete copy per region (a cell). That is itself a form of distribution, but a simple one.
Different technology needs	A recommendation model needs GPUs, search needs a search engine, a legacy integration needs a particular runtime.	Call out to the specialised component and keep everything else together.
Security isolation	Card data handled by one small, separately hardened and audited service reduces the scope of a PCI DSS assessment.	Not really: anything inside the process can reach anything else in memory.

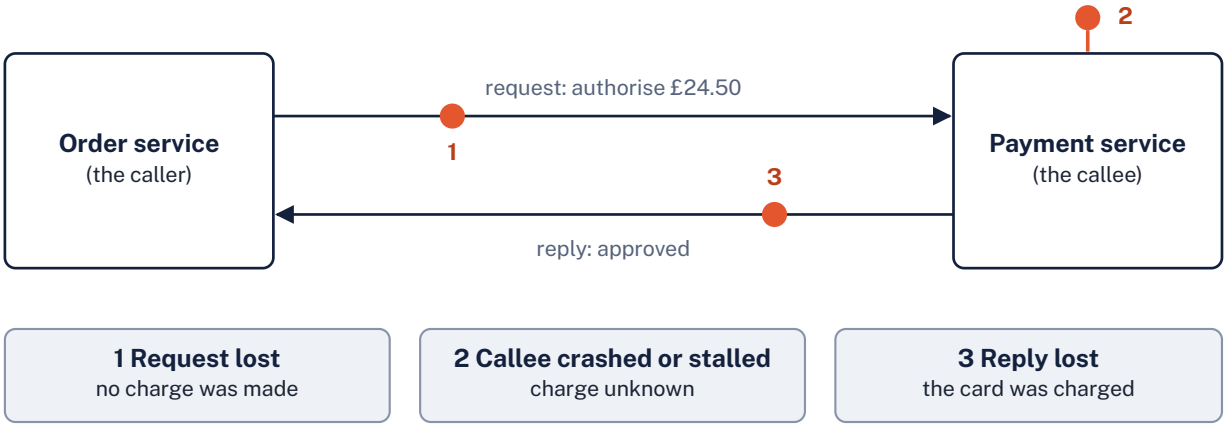
Two observations follow from the table. First, the strongest reasons are often organisational: independent deployability and team autonomy matter most when many teams work on one product, which is why Conway's law (Chapter 3) keeps reappearing in this chapter. Second, distribution is rarely all or nothing. Many successful systems are a modular monolith plus two or three separate services, each split off for a specific, named reason.

Partial failure: the defining problem

Inside one process a function call either returns or the whole process dies with it. Across a network there is a third outcome, and it changes everything: partial failure, in which some parts of the system work while others have failed, are slow, or cannot be reached. Worse, the caller often cannot tell which.

Consider the order service asking the payment service to authorise £24.50. If no reply arrives before the timeout, the order service does not know whether the request was lost on the way, whether the payment service crashed or stalled before or after charging the card, or whether the card was charged and only the reply was lost. All three look identical from the caller's side.

Three failures that look the same



To the caller all three look identical: no reply before the timeout.

After a timeout the caller cannot know whether the work was done; only idempotent retries or a later status check can resolve it.

The practical consequences shape every distributed design. A retry might charge the customer twice, so operations that may be retried must be made safe to repeat: this property is idempotency, and it is usually achieved by giving each request a unique key that the receiver remembers. A slow dependency is often worse than a dead one, because callers keep waiting and tie up threads and connections; when one service slows down, its callers run out of resources and slow down in turn, a cascading failure. And because a network can split so that two groups of machines cannot reach each other while both keep running, data held in both places can diverge: the subject of the CAP section later in this chapter.

There is no global clock either. Each machine's clock drifts and is corrected separately, so timestamps from two services cannot be compared with complete confidence, and 'which happened first?' can be a hard question. Lamport's 1978 paper on time and clocks in distributed

systems showed how to order events without relying on physical clocks; for architects the lesson is simply not to build correctness on comparing wall-clock times across machines.

IF YOU ARE NEW TO THIS

Step by step: when a reply never comes

The idea that needs the most care here is partial failure. Inside one program, a call to another part of the program either finishes or the whole program stops. Across a network there is a third possibility: you ask, and no answer comes back.

- The order service asks the payment service to take £24.50 from a customer's card, and waits for the answer.
- It cannot wait for ever, so it gives up after a set time limit, called a timeout: say two seconds. The two seconds pass with no reply.
- Three different things could have happened, and from the order service's side they look exactly the same: the request was lost on the way and nothing was charged; the payment service crashed or stalled, and nobody knows whether it charged the card; or the card was charged and only the reply was lost.
- If the order service simply tries again, it might charge the customer twice. If it gives up, the customer may have paid for an order that was never placed.
- The safe ways out are to make the request idempotent, meaning that sending it twice has the same effect as sending it once, or to ask the payment service afterwards what actually happened. Idempotency usually works by sending a unique request number with the charge: the payment service remembers the numbers it has handled and never charges the same one twice.
- **The lesson:** on a network, silence is not an answer. Every remote call needs a time limit and a plan for not knowing, which is why this chapter keeps returning to retries and duplicates.

The eight fallacies of distributed computing

The fallacies of distributed computing are eight assumptions that people new to distributed systems make, all of which are false and all of which cause trouble eventually. The list came from Sun Microsystems: L Peter Deutsch drew up seven in 1994, building on four that colleagues including Bill Joy had already noted, and James Gosling, the creator of Java, added the eighth around 1997. Three decades later every item still describes a real outage somewhere this week.

The eight fallacies, with a consequence and a design response for each

Fallacy	What goes wrong when you believe it	What to do instead
1. The network is reliable	A payment call hangs; the calling thread waits for ever; under load all threads are soon waiting and the order service stops answering anyone.	Set a timeout on every remote call, retry only idempotent operations, put queues where work can wait (resilience patterns).
2. Latency is zero	A basket page that made 40 in-process calls now makes 40 network calls at 2 ms each; 80 ms appears from nowhere, and far more at the tail.	Design coarse-grained interfaces, batch requests, call independent services in parallel, cache stable data.
3. Bandwidth is infinite	Every `MenuUpdated` event carries all the dish photographs; a busy evening saturates the link and delays location updates sharing it.	Send identifiers and the fields consumers need, paginate, compress, keep bulk data next to the code that processes it.
4. The network is secure	Services inside the data centre accept any request without authentication; one compromised container can call the refund API.	Encrypt and authenticate service-to-service traffic (for example mutual TLS) and authorise every call: zero trust.
5. Topology doesn't change	An IP address hard-coded in configuration stops working when the container is rescheduled to another machine.	Use service discovery and DNS names, and expect instances to come and go every minute.
6. There is one administrator	A cloud provider, a partner's firewall team or another department changes a rule, rotates a certificate or lowers a rate limit, and your calls start failing.	Know who controls each hop, monitor the dependencies you do not own, agree contracts and change notice.
7. Transport cost is zero	Serialisation burns CPU, and data crossing availability zones, regions or the public internet is billed per gigabyte; the network bill grows faster than traffic.	Measure the cost of data flows, keep chatty traffic local, choose compact formats where volume is high.
8. The network is homogeneous	A service assumes every client speaks the same protocol version, character encoding and clock; a mobile network, an old partner system or a proxy breaks the assumption.	Use standard, explicitly versioned formats and protocols, and test interoperability with the real clients.

WHERE THIS CAME FROM

The list keeps growing

In a 2021 interview Deutsch endorsed a proposed ninth fallacy, which he described as an extension of the fourth: that the party you are communicating with is trustworthy. Mark Richards and Neal Ford, in the second edition of *Fundamentals of Software Architecture* (2025), proposed three more drawn from microservice projects: versioning is easy, compensating updates always work, and observability is optional. All three return later in this chapter: versioning in communication and APIs, compensation in sagas, and observability below.

What distribution costs

Every reason to distribute is bought with the same currency. **Latency** is the first cost. A function call inside a process takes nanoseconds; a network round trip between machines in the same data centre typically takes from a few hundred microseconds to a millisecond or more, before any serialisation or work. Between continents physics sets the floor: light in optical fibre travels at roughly 200,000 km per second, so a London to New York round trip (about 5,600 km each way) cannot take less than about 56 ms. Chapter 2's tail-latency arithmetic applies to every hop.

Observability is the second. In a monolith a stack trace shows the whole story of a failed request. In a distributed system the story is scattered across the logs of many services on many machines. Observability, the ability to understand what the system is doing from its outputs, has to be designed in: correlation identifiers passed on every call, centralised logs, metrics per service and distributed tracing (Chapter 9).

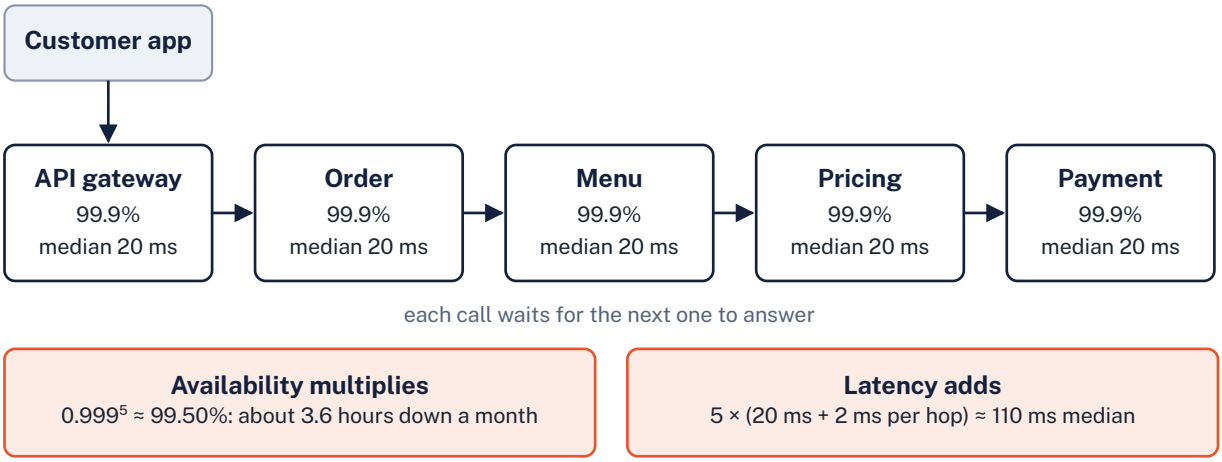
Consistency is the third. A single database transaction can update an order and its payment atomically. When the order and the payment live in different services with different databases, there is no shared transaction; the system must accept a period during which the two disagree and must make sure they converge. Later sections cover the tools: consistency models and sagas and the outbox pattern.

Operational load is the fourth, and the one most often underestimated. Each service needs a build pipeline, a deployment, configuration, secrets, monitoring, alerts, capacity planning, security patches and someone on call. Testing changes too: a change is only safe if every consumer of its interface still works, which calls for contract tests and backward-compatible changes. These costs are roughly proportional to the number of deployable units, which is why the number of services matters.

Doing the arithmetic: synchronous chains

The cheapest moment to discover that a distributed design is too fragile or too slow is on paper. The tool is the series rule from Chapter 2: if a request needs every component in a chain to work, the chain's availability is the product of the components' availabilities, and its latency is roughly the sum of theirs.

A five-service synchronous chain



Five services that each meet 99.9% produce a request path that meets only about 99.5%, and five times the latency of one.

WORKED EXAMPLE

Worked example: the availability and latency budget of a call chain

A food-delivery platform handles **Place order** with a synchronous chain: the API gateway calls the order service, which calls the menu service, which calls pricing, which calls payment. Each service on its own meets 99.9% availability (time-based, 30-day month of 43,200 minutes), answers in a median of 20 ms of its own work, and has a p99 of 250 ms. Each network hop adds about 2 ms. Assume, generously, that failures are independent and that any failure fails the whole request.

One service	Availability 0.999 = 99.9%; allowed downtime 0.001 × 43,200 = 43.2 minutes a month.
Two in series	0.999 × 0.999 = 0.998 (99.8%); about 86 minutes a month.
Five in series	$0.999^5 \approx 0.99501$ (99.50%); unavailable about 0.005 × 43,200 ≈ 216 minutes, or 3.6 hours, a month. Five individually respectable services have produced a two-nines-and-a-half system.
Ten in series	$0.999^{10} \approx 0.99004$ (99.0%); about 430 minutes, over 7 hours, a month.
Median latency, five in series	Latencies add: roughly 5 × (20 ms + 2 ms) ≈ 110 ms, against 20 ms for the same work done in one process. (Strictly, medians do not add exactly; the sum is a good first estimate.)
Tail latency, five in series	The chance that a request meets at least one hop in its slowest 1% is $1 - 0.99^5 \approx 4.9\%$, so about 1 request in 20 is slowed by at least one 250 ms hop. With ten hops it is $1 - 0.99^{10} \approx 9.6\%$.
What it takes to keep 99.9% end to end	Each of five services would need about 99.98% ($0.9998^5 \approx 0.9990$): roughly 8.6 minutes of downtime a month each, which in practice means automated recovery.
Design responses	Call menu and pricing in parallel (latency becomes the slower of the two, not the sum); cache menu data in the order service; make payment confirmation asynchronous so the order is accepted and paid shortly after; remove hops that only forward requests.

Chapter 2 introduced the rule that availabilities multiply in series; distribution is where that rule starts to bite, because every network call adds a component to the series.

The independence assumption flatters the result. Services that share a database, a cluster, a configuration system or a release train fail together, and a slow callee can exhaust the caller's threads and spread the outage upwards.

The cure is rarely to make each service heroically reliable. It is to shorten synchronous chains and let parts that need not answer immediately work from a queue.

The architecture quantum, revisited

Chapter 5 introduced the architecture quantum: an independently deployable part with high functional cohesion, including its data, to which one set of architecture characteristics applies (Architectural styles, patterns and tactics). In *Software Architecture: The Hard Parts* (2021) Ford, Richards, Sadalage and Dehghani sharpen the idea with two kinds of coupling. Static coupling is what must be present for a part to be deployed and run: its code, libraries, database and infrastructure. Dynamic coupling is how parts depend on each other while running: whether they call synchronously or asynchronously, and how they coordinate and stay consistent.

The worked example shows why the dynamic side matters. For the duration of a synchronous call, the caller cannot complete its work unless the callee is up and quick; the two are bound together operationally, and their availability and latency combine as calculated above. Five separately deployed services in a synchronous chain are five deployment units, but for the availability of **Place order** they behave like one fragile unit. This dependence on the other party being available at the same moment is temporal coupling. Asynchronous messaging removes much of it: if the order service puts a message on a durable queue, the payment service can be down for a minute and catch up afterwards, and the customer still gets an order number. The cost is that the customer does not get a definitive answer immediately, which is a product decision as much as a technical one.

This is the deepest lesson of the section. Drawing service boundaries is half the design; deciding which interactions must be synchronous, and keeping that set small, is the other half.

WATCH OUT

Common mistakes when deciding to distribute

- **Distributing to fix a code-quality problem.** A tangled monolith split into services becomes a tangled distributed system, with network failures added. Fix the boundaries first; Chapter 5's modular monolith is the usual route.
- **Treating remote calls like local ones.** Libraries that make a remote call look like a method call hide latency and failure; Martin Fowler's 'first law of distributed object design' was simply not to distribute your objects.
- **Counting services instead of synchronous hops.** Ten services that mostly exchange events can be more robust than three that call each other in a chain.
- **Assuming independence of failures.** Services sharing a database, a Kubernetes cluster or a configuration store fail together; the series formula is then optimistic and the parallel formula is fiction.
- **Ignoring the people cost.** Ten services need ten sets of pipelines, dashboards and on-call rotas. If one team of six owns all of them, the main benefit of distribution, team autonomy, is absent while every cost remains.

Checklist: before you distribute

Use these questions before splitting a system, or a part of one, across the network.

- [ ] **Name the driver**
  Which reason in the table above applies, to which part, with what evidence? 'Microservices are modern' is not a driver.
- [ ] **Try the cheaper fix first**
  Could replicas, caching, a read replica, a background worker or a clearer module boundary inside the monolith meet the need?
- [ ] **Draw the call graph**
  Count the synchronous hops in each key user journey and compute the availability and latency budget, as in the worked example.
- [ ] **Decide who owns which data**
  Every piece of data needs one owning service; anything that must change atomically with it probably belongs in the same service.
- [ ] **Design for partial failure**
  Timeouts, retries with idempotency, fallbacks and what the user sees when a dependency is down, decided before the first outage.
- [ ] **Check the prerequisites**
  Automated provisioning and deployment, monitoring, centralised logs and distributed tracing must exist, or be planned and funded.
- [ ] **Match teams to boundaries**
  Each service should have one owning team that can change, deploy and support it without asking others.

CHECK YOUR UNDERSTANDING

▼ **A service calls a stock service with no timeout. The stock service becomes very slow rather than failing. Why can this be worse than if it had crashed?**

A crashed service fails fast and callers can react; a slow one keeps callers waiting, so their threads and connections fill up until they stop answering too, spreading the outage as a cascading failure.

▼ **Three services each have 99.95% availability and are called in series on every request. What is the approximate availability of the request path?**

$0.9995^3 \approx 0.9985$, about 99.85%, assuming independent failures.

▼ **Why does the fallacy 'transport cost is zero' matter even when latency and bandwidth are fine?**

Moving data costs CPU for serialisation and money for traffic between zones, regions or providers, so a chatty or bulky design can be expensive even when it is fast enough.

▼ **Why can five separately deployed services still behave like one unit for availability purposes?**

If a request calls them synchronously in a chain, all five must be up at the same moment; their availability combines, so they are temporally coupled despite separate deployment.

▼ **Try it: Place order calls four services synchronously, each 99.9% available. Work out availability and monthly downtime (43,200 minutes). If notification and dispatch become asynchronous, leaving two synchronous, what changes? What must you add if a retry follows a timeout?**

Four: 0.999^4 = 0.99601, about 99.6%, or 173 minutes a month. Two: 0.999^2 = 0.99800, about 86 minutes, roughly half, assuming independent failures. After a timeout the caller cannot tell whether the work happened, so retried calls must be idempotent, for example with an idempotency key, or a customer could be charged twice. The asynchronous steps need queues that are monitored, but they no longer stop the order.

SOURCES

- Leslie Lamport: the 1987 email defining a distributed system (lamport.org): https://lamport.org/pubs/distributed-system.txt
- The Eight Fallacies of Distributed Computing (James Gosling's page): https://nighthacks.com/jag/res/Fallacies.html
- Software Engineering Radio episode 470: L Peter Deutsch on the fallacies of distributed computing (2021): https://se-radio.net/2021/07/episode-470-l-peter-deutsch-on-the-fallacies-of-distributed-computing/
- Martin Fowler: Microservices and the First Law of Distributed Objects: https://martinfowler.com/articles/distributed-objects-microservices.html
- Google SRE book: Addressing Cascading Failures: https://sre.google/sre-book/addressing-cascading-failures/

SECTION 6.2 OF 6.8 · INTERMEDIATE · 22 MIN READ · LAST REVIEWED NOVEMBER 2026

Service-based architecture and service-oriented architecture

BEFORE YOU READ THIS

This section builds on:

- 0.5 Storing data
- 5.3 Modular monolith
- 6.1 Why distribute

Terms it uses from earlier: ACID, Schema, Architecture quantum, Domain partitioning, Bounded context

THE IDEA IN BRIEF

Think of a hospital that outgrows one building. It could knock everything into hundreds of tiny clinics, each with its own reception and records office, or it could move into five departments in separate wings: maternity, surgery, outpatients, pharmacy, accident and emergency. Each wing has its own staff and can be refurbished without closing the others, but they still share one central records office. That is service-based architecture: a few big, separate parts, each looking after a whole area of the business, often still sharing one database.

WHAT YOU NEED TO KNOW

- Service-based architecture (named by Richards and Ford) splits a system into a few separately deployed, coarse-grained domain services, typically 4 to 12, usually sharing one database and often behind an API layer.
- Because most workflows stay inside one domain service, they can use local ACID transactions, and services rarely call each other, which keeps synchronous chains short.
- A shared database couples schema changes and binds the services into one quantum; per-service schemas, small per-domain entity libraries and separate databases where justified reduce that coupling.
- Classic orchestration-driven SOA arranged services in technical layers around an enterprise service bus, with a canonical data model and WS-* standards, aiming at enterprise-wide reuse.
- SOA disappointed because reuse created coupling, the bus became a bottleneck owned by one team, canonical models fitted nobody and the standards were heavy.
- Microservices keep explicit service contracts but put the logic in the services (smart endpoints, dumb pipes), organise by business capability and give each service its own data.

- Service-based architecture is a pragmatic middle ground for a few domains and teams, and a low-risk step out of a monolith.

Between the monolith and microservices

Much of the industry conversation presents a choice between one monolith and many microservices. In practice a large share of real systems sit in between: a handful of separately deployed services, each covering a whole business area, often still sharing one database. Mark Richards and Neal Ford named this style service-based architecture in *Fundamentals of Software Architecture* and argue that it is one of the most pragmatic distributed styles available. This section describes it, contrasts it with the older service-oriented architecture (SOA) that it is often confused with, and shows where microservices differ from both.

Service-based architecture: the topology

A service-based system has three parts. One or more user interfaces call a set of separately deployed domain services, each covering a large, coherent business area such as ordering or payments. Richards and Ford observe that such systems usually have between 4 and 12 domain services, about 7 on average. The services usually share a database, and an optional API layer (a reverse proxy or gateway) may sit between the user interfaces and the services to handle routing, authentication and rate limiting.

Each domain service is coarse-grained: it contains everything needed for its domain, from the API to the business logic to the data access, and it is internally structured like a small monolith, often in layers or in modules (Chapter 5). A service is typically deployed as a few instances behind a load balancer.

Two consequences follow. First, most business workflows run entirely inside one service, so they can use an ordinary local database transaction with full ACID guarantees (Chapter 0). Second, domain services rarely need to call each other: a request from the user interface goes to the one service that owns the domain. This keeps synchronous chains short, which matters for the reasons calculated in Section 6.1.

IF YOU ARE NEW TO THIS

Step by step: how a shared database ties services together

The step most people trip over here is that services deployed separately can still be tied together by the database they share. A small example shows how.

- Suppose the food-delivery platform's services read each other's tables freely. Ordering, Restaurant and Delivery all read the restaurants table in the one shared database.
- The Restaurant team wants to split the table's address column into separate street, town and postcode columns, to make searching by postcode easier. In their own code it is a one-day change.
- But Ordering and Delivery also read the address column. The moment it disappears, they break, even though nobody touched their code.
- So all three services must be changed, tested and released together. Services tied by shared tables cannot be deployed independently, which is why the section says they form one architecture quantum.
- The main remedy in this section is to give each service its own set of tables, its own schema, that no other service may touch. Ordering and Delivery then read addresses through a read-only view that Restaurant publishes, and Restaurant can reorganise its own tables freely as long as that view keeps the same shape.
- **The lesson:** sharing a database is what makes this style cheap and simple, but every table several services read is a promise that several teams must keep together. Give each table one owning service, and share only a few published views.

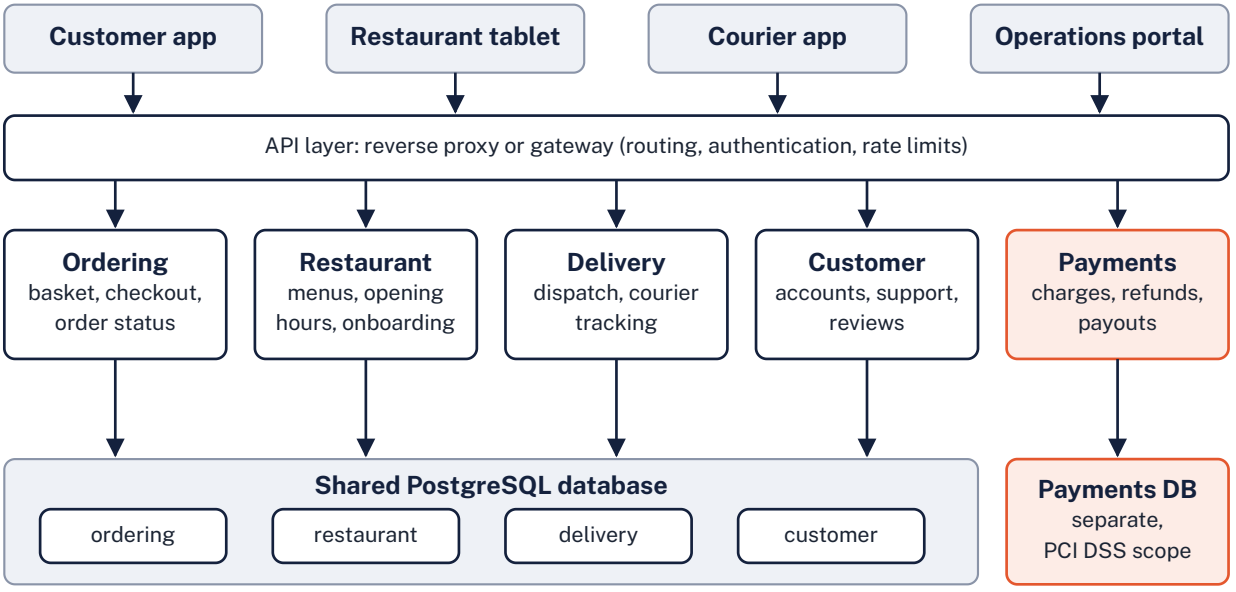
The shared database and how to tame it

Sharing a database is what makes service-based architecture cheap and simple: no data synchronisation between services, no eventual consistency, reporting by ordinary SQL. It is also its main weakness. If every service reads and writes the same tables, a change to one table's columns may break several services, all of which must then be tested and redeployed together. Chapter 5's definition applies: services sharing a database are bound into one architecture quantum, so they share one level of availability and one schema release cadence.

Practitioners use three techniques to reduce the coupling. The first is logical partitioning: give each service its own schema (or set of tables) and forbid it to touch anyone else's, enforced by database permissions as well as by convention. The second is to keep the shared database's entity classes in small, per-domain libraries rather than one large shared library, so that a change to restaurant tables only affects the services that use them. The third is to give a

service its own database when it has a strong reason, such as security scope, a very different load pattern or a different kind of data store. Teams that apply these consistently find that a later move to separate databases, if it ever becomes necessary, is much easier.

The food-delivery platform as five domain services



Each service owns its own schema; only Payments has a separate database

Five coarse-grained services, each owned by one team, behind an API layer; four share a database with a schema each, and Payments keeps its own.

WORKED EXAMPLE

Worked example: the food-delivery platform as five domain services

A food-delivery company has grown from one team to five, about 35 engineers. Its Django monolith is well modularised, but releases have become a queue and a fault in courier tracking took down checkout twice last quarter. The architect proposes a service-based architecture with five domain services, one per existing module and team.

Ordering	Basket, checkout, order lifecycle and status. Owns the `ordering` schema. Busy at meal times; runs 6 to 20 instances.
Restaurant	Menus, prices, opening hours, restaurant onboarding. Read-heavy; menus are cached at the API layer. Runs 3 instances.
Delivery	Courier dispatch, route estimates, live tracking. Receives location updates every few seconds from every active courier, so it is scaled separately from everything else.
Payments	Card authorisation through the payment provider, refunds, restaurant and courier payouts. Has its own database so that the scope of the PCI DSS assessment is one small service rather than the whole platform.
Customer	Accounts, addresses, support tickets, reviews. Low load, but holds most personal data, so access and retention rules are concentrated here.
How a workflow runs	**Place order** runs inside Ordering in one local transaction (create order, reserve the restaurant's slot), then makes one call to Payments. Restaurant and Delivery learn about the order from a message, not a chain of calls.
What is shared	One PostgreSQL server with a schema per service. Each service may only read and write its own schema; a few read-only views (restaurant names, for example) are published for the others.
What it buys	Five deployment pipelines instead of one; a fault in tracking no longer stops checkout; payments can be audited alone; each team releases when ready.
What it costs	Five deployables and dashboards instead of one; network calls between Ordering and Payments that need timeouts and idempotency; schema changes still coordinated through the shared database server.

The step is modest on purpose. It moves from one quantum to roughly two (Payments with its own data, and the four services sharing the database), and it gives each team its own release, which was the main pain.

If Delivery's location stream later outgrows the shared database, that one service can be given its own store and split further; the rest can stay as they are.

The architect writes the boundary rules down as architecture tests: no service may query another's schema, and services may call each other only through published APIs.

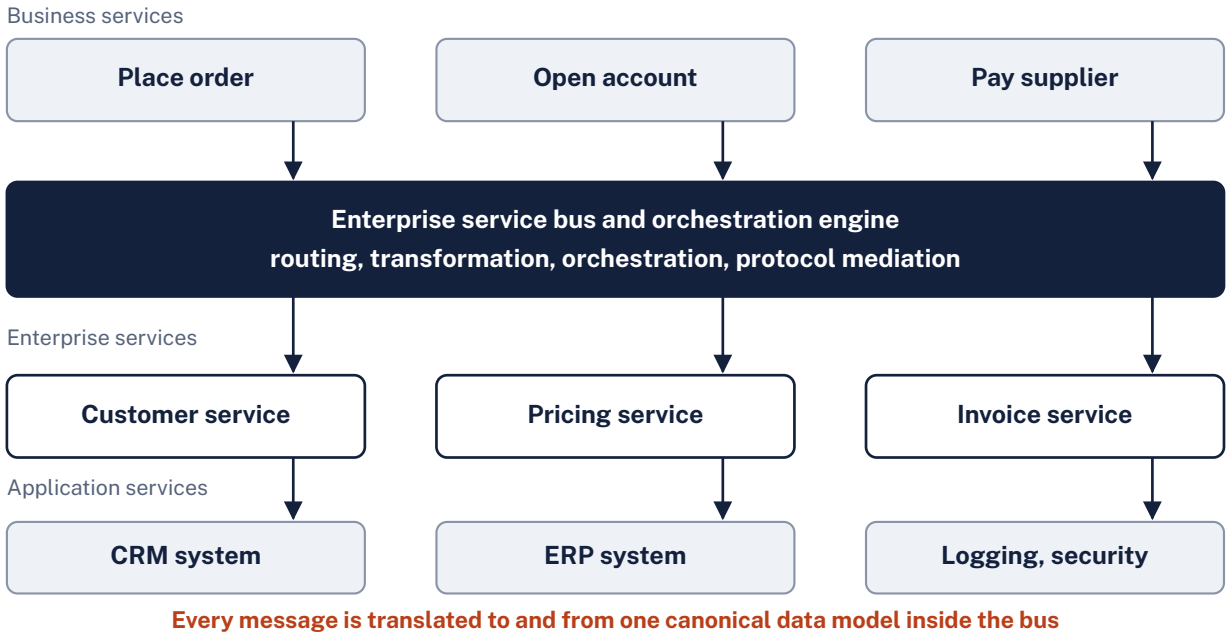
Classic service-oriented architecture

Service-oriented architecture is older and broader. The term entered the industry through a 1996 Gartner report by Roy Schulte and Yefim Natis, and from the early 2000s it became the dominant model for enterprise integration. Its central idea was sound: expose business capabilities as services with explicit, technology-neutral contracts, so that applications across a large organisation can be assembled from them instead of being rebuilt or integrated point to point.

What most enterprises actually built, however, was what Richards and Ford call orchestration-driven SOA. Services were arranged in technical layers: business services describing processes (place order, open account), shared enterprise services providing reusable capabilities (customer, pricing, invoice), and application and infrastructure services wrapping existing systems and technical concerns. At the centre sat an enterprise service bus (ESB): middleware that routed messages, transformed them between formats, mediated between protocols and, through an orchestration engine, ran the business processes themselves.

Two further ideas completed the picture. A canonical data model defined one enterprise-wide representation of each business entity (one Customer, one Order), and every message was translated to and from it in the bus. And a family of web-service standards, usually called WS-*, defined how services described and called each other: SOAP for messages, WSDL for interface descriptions, UDDI for registries, WS-Security, WS-ReliableMessaging and WS-AtomicTransaction for qualities, and WS-BPEL (version 2.0 approved by OASIS in 2007) for executable process definitions.

Orchestration-driven SOA



In classic SOA the bus held routing, transformation and process logic, and all services shared one canonical model of the business.

Why classic SOA fell out of favour

By the late 2000s many SOA programmes had disappointed. Anne Thomas Manes of the Burton Group captured the mood in a widely read January 2009 post titled *SOA is Dead; Long Live Services*: the label had become toxic, but the idea of services survived. The reasons are worth knowing, because each is a trap that modern architectures can fall into again.

Reuse created coupling. A shared enterprise Customer service used by forty applications could not change without forty consumers agreeing; reuse, the main goal, turned every change into a negotiation. **The bus became the bottleneck.** Putting routing, transformation and process logic into a central ESB, usually owned by one integration team, meant that almost every business change needed that team, and the bus became a single point of failure and of delay. **The canonical model was too big.** One definition of Customer that satisfies sales, billing, support and logistics is either enormous or wrong for most of them; Chapter 5's bounded contexts explain why a term legitimately means different things in different parts of a business. **The standards were heavy.** WS-* specifications were complex, tooling was expensive, and distributed transactions across services proved fragile. **Technical partitioning** meant a simple feature crossed several layers of services owned by different groups.

The lasting lessons are the ones microservices were built on: explicit service contracts, services owned by the people who build them, logic in the services rather than in the middleware, and data owned by one service rather than shared across the enterprise.

How microservices differ

Microservices, covered in depth in the next section, keep SOA's idea of services with explicit contracts and reverse most of its implementation choices. James Lewis and Martin Fowler summarised one reversal as *smart endpoints and dumb pipes*: the network only transports messages, and all logic lives in the services. Services are organised around business capabilities rather than technical layers, reuse is deliberately secondary to independence, each service owns its data, and governance is decentralised.

Compared with service-based architecture, microservices are finer-grained, far more numerous, and insist on separate data. That buys more independence, more targeted scaling and more fault isolation, and it costs much more in operations, consistency and coordination. Service-based architecture can be seen as the point on the spectrum where most of the deployment and team benefits arrive for a fraction of the cost.

Service-based architecture, classic SOA and microservices compared

Aspect	Service-based	Classic (orchestration-driven) SOA	Microservices
Main goal	Pragmatic modularity and independent deployment of a few domain areas	Enterprise-wide reuse of shared services	Independent change and deployment by many autonomous teams
Partitioning	Domain: one service per large business area	Technical: business, enterprise, application and infrastructure layers	Domain: one service per bounded context or smaller
Typical number	About 4 to 12 per system	Dozens to hundreds across an enterprise	Tens to hundreds per system
Granularity	Coarse: a whole domain, for example all of ordering	Mixed; shared enterprise services are often fine-grained	Fine to medium: one cohesive capability
Data	Often one shared database, ideally partitioned by schema	Shared systems of record behind enterprise services	Each service owns its data; no shared tables
Communication	Mostly from the user interface to one service; few calls between services	Through the enterprise service bus, usually SOAP or XML messaging	Lightweight HTTP, gRPC or messaging; smart endpoints, dumb pipes
Where the logic lives	Inside the domain services	Much of it in the bus: routing, transformation, orchestration	Inside the services; the network only transports
Transactions	Local ACID transactions within a domain service	Distributed transactions attempted (WS-AtomicTransaction, XA)	No distributed transactions; sagas and eventual consistency
Governance	Light; one architecture team is enough	Central and heavy: canonical model, service registry, standards board	Decentralised; shared platform and paved roads
Operational cost	Moderate	High, plus expensive middleware licences	High: many deployables, pipelines and on-call rotas

Characteristics of service-based architecture

Characteristic	Rating	Reason
Deployability	Medium to high	Each domain service is deployed on its own, so a change to menus does not redeploy payments; a shared schema change still needs coordination.
Fault tolerance	Medium to high	A crash in the delivery service leaves ordering and payments running; a shared database is still a common point of failure.
Scalability	Medium	Services scale separately, but each is coarse, and the shared database limits how far writes can scale.
Elasticity	Low to medium	Large services start slowly and scale in big steps, so sudden spikes are absorbed less gracefully than with small services.
Data consistency	High	Most business workflows stay inside one service, where an ordinary database transaction keeps data consistent.
Simplicity	Medium	Far fewer moving parts than microservices; still a distributed system with network calls, deployment pipelines and monitoring per service.
Overall cost	Medium (moderate)	A handful of services and one database are cheap to run compared with dozens of services and databases.
Testability	Medium to high	Each service has a smaller scope than the monolith and can be tested on its own against its schema.
Evolvability	Medium	Domain partitioning keeps typical changes inside one service, but the shared database couples schema evolution.

When service-based architecture is the pragmatic middle ground

Choose it when the system has a few clear business domains, when two to perhaps eight teams need to release independently, when most workflows naturally stay inside one domain, and when strong consistency within a domain is valuable. It is also an excellent migration step out of a monolith: carving a well-modularised monolith into a handful of domain services is far less risky than leaping to fifty microservices, and the result is often good enough to stop there.

Avoid it, or plan to move beyond it, when parts of a domain need very different scalability or availability (they belong in separate quanta), when dozens of teams need autonomy, when the shared database has become the bottleneck for writes, or when the domain services keep calling each other to complete ordinary requests, which is a sign that the boundaries are wrong.

WATCH OUT

Anti-patterns

- **The shared-database free-for-all.** Services reading each other's tables directly. Within months the schema cannot be changed without a coordinated release of everything.
- **Chatty domain services.** Ordering calls Restaurant, which calls Customer, which calls Ordering. If ordinary workflows need several services, merge them or move the workflow.
- **The one giant entity library.** A single shared library containing every table's classes, so any schema change forces every service to rebuild and redeploy.
- **SOA by another name.** A central integration layer that slowly acquires routing rules, transformations and business logic: an ESB reinvented, with the same bottleneck.

CHECK YOUR UNDERSTANDING

▼ **Why can most workflows in a service-based architecture use an ordinary database transaction?**

Because each domain service covers a whole business area and the workflow usually stays inside it, touching only data in that service's own schema in one database.

▼ **What made the enterprise service bus a bottleneck in many SOA programmes?**

Routing, transformation and process logic lived in the bus, typically owned by one central team, so most business changes needed that team and the bus was a shared point of failure.

▼ **Name two techniques that reduce the coupling caused by a shared database.**

Giving each service its own schema and forbidding access to others (enforced by permissions), and splitting entity classes into small per-domain libraries; also giving a service its own database where justified.

▼ **What does 'smart endpoints and dumb pipes' reverse compared with classic SOA?**

It moves logic out of the middleware (the ESB) and into the services, leaving the network to transport messages only.

▼ Try it: A ticketing company with 12 engineers and a tidy monolith suffers release queues, and a reporting fault took checkout down twice. Would you recommend service-based architecture or fine-grained microservices? Give your reasoning, and one risk of your choice.

Service-based: about four to six coarse domain services such as Booking, Payments, Venues and Reporting. Most workflows stay inside one service, so local transactions survive and synchronous chains stay short. It isolates reporting faults and eases release queues without the operating cost of dozens of services, which 12 engineers could not carry. The risk is the shared database, which couples schema changes and keeps them one quantum. Use per-service schemas, and separate Reporting's data.

SOURCES

- James Lewis and Martin Fowler: Microservices (2014), on smart endpoints and dumb pipes: https://martinfowler.com/articles/microservices.html
- OASIS: Reference Model for Service Oriented Architecture 1.0: https://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf
- W3C: Web Services Architecture (Working Group Note): https://www.w3.org/TR/ws-arch/
- OASIS: Web Services Business Process Execution Language Version 2.0: https://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
- Enterprise Integration Patterns: messaging patterns catalogue: https://www.enterpriseintegrationpatterns.com/patterns/messaging/

SECTION 6.3 OF 6.8 · ADVANCED · 26 MIN READ · LAST REVIEWED NOVEMBER 2026

Microservices: principles, benefits and costs

BEFORE YOU READ THIS

This section builds on:

- 5.7 Domain-driven design
- 6.1 Why distribute
- 6.2 Service-based and SOA

Terms it uses from earlier: Bounded context, Architecture quantum, Eventual consistency, Enterprise service bus (ESB), Conway's law

THE IDEA IN BRIEF

Imagine a large food hall instead of one big restaurant. Each stall has its own small team, its own kitchen, its own fridge and its own menu, and can change its recipes or close for refurbishment without asking the other stalls. Customers order through a shared front desk that sends each order to the right stall. The food hall can grow by adding stalls, and one stall's burnt kitchen does not close the others. But somebody has to run the shared front desk, deliveries, cleaning and safety checks for every stall, and a meal from three stalls needs careful coordination. Microservices are the food hall.

WHAT YOU NEED TO KNOW

- Microservices (Lewis and Fowler, 2014) build one application as small services, each in its own process, built around a business capability, independently deployable and communicating through lightweight mechanisms.
- Service boundaries work best when aligned with bounded contexts and with teams; boundaries drawn around technical layers or database tables produce chatty, tightly coupled services.
- Independent deployability is the defining property; it depends on backward-compatible contracts, contract tests and versioning, not only on separate containers.
- Database per service makes each service its own quantum but replaces joins and transactions with API calls, events, sagas and eventual consistency.
- Supporting patterns include API gateways, backends for frontends, service discovery and service meshes; each solves a real problem and adds something to run.

- Granularity is decided by weighing disintegrators (scope, volatility, scalability, fault tolerance, security, extensibility) against integrators (transactions, workflow, shared code, data relationships).
- Typical failures are the distributed monolith, entity services, chatty services and shared libraries that couple deployments; the style needs automation, monitoring and teams that run what they build.

Definition and origin

Microservices is an architectural style in which one application is built as a suite of small services, each running in its own process, each built around a business capability, each independently deployable by automated machinery, communicating through lightweight mechanisms such as HTTP APIs or messaging, with a minimum of central management. That summary follows the article that defined the term for most of the industry: *Microservices* by James Lewis and Martin Fowler, published on martinfowler.com in March 2014. The article notes that the name was discussed at a workshop of software architects near Venice in May 2011 and adopted by the group in May 2012; companies such as Amazon and Netflix were already working this way.

Lewis and Fowler did not give a formal definition. They described nine common characteristics, and practitioners still use them as a checklist: componentisation via services; organisation around business capabilities; products rather than projects; smart endpoints and dumb pipes; decentralised governance; decentralised data management; infrastructure automation; design for failure; and evolutionary design. The sections below take the most architecturally significant of these in turn.

IF YOU ARE NEW TO THIS

If you are new to this

A *service* here is simply a program that runs on its own, usually on several servers at once, and offers an API to other programs (Chapter 0). In a monolith, the code for menus, orders and payments is compiled into one program and deployed together. In microservices, each of those becomes a separate program, owned by its own small team, with its own database, released whenever that team is ready. The rest of this section is about where to draw the lines between those programs, and what it costs to run so many of them.

Services aligned to bounded contexts

The hardest question in microservices is where to draw the lines. The most reliable answer comes from strategic domain-driven design (Chapter 5): a bounded context is a part of the business within which a model and its language are consistent. In food delivery, 'order' means a basket being paid for in Ordering, a ticket to cook in the restaurant's kitchen context and a pickup and drop-off in Dispatch. Each context is a natural candidate for one service, or for a small number of services owned by one team.

Aligning services with bounded contexts gives them high cohesion (what changes together lives together) and low coupling (contexts talk through a few explicit contracts, translated at the edges). It also aligns them with teams, because a bounded context is usually the size of thing one team can understand and own. Services drawn around technical layers or around database tables lack both properties, and the anti-patterns later in this section are mostly the result.

Independent deployability: the defining property

If one property defines microservices, it is independent deployability: a team can change its service and release it to production without changing, testing or releasing any other service at the same time. Sam Newman, in *Building Microservices* (second edition, 2021), treats it as the property to protect above all others, because almost every other benefit depends on it. Faster delivery, smaller blast radius, targeted scaling and team autonomy all disappear if releases must be coordinated.

Independent deployability is a property of the contracts, not of the containers. Two services that are deployed separately but must always be released together, because a change in one breaks the other, are not independently deployable. Protecting it requires backward-compatible interface changes (add fields, never remove or rename them without a deprecation period), consumer-driven contract tests and versioned APIs, all covered in communication and APIs.

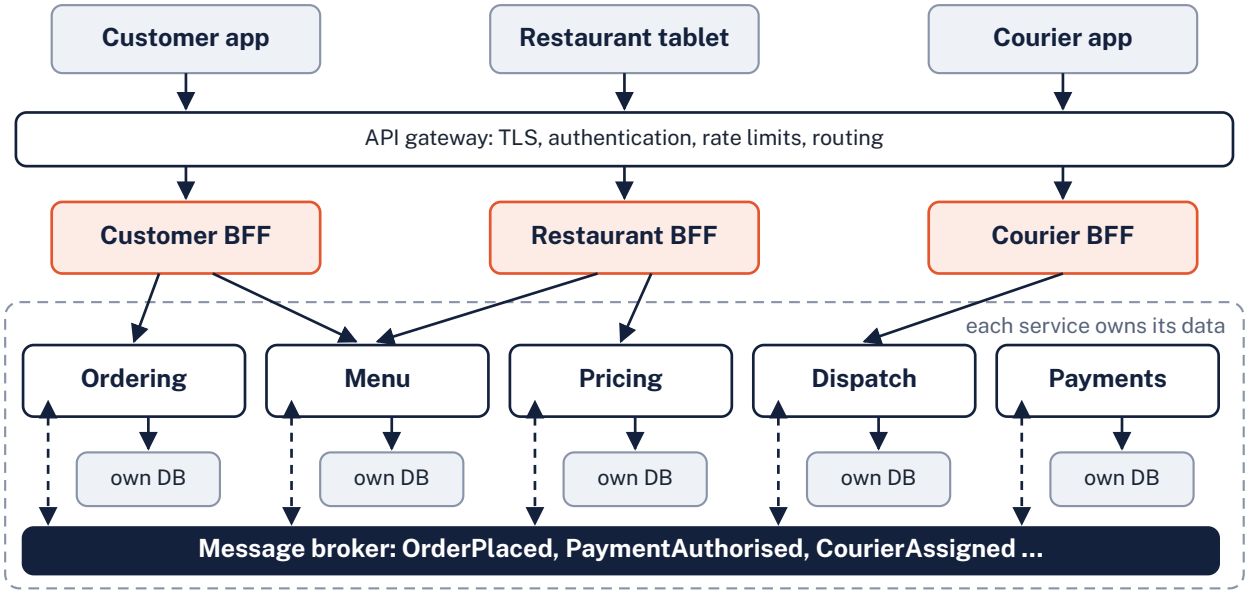
Database per service, and smart endpoints with dumb pipes

Lewis and Fowler's decentralised data management becomes, in practice, the rule of database per service: each service owns its data, and no other service reads or writes it directly. Other services get the data through the owner's API or through events the owner publishes. The rule is what makes a service a separate quantum: its schema can change, and it can be scaled, backed up and restored, without asking anyone. Each service may also pick the store that suits it (a relational database for orders, a key-value store for sessions, a search engine for menus), which is called polyglot persistence.

The price is high and must be paid knowingly. Queries that used to be a SQL join across tables now cross services. Changes that used to be one transaction now need a saga and become eventually consistent. Reporting needs a separate pipeline. Chapter 7 treats data ownership in depth.

The other characteristic with the largest architectural consequences is *smart endpoints and dumb pipes*. Business logic belongs in the services; the network and the message broker only carry messages. This is a direct reaction to the enterprise service bus of classic SOA, where routing rules, transformations and processes accumulated in central middleware.

A microservices topology for food delivery



Solid arrows: synchronous calls. Dashed: events published to and consumed from the broker.

Clients reach services through a gateway and one backend for frontend each; every service owns its data, and events flow through a broker.

Supporting patterns

A set of supporting patterns has grown up around the style. Each solves a real problem and each adds a component that must be run, so adopt them when the problem appears rather than all at once.

Common supporting patterns

Pattern	Problem it solves	What it is	Watch out for
API gateway	Clients should not know the addresses of dozens of services or repeat authentication against each.	A single entry point that terminates TLS, authenticates, rate-limits and routes requests to services; it may combine several calls into one response.	Business logic creeping into the gateway, recreating an ESB; a single point of failure that needs high availability.
Backend for frontend (BFF)	A mobile app, a web app and a partner API need differently shaped data; one general API serves none of them well.	A separate, thin back end per kind of client, owned by that client's team, which calls services and shapes responses. The name is Phil Calçado's, from SoundCloud; Sam Newman popularised the pattern in 2015.	Duplicated logic across BFFs; a BFF that becomes a mini-monolith.
Service discovery	Service instances start, stop and move all the time; addresses cannot be configured by hand.	A registry of the current healthy instances, queried by clients or by a load balancer. Kubernetes provides it through DNS names for Services.	Stale registry entries; clients caching addresses too long.
Service mesh	Every service needs mutual TLS, retries, timeouts, traffic shifting and telemetry; implementing them in every language is wasteful.	An infrastructure layer of proxies (traditionally a sidecar next to each service instance) controlled centrally, such as Istio or Linkerd.	Considerable operational complexity; retries configured in both the mesh and the code multiply load.

Fowler's prerequisites, and the organisational costs

In August 2014 Martin Fowler listed three things an organisation should have before building microservices at all. **Rapid provisioning**: the ability to create a new server or environment in hours, which means automation. **Basic monitoring**: both technical measures (errors, latency, availability) and business measures (orders per minute) for every service, so that problems are spotted quickly. **Rapid application deployment**: an automated pipeline that takes a service to test and production quickly, with easy rollback. Without these, many small services multiply manual work. His companion idea, the *microservice premium* (2015), is that the style carries a fixed cost in complexity and productivity that only pays off for systems complex enough to need it.

The organisational costs are as real as the technical ones. The style assumes long-lived teams that own their services in production, the principle Amazon's CTO Werner Vogels described in 2006 as “You build it, you run it”. That means every team needs operational skills and an on-call rota. It needs a platform: deployment pipelines, observability, secrets management and service templates, usually provided by a platform team (Chapter 10). It needs agreement on cross-cutting standards (logging format, correlation identifiers, API conventions) without central approval of every change. And it needs Conway's law to be used deliberately: team boundaries and service boundaries should match, otherwise every change needs several teams to coordinate.

The client side: rendering, micro-frontends, mobile, offline and the BFF

Customers meet a system through its clients, and clients have their own architecture. The first decision is where a page is rendered. In a **single-page application** (SPA) the browser builds each screen with JavaScript, which suits long, interactive sessions such as a restaurant's order screen, but the code tends to grow with the application and can slow the response to taps. With **server-side rendering** (SSR) the server sends finished HTML, so content appears quickly and less code runs on the device, at the cost of server work for each request. **Static rendering** produces HTML at build time for a CDN, which suits help pages but not personalised ones. Many frameworks render on the server and then *hydrate*, attaching behaviour in the browser; Google's web.dev warns that such a page can look ready before it responds. The choice can differ page by page.

Micro-frontends carry independent deployment into the interface: "independently deliverable frontend applications are composed into a greater whole" (Cam Jackson, 2019). The menu and checkout teams could each release their own part of the website, composed at run time by a tool such as webpack's Module Federation. The costs are duplicated libraries in every download, or shared versions that bring coupling back, plus more pipelines and more effort to keep the look consistent. The style answers the problem of many teams on one front end; a small team gains little.

Mobile changes the release rules. An update reaches users gradually, through Apple's seven-day phased release or Google Play's staged rollout, and halting a Play rollout leaves the update with those who already have it. Old versions stay in use, so APIs must remain compatible for a long time (see Section 6.5).

An **offline-first** client treats a local store as its source of truth: screens read local data and a sync process reconciles it with the server, which keeps final authority. Android's guidance names three write strategies: online-only, suited to payments; queued; and lazy, which writes locally first and must resolve conflicts. A courier in an underground car park can mark an order

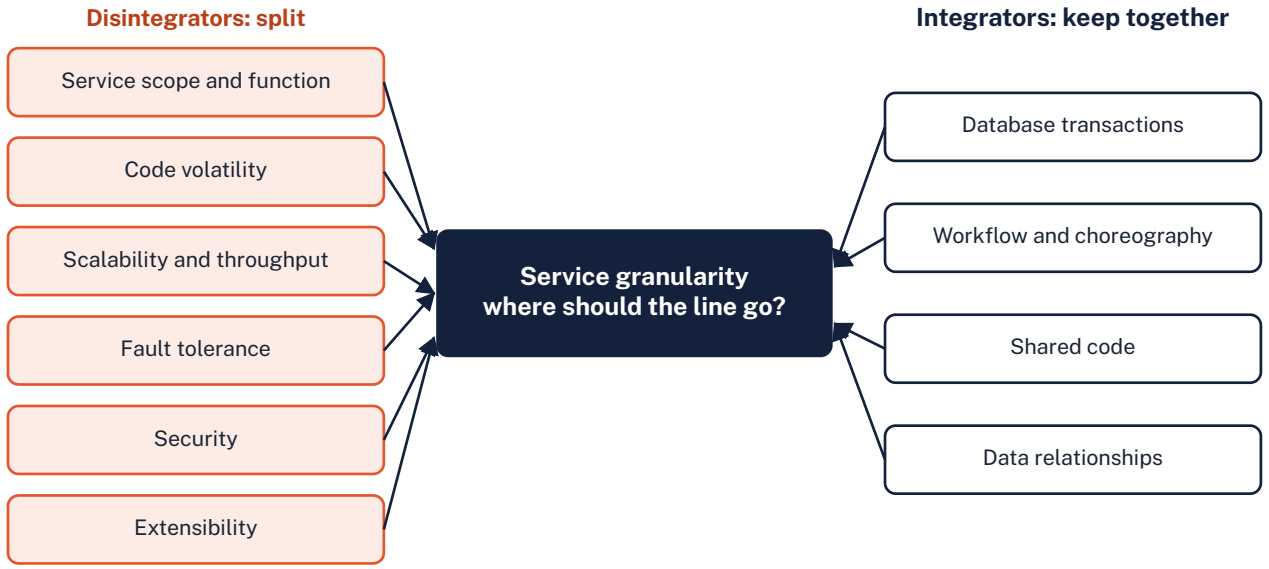
delivered and let the app send it later, with an idempotency key. In browsers, service workers can serve cached pages, but the Background Sync API does not work in all major browsers, so check the current support.

The **backend for frontend** ties this together. Sam Newman's guideline is one experience, one BFF, kept by the team that builds that interface, so the courier BFF can return small payloads for poor networks while the web BFF shapes data page by page. It can also keep tokens out of the browser (see Section 8.3). Business rules stay in the services: Newman suggests moving logic shared by BFFs into a domain service. His warning about the alternative, one general API for every client, is that it becomes a bottleneck.

Granularity: how big should a service be?

There is no correct size in lines of code or in people. Lewis and Fowler found teams of about a dozen people owning a service and teams of about six owning six services, and regretted that the word 'micro' focuses attention on size. Ford, Richards, Sadalage and Dehghani give a better tool in *Software Architecture: The Hard Parts* (2021): weigh the forces that push towards splitting a service, which they call **granularity disintegrators**, against those that push towards keeping it together, the **integrators**.

Forces on service granularity



Forces from Ford, Richards, Sadalage and Dehghani, Software Architecture: The Hard Parts (2021)

Six disintegrators argue for smaller services; four integrators argue for larger ones. Granularity is a balance, decided per case.

Granularity disintegrators and integrators, with food-delivery examples

Force	Direction	Ask	Example
Service scope and function	Split	Does the service do one cohesive thing, or several unrelated things?	A Notifications service that also calculates loyalty points is doing two jobs.
Code volatility	Split	Does one part change far more often than the rest?	Promotions rules change weekly; the rest of Pricing changes quarterly.
Scalability and throughput	Split	Do parts need very different capacity?	Courier location updates arrive thousands of times a second; courier onboarding a few times a day.
Fault tolerance	Split	Would a failure in one part needlessly take down another?	A crash in the PDF receipt generator should not stop order placement.
Security	Split	Do parts need different access controls or audit?	Stored card tokens versus public menu data.
Extensibility	Split	Will new variants be added often?	New payment methods every few months, each a separate deployable.
Database transactions	Keep together	Must these changes succeed or fail atomically?	Creating an order and reserving its delivery slot.
Workflow and choreography	Keep together	Would splitting create long chains of calls for ordinary requests?	Basket, checkout and order status call each other on every request.
Shared code	Keep together	Do the parts share significant business logic that changes together?	Menu and Pricing share complex dish-option rules.
Data relationships	Keep together	Are the data tightly related, with joins and foreign keys?	Orders and order lines; splitting them gains nothing.

WORKED EXAMPLE

Worked example: should Pricing be split?

The food-delivery platform has a Pricing service that calculates basket totals: dish prices, option surcharges, delivery fees (which depend on distance and demand) and promotions (vouchers, first-order discounts, restaurant offers). The promotions team ships changes several times a week, and each Pricing release needs a full regression run of fee calculation. Someone proposes splitting Pricing into Fees and Promotions services. The architect weighs the forces.

Code volatility (split)	Strong: promotions change weekly, fee rules quarterly. Separating them would let promotions ship without re-testing fees.
Scalability (split)	Weak: both are called once per basket view, at the same rate.
Fault tolerance (split)	Moderate: a bad promotion rule has crashed Pricing twice. If Promotions were separate, a failure could be degraded to 'no discount shown' while fees still work.
Security (split)	Weak: same data sensitivity.
Database transactions (keep)	Weak: no write needs to span both; a voucher is redeemed when the order is placed, in Ordering.
Workflow (keep)	Moderate: every basket total needs both. After a split, Pricing (or a BFF) must call Promotions synchronously on every basket view, adding a hop and its failure modes.
Shared code (keep)	Moderate: both need the basket model and currency rounding rules, which change rarely.
Data relationships (keep)	Weak: promotions refer to restaurants and dishes by identifier only.
Decision	Split, with care: extract Promotions; Pricing calls it with a 150 ms timeout and, on failure, prices the basket without discounts and flags it for re-pricing at checkout. Share only a small, versioned basket-model library.

The decision rests on two strong disintegrators (volatility and fault isolation) outweighing moderate integrators. Had promotions needed to update the same rows as fee calculation in one transaction, the answer would probably have been different.

Writing the forces down turns a matter of taste into an argument others can check, and the record belongs in an architecture decision record.

Anti-patterns

Most failed microservice programmes fail in recognisable ways. The most serious is the distributed monolith: a system with all the costs of distribution and none of the independence, because its services share a database, call each other in long synchronous chains or depend on the same deployable libraries.

Microservice anti-patterns

Anti-pattern	Symptom	Cause and remedy
Distributed monolith	Services must be released together in a fixed order; one service's outage stops all of them.	Shared database, synchronous chains and shared deployable libraries. Remedy: redraw boundaries around bounded contexts, give each service its data, move calls to events, or merge back into a modular monolith.
Entity services	Services named after nouns (Customer, Order, Restaurant) with create, read, update and delete operations; every business process calls several of them.	Boundaries drawn around data rather than behaviour. Michael Nygard calls this the entity service anti-pattern. Remedy: organise around business capabilities and processes.
Chatty services	One user request causes dozens of calls between services; latency and failure rates rise with every feature.	Boundaries cut through a cohesive workflow, or interfaces are too fine. Remedy: coarser APIs, local copies of reference data, merging services that always change together.
Shared libraries that couple deployments	A change to a common library forces every service to rebuild and redeploy; versions drift apart.	Business logic or domain models placed in shared code. Remedy: share only stable, technical libraries; duplicate small amounts of business code; version anything shared and allow several versions in use.
Microservice envy	A small team with a new product starts with thirty services and spends most of its time on infrastructure.	Choosing the style for fashion or CV value rather than drivers. Remedy: start with a modular monolith and extract services when a driver appears.
Nanoservices	Services so small that most of their code is plumbing; a feature needs changes in five of them.	Over-applying 'small'. Remedy: use the granularity integrators; size services by cohesion, not lines of code.

Characteristics of microservices

Characteristic	Rating	Reason
Deployability	High	Each service is released on its own, many times a day, with a small blast radius.
Scalability and elasticity	High	Each service scales independently; small services start quickly and scale in fine steps.
Fault tolerance	High (if designed for)	Failures can be isolated per service, but only with timeouts, bulkheads, fallbacks and asynchronous communication; otherwise faults cascade.
Evolvability	High	Small, well-bounded services can be rewritten or replaced individually.
Testability of one service	High	Each service is small; contract tests check its interactions.
Testing the whole	Low	End-to-end behaviour emerges from many services and is hard to reproduce in a test environment.
Performance	Low to medium	Network hops, serialisation and security checks add latency to every interaction.
Data consistency	Low	No shared transactions; business invariants spanning services rely on eventual consistency and sagas.
Simplicity	Low	Many moving parts, many failure modes, much infrastructure.
Overall cost	Low (expensive)	Platform, tooling, observability and people to run dozens of services.

When microservices fit, and when they do not

Microservices fit organisations with many teams working on one product who need to release independently, systems whose parts have genuinely different scalability, availability or security needs, and domains that are understood well enough to draw stable boundaries. They need investment in a platform and in operational skills before the first service is split off.

They do not fit a small team, a new product whose domain is still being discovered (boundaries drawn too early are expensive to move across the network), a system whose core workflows need strong transactional consistency across most of its data, or an organisation without automated deployment and monitoring. Fowler's *MonolithFirst* advice (Chapter 5) and the reversals that some companies have documented, such as Segment's 2018 consolidation of more than 140 destination services back into one, show that the style is a tool with a cost, not a destination.

NOTE

Practitioners disagree, usefully

Some experienced architects argue that most organisations should never go beyond a modular monolith or a service-based architecture; others argue that small services, with a good platform, are the most productive way for many teams to work. Both camps agree on the conditions: clear domain boundaries, independent deployability, data ownership and automation. Where those conditions are missing, the number of services is not the problem to solve.

CHECK YOUR UNDERSTANDING

▼ Two services are deployed in separate containers but must always be released together. Are they independently deployable?

No. Independent deployability is about being able to change and release one without the other; a coupled contract or shared database makes them one unit for release purposes.

▼ What is the difference between an API gateway and a backend for frontend?

A gateway is a general entry point handling cross-cutting concerns such as TLS, authentication, rate limiting and routing; a BFF is a client-specific back end that shapes data for one kind of front end and is owned by that front end's team.

▼ Which granularity integrator most strongly argues against splitting two parts that must update data atomically?

Database transactions: splitting them would replace one ACID transaction with a distributed workflow such as a saga.

▼ Why are services named Customer, Order and Restaurant with only create, read, update and delete operations a warning sign?

They are entity services: boundaries around data rather than behaviour, so every business process must call several services, creating chatty, tightly coupled interactions.

▼ **Try it: A team proposes CustomerService, OrderService and OrderLineService, each wrapping one table. Placing an order needs one order and five lines written together. Using disintegrators and integrators, is OrderLineService a good split? What would you do?**

No. It is an entity service around a table, not a business capability. Orders and lines share a transaction, a workflow and data relationships, all strong integrators, and no disintegrator such as different scaling, volatility or security applies. Split, it would be chatty, with a saga needed for what one local transaction does now. Merge lines into an Ordering service aligned with its bounded context. Split only when real forces appear.

SOURCES

- James Lewis and Martin Fowler: Microservices (2014): https://martinfowler.com/articles/microservices.html
- Martin Fowler: Microservice Prerequisites (2014): https://martinfowler.com/bliki/MicroservicePrerequisites.html
- Sam Newman: Pattern: Backends For Frontends: https://samnewman.io/patterns/architectural/bff/
- Michael Nygard: The Entity Service Antipattern: https://michaelnygard.com/blog/2017/12/the-entity-service-antipattern/
- Thoughtworks: Software Architecture: The Hard Parts, chapter 7 on service granularity (sample PDF): https://www.thoughtworks.com/content/dam/thoughtworks/documents/books/bk_software_architecture_hard_parts_ch7_en.pdf
- Cam Jackson: Micro Frontends (martinfowler.com, 2019): https://martinfowler.com/articles/micro-frontends.html
- web.dev: Rendering on the Web: https://web.dev/articles/rendering-on-the-web
- webpack: Module Federation: https://webpack.js.org/concepts/module-federation/
- Apple: Release a version update in phases: https://developer.apple.com/help/app-store-connect/update-your-app/release-a-version-update-in-phases/
- Google Play Console Help: staged rollouts: https://support.google.com/googleplay/android-developer/answer/6346149
- Android Developers: Build an offline-first app: https://developer.android.com/topic/architecture/data-layer/offline-first
- MDN: Offline and background operation: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Offline_and_background_operation
- MDN: Background Synchronization API: https://developer.mozilla.org/en-US/docs/Web/API/Background_Synchronization_API

SECTION 6.4 OF 6.8 · ADVANCED · 28 MIN READ · LAST REVIEWED NOVEMBER 2026

Event-driven architecture: brokers, mediators and event streams

BEFORE YOU READ THIS

This section builds on:

- 0.5 Storing data
- 0.6 APIs and integration
- 6.1 Why distribute

Terms it uses from earlier: Asynchronous messaging, Idempotent, Transaction, Bounded context, Coupling

THE IDEA IN BRIEF

Think of a busy restaurant kitchen. When a waiter pins an order ticket on the rail, nobody is told what to do: the grill cook sees there is a steak to cook, the salad station sees a salad, and the pastry chef sees a dessert for later. Each station reacts to the ticket in its own time. If the pastry chef is on a break, the ticket waits on the rail until they return. Adding a new station does not change how waiters write tickets. An event-driven system works the same way: parts announce what has happened, and other parts react.

WHAT YOU NEED TO KNOW

- An event is a fact that has happened (past tense, many possible receivers); a command is a request to one handler that may be refused; a message is the envelope that carries either.
- Broker topology spreads a workflow across independent reactions (choreography); mediator topology puts the sequence in one coordinator (orchestration). Choreography couples least; orchestration makes workflows visible and failures easier to handle.
- Fowler distinguishes event notification, event-carried state transfer, event sourcing and CQRS; the first two are ways for services to communicate, the last two are ways for one service to store data.
- In Kafka a topic is split into partitions; events with the same key go to the same partition, ordering is guaranteed only within a partition, and each consumer group divides partitions among its members and tracks its own offsets.
- Delivery can be at-most-once, at-least-once or effectively-once; exactly-once delivery is not achievable in general, so assume at-least-once and make every consumer idempotent.

- Published events are long-lived contracts: give them identifiers, versions and timestamps, evolve them compatibly and check them with a schema registry.
- Retry transient failures with back-off, move poison messages to a monitored dead-letter queue, and never let one bad message block a partition.

Definition: reacting to things that happen

In event-driven architecture (EDA), components communicate by producing and reacting to events: records of facts that have happened in the business, such as *an order was placed* or *a courier was assigned*. The component that publishes an event does not call the components that react to it and usually does not know they exist. A message broker carries events from producers to consumers, holds them while consumers are busy or down, and lets new consumers be added without changing the producer.

Event-driven designs are not new: graphical user interfaces, operating systems and trading systems have worked this way for decades, and enterprise messaging products predate the web. What changed in the 2010s was scale and durability. Log-based brokers such as Apache Kafka, created at LinkedIn and open-sourced in 2011, made it practical to keep every business event for days or years and to replay them. EDA can be the main style of a system or, more commonly, the way the services of a microservices or service-based system talk to each other.

IF YOU ARE NEW TO THIS

If you are new to this

A *message broker* is a program that sits between other programs and passes messages along, a little like a post office. A sender hands a message to the broker and carries on with its work; the broker stores it safely until each receiver is ready to collect it. Because the sender does not wait, it does not matter if a receiver is busy or briefly offline. Chapter 0 introduced this idea as asynchronous messaging through a queue; this section shows how whole architectures are built around it.

Events, commands and messages

Three words are often used loosely, and the difference matters. A message is the envelope: any data sent from one program to another through a channel. What the message carries can be an event (a fact, named in the past tense), a command (a request addressed to one handler, named as an instruction) or a query and its reply. Gregor Hohpe and Bobby Woolf's *Enterprise Integration Patterns* (2003) catalogued these message types and the channels that carry them, and their vocabulary is still the standard one.

Event, command and message

	Event	Command	Message (the envelope)
Meaning	A fact: something that has happened	A request: please do something	Any data sent from one program to another through a channel
Naming	Past tense: `OrderPlaced`, `PaymentAuthorised`	Imperative: `AuthorisePayment`, `AssignCourier`	Carries an event, a command, a query or a reply
Receivers	Zero, one or many; the publisher does not know them	Exactly one intended handler	Depends on the channel: a queue (one) or a topic (many)
Can it be refused?	No; it has already happened. Receivers decide how to react	Yes; the handler may reject or fail it	Not applicable
Coupling	Publisher does not depend on consumers	Sender depends on the receiver's ability and contract	Depends on what it carries

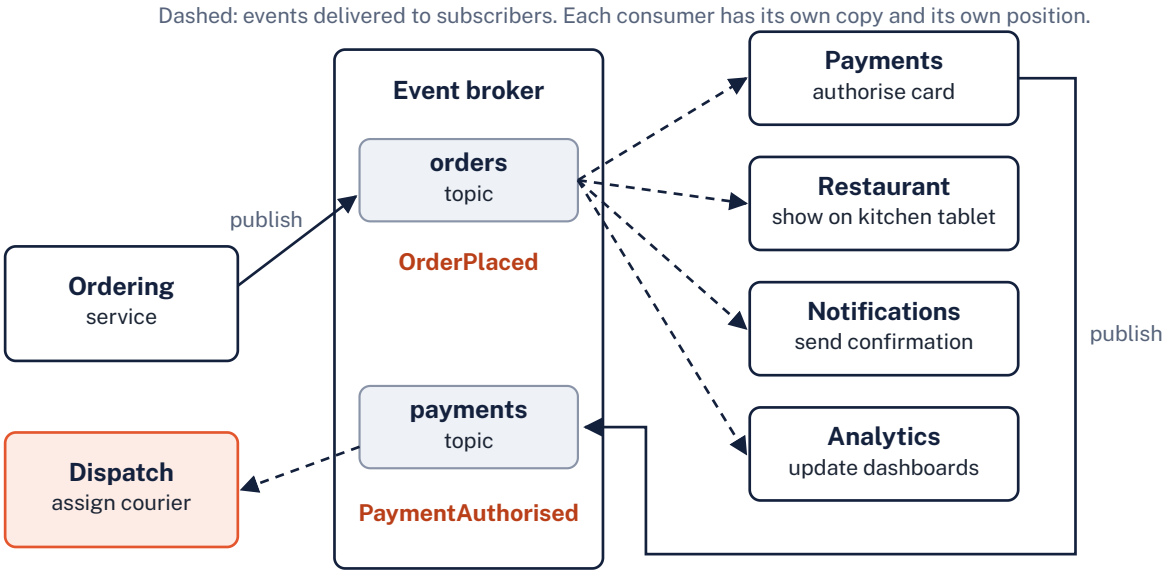
Messages travel through two kinds of channel. A queue delivers each message to one of its consumers, which suits commands and work distribution. A topic delivers a copy of each message to every subscriber, which is publish-subscribe (pub-sub) and suits events. A design that sends `OrderPlaced` to a topic lets Payments, Restaurant, Notifications and Analytics all react, each in its own time; adding a fraud-detection service next year means one new subscription and no change to Ordering.

Broker topology and mediator topology

Richards and Ford describe two topologies for event-driven systems. In the broker topology there is no central coordinator. An initiating event is published; every interested processor reacts and may publish further events, which others react to in turn. The workflow exists only as the sum of these reactions. This is choreography: like dancers who each know their own steps, no one directs the whole.

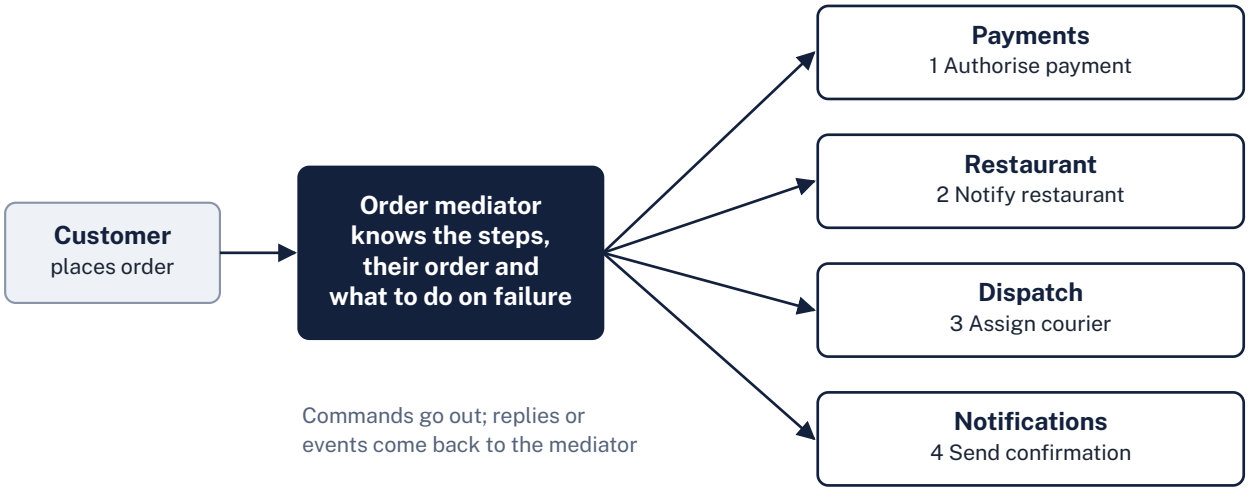
In the mediator topology an event mediator receives the initiating event and directs the workflow, sending commands to processors in a defined order and acting on their replies. This is orchestration: a conductor knows the score. Mediators range from a simple service with a state machine to a workflow engine such as Temporal, AWS Step Functions or a BPMN engine.

Broker topology: the OrderPlaced flow



In broker topology Ordering only announces the fact; each consumer reacts independently, and Payments' own event drives the next step.

Mediator topology: the same flow, orchestrated



In mediator topology one component owns the sequence, so the workflow is visible and failures are handled in one place.

Choreography or orchestration?

Question	Choreography (broker)	Orchestration (mediator)
Coupling	Lowest: publishers do not know consumers	Mediator depends on every participant's contract
Seeing the whole workflow	Hard: it is spread across services; needs tracing to reconstruct	Easy: it is written down in one place
Error handling	Each participant must detect and react to failures, often by publishing compensating events	The mediator decides retries, timeouts and compensations centrally
Adding a step	Add a subscriber; nothing else changes	Change the mediator
Risks	Cyclic event chains; nobody notices when a step silently stops	The mediator becomes a bottleneck or accumulates business logic
Good fit	Simple, stable flows; many independent reactions to one fact	Long, conditional workflows with compensation, such as the order saga

Most real systems mix both: choreography between bounded contexts (Ordering announces facts and does not care who listens) and orchestration inside a context where a multi-step process needs a clear owner.

Four ways to use events

In a 2017 article Martin Fowler pointed out that 'event-driven' is used for at least four quite different patterns, and that conversations go wrong when people mean different ones. Telling them apart is one of the most useful habits an architect can have.

Fowler's four event patterns (2017)

Pattern	What the event carries and why	Benefit	Cost
Event notification	A small signal that something changed, often just an identifier: `OrderPlaced {order_id}`. Receivers call back for details if they need them.	Very low coupling; the publisher's model stays private.	Callbacks add load and a runtime dependency on the publisher; the flow across many events is hard to see.
Event-carried state transfer	Enough data for receivers to keep their own copy: the full order with items and totals.	Receivers work without calling the publisher, even when it is down; lower latency.	Data is copied widely and is eventually consistent; larger events; schemas become contracts.
Event sourcing	Every change to an entity's state is stored as an event; the event log is the source of truth and current state is derived by replaying it.	Full audit trail; rebuild past states; new views from old history.	Complex; replay is hard when results depended on external systems; event schemas must evolve for ever.
CQRS	Separate models (often separate stores) for writing and for reading, frequently fed by events.	Each side optimised for its job; suits read-heavy systems with complex queries.	Two models to keep in step; often applied where one model would do.

Event notification and event-carried state transfer are about communication between services and belong in this chapter. Event sourcing and CQRS are about how one service stores its data; they are often combined with EDA but are independent choices, covered in Chapter 7.

Event streams and logs: how Kafka works

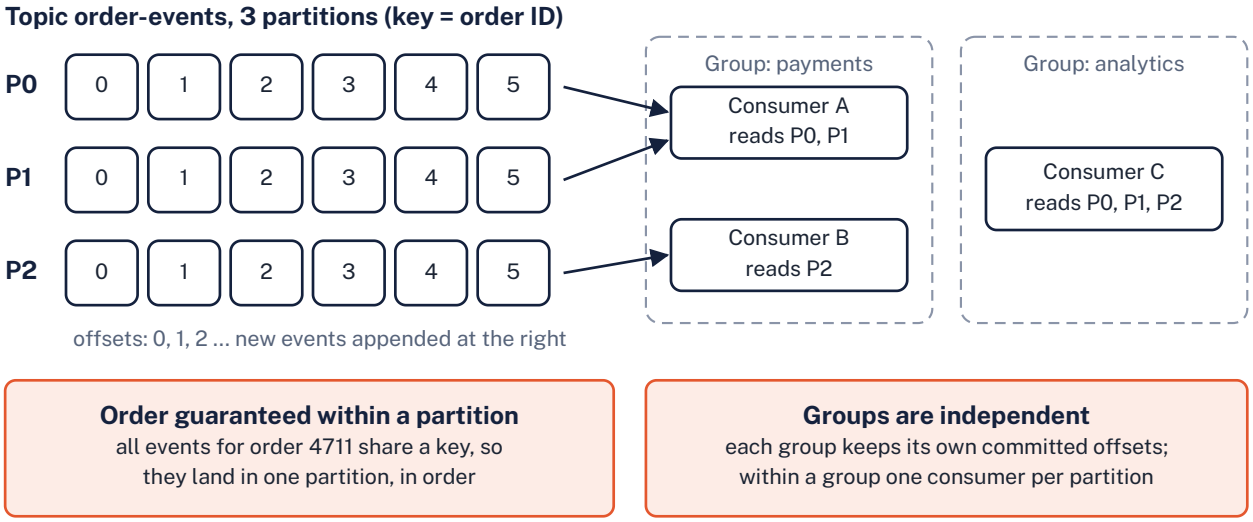
Traditional message queues delete a message once a consumer acknowledges it. A log-based broker such as Apache Kafka instead stores events in an append-only log and keeps them for a configured retention period (or for ever) whether or not they have been read. Several things follow from that design, and they explain both Kafka's strengths and its traps.

A topic is split into partitions, each an ordered log stored and replicated across brokers. A producer chooses the partition, normally by hashing the event's key, so all events with the same key (the same order ID, say) go to the same partition. Each event in a partition has a sequential position called its offset. Consumers pull events and record the offset they have reached; they can also rewind and replay.

Consumers join a consumer group. Within one group, each partition is read by exactly one consumer at a time, so the group shares the work; different groups read independently, each with its own offsets, which is how pub-sub is achieved. The number of partitions therefore limits the parallelism of one group: a topic with 12 partitions can keep at most 12 consumers in a group busy. (Kafka's *share groups*, previewed in earlier 4.x releases and production-ready from Kafka 4.2 in 2026, are a queue-like alternative: several consumers can share one partition and acknowledge messages individually, but ordering within the partition is then not preserved. This section describes classic consumer groups.)

Kafka guarantees ordering only within a partition. Events for one order arrive in the order they were written if they share a key; events for different orders, in different partitions, have no defined order relative to each other. Designs that need a global order must use a single partition (giving up parallelism) or, better, be redesigned so that only per-entity order matters.

Partitions, offsets and consumer groups



Each partition is an ordered log; a consumer group divides partitions among its members, and each group tracks its own offsets.

Delivery guarantees, and what 'exactly once' really means

Because networks and processes fail (Section 6.1), a broker and its clients must choose what happens when a crash or timeout leaves it unclear whether a message was processed. There are three possible guarantees, and the choice is made by when acknowledgements and offsets are recorded.

Delivery guarantees

Guarantee	What it promises	How it arises	Typical consequence
At-most-once	Each message is processed once or not at all; never twice.	The consumer records its position (commits the offset or acknowledges) before processing, or the producer never retries.	Messages can be lost after a crash. Acceptable for metrics samples or courier location pings, where the next update supersedes the last.
At-least-once	Each message is processed one or more times; never lost.	The producer retries until acknowledged, and the consumer commits its position only after processing.	Duplicates after crashes, retries and rebalances. The default for most brokers and the usual choice for business events.
Effectively-once	Each message's effect is applied exactly once, even though it may be delivered more than once.	At-least-once delivery plus an idempotent consumer, or a transaction that stores the result and the consumer's position together.	The practical meaning of 'exactly once' in business systems; requires deliberate design in every consumer.

'Exactly once' needs careful definition. Exactly-once *delivery* over an unreliable network is not achievable in general: a sender that gets no acknowledgement cannot know whether to resend. What systems can provide is exactly-once *effect* within a defined boundary. Kafka, for example, has offered an idempotent producer and transactions since version 0.11 (2017), so that a read-process-write cycle from Kafka topics to Kafka topics commits its output and its consumer offsets atomically. That guarantee stops at Kafka's edge. If the consumer charges a card, sends an email or writes to another database, a crash between that action and the offset commit will lead to the action being repeated, unless the consumer itself is idempotent or stores its offsets in the same transaction as its results.

The practical rule for architects is therefore simple: assume at-least-once delivery and make every consumer an idempotent consumer. Give each event a unique identifier; in the consumer, record processed identifiers in the same local transaction as the state change, and skip any event already recorded. Where possible, also make the operation naturally idempotent, for example 'set status to PAID if it is AWAITING_PAYMENT' rather than 'add one payment'.

```
# Idempotent consumer: processing the same event twice has the effect of once
def handle_payment_authorised(event, db):
    with db.transaction():                       # one local ACID transaction
        inserted = db.execute(
            "INSERT INTO processed_events (event_id) VALUES (%s) "
            "ON CONFLICT (event_id) DO NOTHING",
            [event["event_id"]],
        ).rowcount
        if inserted == 0:
            return                               # seen before: do nothing
        db.execute(
            "UPDATE orders SET status = 'PAID' WHERE order_id = %s "
            "AND status = 'AWAITING_PAYMENT'",
            [event["order_id"]],
        )
    # commit the broker offset only after the transaction has committed
```

An idempotent consumer: the processed-events record and the state change commit together, so a redelivered event changes nothing.

Event schemas and versioning

Once other teams consume an event, its structure is a published contract, just like an API, and it lasts longer: with a log-based broker, consumers may read events written months ago. A good event has a unique identifier, a type, a schema version, the time it occurred, the identifiers of the entities involved and the business data consumers need. Money is stored in minor units with a currency, never as a floating-point number (Chapter 0).

```
{
  "event_id": "9f1c2e7a-4b0d-4e55-9a51-0c3d8f2b6a10",
  "type": "OrderPlaced",
  "schema_version": 2,
  "occurred_at": "2026-10-22T18:42:07Z",
  "order_id": "ord-4711",
  "customer_id": "cus-208",
  "restaurant_id": "res-31",
  "items": [{"dish_id": "d-5", "qty": 2, "price_minor": 1150}],
  "total_minor": 2450,
  "currency": "GBP"
}
```

An OrderPlaced event using event-carried state transfer: identifiers, version, time and the data consumers need.

Schemas should evolve in backward-compatible ways: add optional fields, never remove or rename fields or change their meaning without a new version, and never reuse a field name for something different. Many teams enforce this with a schema format such as Avro, Protocol

Buffers or JSON Schema and a schema registry that rejects incompatible changes at publish time. The CloudEvents specification, a graduated Cloud Native Computing Foundation project, standardises the common envelope attributes such as id, source, type and time. Compatibility modes and migration techniques are covered in schema evolution.

Error handling: retries, poison messages and dead-letter queues

Consumers fail for two very different reasons, and they need different treatment. **Transient failures**, such as a database that is briefly unavailable or a timeout calling another service, usually succeed on retry; retry a few times with exponential back-off and jitter (increasing, randomised delays) so that a struggling dependency is not overwhelmed. **Permanent failures**, such as a malformed event or one that violates a business rule, will fail every time. Such a 'poison message' must not block the consumer for ever, which on a partitioned log would stall every later event in that partition.

The standard answer is a dead-letter queue (DLQ): after a set number of attempts the message is moved to a separate channel, with the error recorded, and the consumer moves on. A DLQ is only useful if someone watches it: alert on its size, give a named team ownership, provide tools to inspect, fix and replay messages, and decide in advance what ordering guarantees are lost when a message is replayed late. Some teams add one or more intermediate retry topics with longer delays before the DLQ.

Characteristics of event-driven architecture

Characteristic	Rating	Reason
Scalability and elasticity	High	Consumers scale independently and absorb bursts from the broker's buffer; partitions allow parallelism.
Responsiveness	High	The publisher returns as soon as the event is durably recorded; slow work happens later.
Fault tolerance	High	A consumer can be down for minutes and catch up from the broker; failures do not cascade back to the publisher.
Extensibility	High	New consumers subscribe to existing events without any change to the publisher.
Data consistency	Low	State spreads across services with a delay; readers see stale data for a while.
Testability and debuggability	Low	Behaviour emerges from asynchronous reactions; reproducing a sequence of events across services is hard without good tracing.
Workflow visibility	Low (choreography) to medium (mediator)	Nobody owns the end-to-end flow in choreography; a mediator makes it explicit.
Simplicity	Low	Brokers, schemas, retries, dead-letter handling, ordering and duplicates must all be designed.

When to use it, and common mistakes

Event-driven architecture fits when many parts of a business must react to the same facts, when work can happen after the user's request returns, when load is bursty and must be smoothed, and when new consumers are expected over time. It fits less well when a user needs an immediate, definitive answer that depends on several services (although a synchronous call for the answer plus events for the follow-up is a common hybrid), or when strong consistency across services is required.

WATCH OUT

Common mistakes

- **Commands disguised as events.** `SendConfirmationEmail` published on a topic is a command with the coupling hidden; publish `OrderPlaced` and let Notifications decide.
- **Assuming exactly-once.** A consumer that charges a card or sends an email without an idempotency check will eventually do it twice.
- **Assuming global order.** Events in different partitions, or from different producers, can arrive in any order; design per-entity ordering with keys and version numbers.
- **Writing to the database and publishing separately.** If the service crashes between the two, the event is lost or announces something that never happened. The transactional outbox pattern solves this.
- **Event spaghetti.** Dozens of services reacting to each other's events with no map of the flows. Keep an event catalogue, trace with correlation identifiers, and use a mediator where a workflow needs an owner.

CHECK YOUR UNDERSTANDING

▼ **Is `ReserveCourier` an event or a command, and how would you know?**

A command: it is an instruction addressed to one handler (Dispatch), which may refuse it. An event would be named in the past tense, such as `CourierReserved`.

▼ **A Kafka topic has 6 partitions and a consumer group has 10 consumers. How many consumers will be busy?**

At most 6. Within a group each partition is assigned to one consumer, so 4 consumers sit idle.

▼ **Why does Kafka's exactly-once support not prevent a consumer from sending a confirmation email twice?**

Its transactions cover reading from and writing to Kafka topics and committing offsets; an email sent to an external system is outside that boundary, so a crash before the offset commit causes reprocessing unless the consumer is idempotent.

▼ **What is the difference between event notification and event-carried state transfer?**

Notification sends a small signal, often only an identifier, so receivers call back for details; state transfer includes the data receivers need, so they can keep their own copy and avoid calling the publisher.

▼ **Try it: A consumer sometimes sees OrderCancelled before OrderPaid for one order. The topic has 12 partitions and events have random keys. It also occasionally processes an event twice. Diagnose both problems and give fixes.**

Ordering holds only within a partition, and random keys scatter one order's events across partitions. Key events by order ID, so all of an order's events land in the same partition, in order. Duplicates are normal under at-least-once delivery, so make the consumer idempotent, for example by recording processed event IDs in the same transaction as its update. Include a version number, and send bad messages to a monitored dead-letter queue rather than blocking the partition.

SOURCES

- Martin Fowler: What do you mean by "Event-Driven"? (2017): https://martinfowler.com/articles/201701-event-driven.html
- Apache Kafka: Introduction (topics, partitions and ordering): https://kafka.apache.org/intro
- Confluent documentation: Kafka message delivery guarantees: https://docs.confluent.io/kafka/design/delivery-semantics.html
- Enterprise Integration Patterns: Idempotent Receiver: https://www.enterpriseintegrationpatterns.com/patterns/messaging/IdempotentReceiver.html
- CloudEvents specification: https://cloudevents.io/

Synchronous and asynchronous communication: REST, gRPC, GraphQL and messaging

BEFORE YOU READ THIS

This section builds on:

- 0.6 APIs and integration
- 6.1 Why distribute
- 6.4 Event-driven

Terms it uses from earlier: HTTP, HTTP status code, Partial failure, Breaking change, Publish-subscribe

THE IDEA IN BRIEF

Services talk in two ways. A phone call is synchronous: you wait on the line until you get an answer, and if the other person is not there you get nothing. A letter is asynchronous: you post it and get on with your day, and the reply comes later. Phone calls are simple but tie both people to the same moment; letters are more flexible but you do not know at once whether your request worked. On top of that, the two sides need a shared, written agreement about what can be asked and what comes back, and a way to change that agreement without upsetting the people who rely on it.

WHAT YOU NEED TO KNOW

- Synchronous calls create temporal coupling: caller and callee must be up together, so latencies add and availabilities multiply along a chain; asynchronous messaging removes that coupling but not format or semantic coupling.
- REST is Fielding's set of constraints (client-server, stateless, cacheable, uniform interface, layered system, optional code on demand); most real APIs reach level 2 of Richardson's maturity model, not level 3 hypermedia.
- RFC 9110 defines GET, HEAD, OPTIONS, PUT and DELETE as idempotent and POST as neither safe nor idempotent; clients should retry automatically only idempotent requests after a timeout, 502, 503 or 504, so POST needs an idempotency key before it can be retried safely.
- gRPC uses Protocol Buffers over HTTP/2 with generated code and streaming and suits internal calls; GraphQL lets diverse clients choose fields but needs batching, query-cost limits and field-level authorisation.

- Queues deliver each message to one competing consumer; topics deliver a copy to every subscriber.
- Durable APIs model domain resources, paginate with cursors, return RFC 9457 problem details, change only in backward-compatible ways and version or use expand and contract when they must break.
- OpenAPI and AsyncAPI describe HTTP and message contracts; consumer-driven contract tests such as Pact let providers know which changes would break which consumers.

Two ways to talk, and the coupling each creates

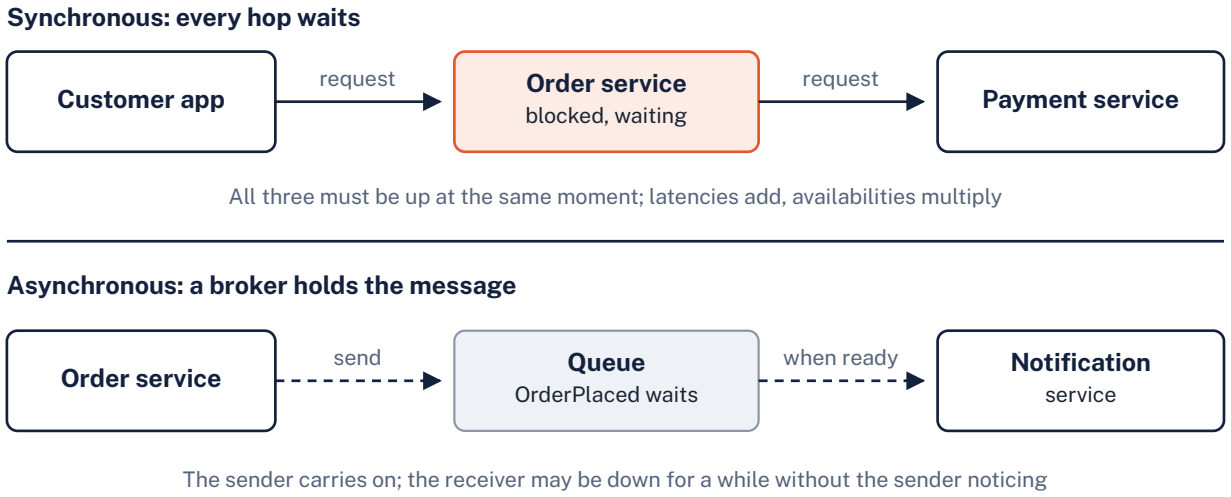
Once a system is split into separately deployed parts (see why distribute), those parts must talk over a network, and the way they talk shapes the system as much as the way they are divided. There are two basic modes. In synchronous communication the caller sends a request and waits for the reply before it continues, like a telephone call. In asynchronous communication the sender hands a message to an intermediary, usually a message broker, and carries on; the receiver picks the message up when it is ready, like a letter in a post box. Section 0.6 introduced the difference; here it is examined as an architectural decision.

Synchronous calls create temporal coupling: both parties must be running and reachable at the same moment. The caller is therefore at best as available as the product of its dependencies' availabilities, and its response time, including its tail latency, contains theirs: the arithmetic of Section 6.1. A slow payment service makes the order service slow, and if all the order service's threads are waiting on it, the order service stops answering anybody. Timeouts, retries and circuit breakers limit the damage and are covered in resilience patterns.

Asynchronous messaging removes temporal coupling: the broker stores the message, so the receiver can be slow, busy or briefly down without the sender noticing. It does not remove every kind of coupling. Both sides still depend on the message's format and meaning (format and semantic coupling), and on the broker itself, which becomes critical infrastructure. The sender also loses immediate knowledge of the outcome, so the design must cope with work that fails later, messages that arrive twice and messages that arrive in a different order. Asynchrony moves complexity rather than deleting it.

A useful rule of thumb: use a synchronous call when the caller genuinely cannot continue without the answer ("is this card authorised?"), and asynchronous messaging when the caller only needs to know that the work will happen ("send the receipt", "tell the restaurant"). Many good designs keep the user-facing path short and synchronous and push everything else behind a queue.

Temporal coupling, with and without a broker



In a synchronous chain the caller is blocked and depends on every service downstream; with a broker in between, sender and receiver no longer need to be available at the same moment.

IF YOU ARE NEW TO THIS

Step by step: a phone call or a letter?

This section's hardest idea is what posting a message, instead of waiting for an answer, does and does not free you from. Chapter 0 introduced the two modes; follow one food order through both.

- A customer taps Place order. The order service must know whether the card is authorised before it can say yes, so it calls the payment service and waits: a synchronous call, like a phone call.
- If the payment service is down, the order cannot be placed: both must be running and reachable at the same moment. That is temporal coupling.
- Next, the customer needs a receipt by email. The order service does not need to wait for that, so it leaves a ReceiptNeeded message with a message broker, a kind of post box, and answers the customer straight away: asynchronous, like a letter.
- Suppose the email service is down for ten minutes. The broker keeps the message and the receipt simply arrives late. Had the order service phoned the email service and waited, every order in those ten minutes would have failed or hung.
- The letter does not free the two completely. If the order service renames a field in the message, say from email to emailAddress, the email service can no longer read it: both sides still rely on the same agreed layout and meaning of what is sent. And the order service no longer learns at once whether the email went.
- **The lesson:** call and wait only when you cannot continue without the answer; post a message when you only need to know that the work will happen. Either way, the format of what is sent is a promise, to be changed with care, as the end of this section explains.

REST as Fielding defined it

REST (Representational State Transfer) was defined by Roy Fielding in his 2000 doctoral dissertation as an architectural style for large, networked hypermedia systems: the web itself. Fielding derived it as a set of constraints, each of which buys a quality. **Client-server** separates user interface from data storage so both evolve independently. **Stateless** interaction means each request carries everything needed to understand it, so any server instance can handle it, which helps scalability and recovery. **Cacheable** responses say whether they can be reused, cutting latency and load. A **uniform interface** (resources identified by URIs, manipulated through representations, self-descriptive messages, and hypermedia as the engine of application state) lets generic intermediaries such as caches and proxies understand traffic. A **layered system** lets proxies, gateways and load balancers sit in the path invisibly. **Code on demand**, such as JavaScript sent to a browser, is the only optional constraint.

Most APIs described as RESTful satisfy only some of these. Leonard Richardson's maturity model, presented in 2008 and popularised by Martin Fowler in 2010, describes the steps. At **level 0**, HTTP is merely a tunnel: every call is a POST to one endpoint with the operation named in the body, as in SOAP. At **level 1**, the API has separate resources (`/orders/1042` , `/restaurants/7`). At **level 2**, it uses HTTP methods and status codes as HTTP defines them. At **level 3**, responses carry hypermedia controls: links telling the client what it can do next (a placed order contains a link to cancel it, until cancellation is no longer allowed).

Practitioners genuinely disagree about level 3. Fielding has insisted that an API without hypermedia is not REST. Most public APIs stop at level 2 because their clients are written against documentation rather than discovered at run time, and the benefit of following links rarely pays for the extra design. A pragmatic reading is that level 2, done consistently, gives most of the value; hypermedia earns its keep when the server must steer many independently written clients through a workflow that changes.

HTTP semantics that matter for architecture

HTTP's semantics are defined in RFC 9110 (2022). Two method properties matter more than any other for distributed design. A **safe** method only reads: GET, HEAD and OPTIONS change nothing the client asked to change. An idempotent method may change state, but repeating it has the same intended effect as sending it once: PUT (replace this resource with this representation) and DELETE are idempotent, as are all safe methods. POST is neither, and PATCH is not idempotent in general. The property matters because networks fail between a request being processed and the response arriving. RFC 9110 says that idempotent requests can be repeated automatically after such a failure, and that clients should not automatically retry non-idempotent requests unless they know the request is in fact safe to repeat.

Status codes are part of the contract, because client code branches on them; Section 0.6 introduced the common ones. Success codes should say what happened: 201 Created (with a `Location` header) when a resource is created, 202 Accepted when work has only been queued, 204 No Content for success with no body. For conflicts, 409 means the request clashes with the resource's current state, 412 that a precondition failed and 422 that a well-formed request is semantically invalid. The architectural rule is about retries: a 4xx response will fail again if the request is sent unchanged (429 Too Many Requests is the exception: wait as told, then retry), while a timeout, 502, 503 or 504 may succeed later. Clients should retry only those, with backoff and jitter, and only when the request is idempotent or carries an idempotency key.

Two further HTTP features solve distributed problems cheaply. **Conditional requests** give optimistic concurrency control: a client reads an order with its version tag (`ETag`), then sends `If-Match` with that tag when updating; if someone else changed the order in between, the

server answers 412 and nothing is overwritten. **Caching headers** (`Cache-Control`, `ETag`) let browsers, CDNs and proxies reuse responses, which is often the cheapest performance tactic available.

HTTP methods: safe, idempotent, or neither (RFC 9110; PATCH from RFC 5789)

Method	Typical use	Safe?	Idempotent?	Safe to retry blindly?
GET, HEAD	Read a resource	Yes	Yes	Yes
PUT	Create or replace at a known URI	No	Yes	Yes
DELETE	Remove a resource	No	Yes	Yes (a second call may return 404, but the effect is the same)
POST	Create in a collection, or run an action	No	No	Only with an idempotency key or another duplicate check
PATCH	Partial update	No	Not in general	Only if the patch is written to be idempotent, or is conditional on a version

gRPC: contracts first, binary on the wire

gRPC is a remote procedure call framework that Google open-sourced in 2015 as the successor to Stubby, the RPC system it had used internally for over a decade; version 1.0 followed in 2016. The contract is written in a `.proto` file using Protocol Buffers, a compact binary serialisation format, and code generators produce typed client stubs and server skeletons in many languages. Calls travel over HTTP/2, which multiplexes many concurrent requests over one connection.

gRPC offers four kinds of call: **unary** (one request, one response), **server streaming** (one request, a stream of responses, such as live rider positions), **client streaming** (a stream of requests, one response, such as uploading telemetry) and **bidirectional streaming**. Clients set a **deadline**; when it passes, the call ends with `DEADLINE_EXCEEDED`, and deadlines can be propagated down a call chain so that work stops everywhere once nobody is waiting for the answer.

Its strengths are efficiency, strong typing and generated code, which make it popular for internal service-to-service calls in polyglot organisations. Its costs are the mirror image. Binary payloads are hard to read in logs and with ordinary tools. Browsers cannot call gRPC directly, so public and web-facing APIs usually need a proxy or a REST or GraphQL facade. HTTP

caching does not apply in the usual way. Compatibility depends on discipline with field numbers: a field's number, not its name, identifies it on the wire, so numbers must never be reused, and removed fields should be marked as reserved.

```
syntax = "proto3";

service Dispatch {
  rpc AssignCourier (AssignRequest) returns (Assignment);
  rpc TrackCourier (TrackRequest) returns (stream Position);
}

message AssignRequest {
  string order_id = 1;
  string restaurant_id = 2;
  reserved 3;            // field removed in v2: never reuse 3
}

message Position {
  double lat = 1;
  double lng = 2;
  int64 recorded_at_ms = 3;
}
```

A gRPC contract for the food-delivery dispatch service, with one unary and one server-streaming call (Protocol Buffers syntax).

GraphQL: the client chooses the shape

GraphQL is a query language and runtime for APIs that Facebook developed internally from 2012 for its mobile apps and released publicly, with a draft specification and reference implementation, in 2015; it moved to the vendor-neutral GraphQL Foundation in 2018. The server publishes a typed **schema**; clients send queries to a single endpoint naming exactly the fields they want, possibly across related objects, and receive a JSON response of the same shape. Mutations change data, and subscriptions push updates.

Its main benefit is for diverse clients. A food-delivery mobile app on a slow connection can ask for an order's status, the restaurant's name and the courier's first name in one round trip, without receiving the twenty other fields the web dashboard needs. That avoids both over-fetching and the chain of requests a resource-by-resource REST API can require, and it can replace a separate backend for each frontend. The schema is also self-documenting and lets tools validate queries before they run.

The trade-offs are real and are mostly paid by the server team. Each field is filled by a **resolver**, so a naive implementation that fetches each order's restaurant separately produces the N+1 query problem: one query for fifty orders, then fifty more for their restaurants. The usual fix is

batching and per-request caching, as popularised by the DataLoader library. Because clients can compose arbitrarily deep queries, servers need query depth and cost limits, timeouts and often an allow-list of approved (persisted) queries to stop accidental or malicious overload. HTTP caching is weaker when everything is a POST to one URL. Authorisation must be enforced per field or per object rather than per URL. And errors are typically reported in an `errors` array inside a successful HTTP response, which monitoring based on status codes will miss.

Messaging: queues and topics

Messaging moves the conversation into a broker. As event-driven architecture explains, a queue delivers each message to one of several competing consumers, the standard way to scale a worker pool, while a topic delivers a copy to every subscriber, so the restaurant, dispatch and notification services can each react to `OrderPlaced` without the order service knowing they exist. That section also covers delivery guarantees, ordering, duplicates and dead-letter queues; queues and back pressure covers how queues absorb load spikes.

Messaging can also carry request and reply: the requester sends a command with a **correlation ID** and a reply-to address, and matches the eventual reply to its request. This keeps the temporal decoupling of a broker while still getting an answer, at the cost of more moving parts.

REST, gRPC, GraphQL and messaging compared

Aspect	REST over HTTP	gRPC	GraphQL	Messaging (queues and topics)
Interaction	Synchronous request-response	Synchronous; unary or streaming	Synchronous queries and mutations; subscriptions push	Asynchronous; fire-and-forget or request-reply via correlation IDs
Contract	OpenAPI document (optional but common)	.proto files (required)	Schema (required)	AsyncAPI document and message schemas (optional but advisable)
Payload	Usually JSON text	Protocol Buffers binary	JSON	Any: JSON, Avro, Protocol Buffers
Temporal coupling	Yes	Yes	Yes	No: the broker buffers
Caching	Excellent: standard HTTP caches and CDNs	Little built in	Harder: needs persisted queries or client caches	Not applicable
Browser support	Native	Needs a proxy (gRPC-Web)	Native	Not directly; via a backend or WebSockets
Best fit	Public and partner APIs; resource-oriented CRUD	Internal, high-volume, low-latency service calls; streaming	Many client types with different data needs; aggregating several services	Work that can happen later; fan-out to many consumers; smoothing load
Main pitfalls	Chatty APIs; inconsistent conventions	Opaque in debugging; field-number discipline	N+1 resolvers; unbounded query cost; field-level authorisation	Duplicates, ordering, invisible flows, broker operations

The messaging landscape: choosing a queue or a log

The first decision is between a queue and a log. A **queue** hands out work and a message is gone once processed, which suits jobs such as sending a receipt, where per-message retry and dead-lettering matter. A **log** keeps messages for later readers, which suits facts that several

consumers need, or that must be replayed, such as order events that rebuild a read model. Section 6.4 explains Kafka, the best-known log. The line has blurred: Kafka 4.2 declared its queue-like share groups production-ready, and RabbitMQ added append-only streams.

Four other names come up often. **RabbitMQ** is a broker you run or buy as a hosted service; its queues suit work distribution, and its streams add replay and large fan-out. **Amazon SQS and SNS** are managed services: consumers poll an SQS queue, and an SNS topic pushes a copy to each subscribed queue, so one event reaches restaurant, dispatch and notification queues that each drain at their own pace. **Google Cloud Pub/Sub** is managed publish-subscribe with at-least-once delivery. **NATS** is lightweight: core NATS is at-most-once, so a subscriber that is not connected never sees the message, while its JetStream layer adds storage, replay and acknowledgements. Limits and features change, so check the current version of each.

Competing consumers and strict order pull against each other. Even if a broker hands out messages in order, workers finish in any order: a slow worker holding `OrderPaid` can finish after a quick one holding `OrderCancelled`. Redelivery adds to this; RabbitMQ's documentation warns that with several active consumers "any redelivery can change order". The usual answer is to give up overall order and keep order per key, with at most one worker per key at a time. Kafka does this with keyed partitions and SQS FIFO queues with message groups: order holds within a group, and groups run in parallel. Standard SQS queues offer only best-effort order. Pub/Sub uses ordering keys, enabled on the subscription. RabbitMQ's single active consumer keeps order by letting only one consumer receive from the queue, which gives up parallelism on it.

Per-key order has a price. One slow or failing message holds up every later message for that key, and a very busy key builds a backlog; Pub/Sub's documentation also says that redelivering one message redelivers the later ones for that key, even those already acknowledged. Ordered delivery does not remove duplicates either, so consumers must still be idempotent. Where a stale update can be recognised, an alternative is to carry a version number in each event and let consumers ignore anything older than they have applied.

Designing APIs that last: resources, pagination and filtering

Model resources around the domain, not the database. Expose the nouns your consumers think in (`/orders`, `/orders/1042/items`, `/restaurants/7/menu`), not tables or internal services. Actions that do not fit create, read, update and delete can usually be modelled as resources themselves: `POST /orders/1042/cancellation` creates a cancellation, which can carry a reason and be refused with 409 if the food is already on its way. This keeps the uniform interface while expressing the business process honestly.

Paginate every collection, because collections grow. **Offset pagination** (`?offset=40&limit=20`) is simple and allows jumping to page 9, but it slows down on deep pages, because the database must skip all earlier rows, and it shows duplicates or skips items when rows are inserted while someone is paging. Cursor pagination returns an opaque token pointing just past the last item (`?after=eyJpZCI6MTA0Mn0&limit=20`); it stays fast and stable under inserts, at the cost of no random access. For feeds and large or changing data, cursors are the safer default. Always cap the page size on the server.

Filtering and sorting should use predictable query parameters (`?status=delivered&placed_after=2026-10-01&sort=-placed_at`) and be backed by indexes; an API that lets clients filter on any field invites queries the database cannot answer quickly. Support only the filters consumers need, and document them in the contract.

Long-running work: 202, polling and reliable webhooks

Some requests take too long to hold open, such as importing a restaurant chain's whole menu. The API checks the request and returns `202 Accepted` with a `Location` header naming a status resource, and perhaps `Retry-After`. RFC 9110 calls 202 "intentionally noncommittal": the work may still be refused later. The client polls, receiving 200 with the state, then 303 to the result. To avoid polling, the server can call a **webhook** instead. A sender should retry with back-off and jitter, keep one event ID across retries and sign each delivery with a timestamp. The receiver should verify the signature (Section 8.4), reply 2xx quickly, queue the work and ignore duplicates. Stripe, for one, does not guarantee order. Keep the status endpoint as a way to recover missed deliveries.

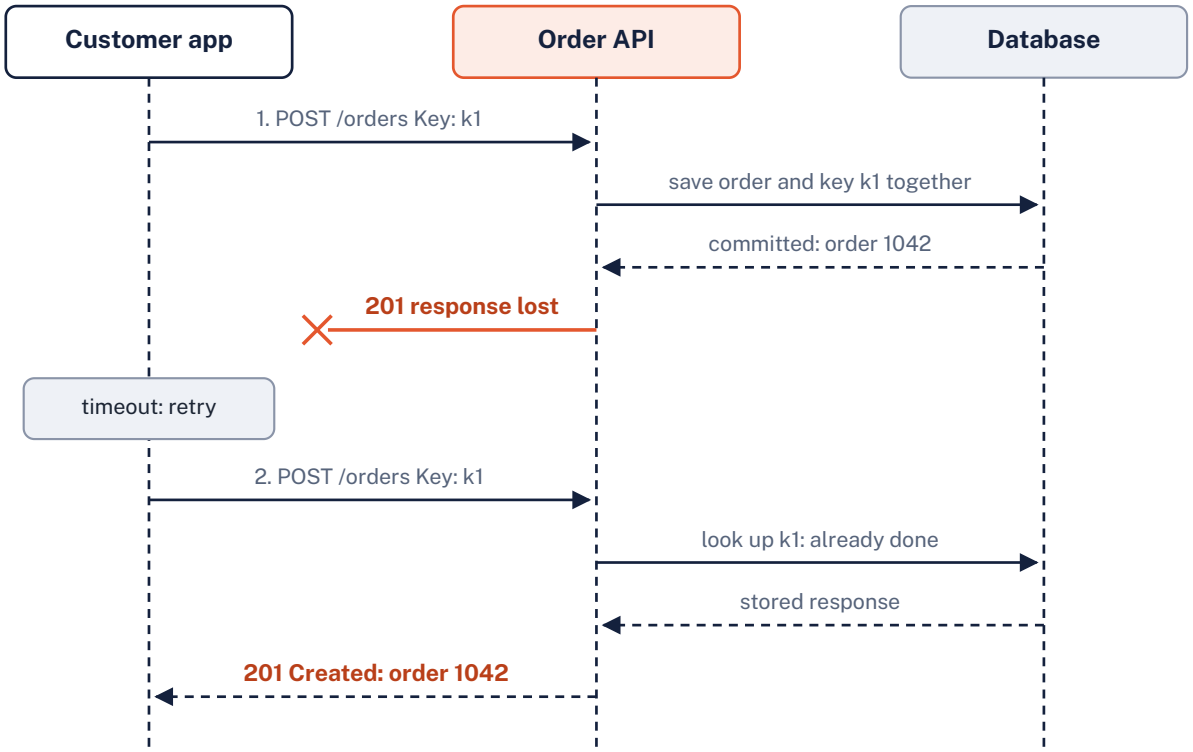
Idempotency keys: making POST safe to retry

Partial failure (Section 6.1) leaves a client that got no answer unsure whether its request was processed. Retrying a GET is harmless; retrying `POST /orders` may create a second order and charge the customer twice. An idempotency key turns the unsafe retry into a safe one. The client generates a unique value (typically a random UUID) for each logical operation and sends it in a header such as `Idempotency-Key`. The server records the key with the outcome of the first request; any later request with the same key returns the recorded outcome instead of doing the work again.

Payment providers popularised the technique, and the IETF HTTP APIs working group has drafted a standard `Idempotency-Key` header. At the time of writing it has not been published as an RFC (the latest draft, from October 2025, has expired), but its rules reflect common practice: a key reused with a different request body should be rejected with 422, a retry that

arrives while the first request is still running should get 409 Conflict so that two copies never run at once, and keys are scoped to a client and expire after a published period, often 24 hours. The worked example applies each rule.

A retried order with an idempotency key



The first attempt succeeds but its response is lost; the retry carries the same key, so the API returns the stored result instead of creating order 1043.

WORKED EXAMPLE

Worked example: the double-tap at checkout

A customer on a weak mobile connection taps Place order. The app generates key `7f3c…` and sends `POST /v1/orders`. The order service stores the key together with a hash of the request body and, later, the response.

Attempt 1	Order 1042 is created and the payment authorised; the key, body hash and response 201 are saved in the same database transaction as the order. The response is lost when the phone switches from Wi-Fi to mobile data.
Attempt 2 (automatic retry, same key)	The service finds the key with status *completed*, checks that the body hash matches, and returns the stored 201 for order 1042. No second order, no second charge.
Attempt 3 (customer taps again while attempt 2 is in flight)	The app reuses the same key for the same basket. If the first request were still processing, the service would answer 409 and the app would wait and retry.
A bug sends the same key with a different basket	Body hashes differ: the service answers 422 rather than silently returning the wrong order.
After 24 hours	The key expires and is deleted; the app never retries that old basket.

The key must be generated once per logical operation (per basket submission), not per HTTP attempt; a fresh key on each retry defeats the purpose.

Saving the key in the same transaction as the order is what makes the check reliable: a crash cannot leave an order without its key, or a key without its order.

The same idea, applied to messages instead of HTTP requests, is the idempotent consumer of event-driven architecture.

```
def create_order(request, db):
    key = request.headers.get("Idempotency-Key")
    if not key:
        return problem(400, "missing-idempotency-key")
    body_hash = sha256(request.body)
    with db.transaction():
        row = db.find_idempotency(request.client_id, key)
        if row and row.body_hash ≠ body_hash:
            return problem(422, "idempotency-key-reused")
        if row and row.status == "in_progress":
            return problem(409, "request-in-progress")
        if row:
            return row.saved_response          # replay, do nothing new
        db.insert_idempotency(request.client_id, key, body_hash, "in_progress")
        order = db.insert_order(request.json)
        response = created(order)
        db.complete_idempotency(request.client_id, key, response)
    return response
```

Server-side handling of an idempotency key (simplified: one database transaction per request; the unique constraint on the key stops two concurrent requests both running).

Errors that machines can read: RFC 9457

Clients need errors they can act on programmatically, and humans need errors they can understand. Rather than every team inventing an error shape, use Problem Details for HTTP APIs, RFC 9457 (July 2023), which replaced RFC 7807 (2016). A problem details object has the media type `application/problem+json` and five standard members: `type` (a URI identifying the kind of problem, which can link to documentation), `title` (a short, human-readable summary of that kind), `status` (the HTTP status code), `detail` (an explanation of this occurrence) and `instance` (a URI identifying this occurrence). Extension members can carry machine-readable specifics such as which fields failed validation.

Two rules keep error responses safe. Client code should branch on `type` and `status`, never on the wording of `title` or `detail`, which may be translated or improved. And errors must not leak internals: stack traces, SQL fragments and internal host names help attackers more than callers. Log the details with a correlation ID and return the ID instead.

```
{
  "type": "https://api.fooddelivery.example/problems/outside-delivery-area",
  "title": "Address is outside the delivery area",
  "status": 422,
  "detail": "Restaurant 7 delivers within 4 km; the address is 6.2 km away.",
  "instance": "/v1/orders/attempts/9b1e",
  "max_distance_km": 4,
  "actual_distance_km": 6.2
}
```

A problem details response (HTTP 422, Content-Type: application/problem+json) for an order outside the delivery area.

Evolving an API without breaking its consumers

An API with consumers is a promise, and Hyrum's law (Section 0.6) warns that someone depends on every observable behaviour. Change is therefore governed by one principle: make backward-compatible changes freely and avoid breaking ones. Consumers help by being **tolerant readers** (a term Martin Fowler uses): they ignore fields they do not understand and read only what they need, so additions never break them. The table below summarises the usual rules; the same logic, applied to stored data and event schemas, appears in schema evolution.

When a breaking change is unavoidable, version. The common strategies are a version in the URL path (`/v2/orders`: explicit, easy to route and cache, but encourages large, infrequent jumps), a custom header or media type parameter (`Accept: application/vnd.fooddelivery.v2+json`: keeps URLs stable but is less visible and harder to test in a browser), and dated versions pinned per client account, which some large API providers use to release small breaking changes continuously while each client upgrades when it chooses. Internal APIs between services owned by the same organisation often avoid versions altogether through the **expand and contract** approach: add the new field or endpoint, migrate every consumer, then remove the old one. Whatever the strategy, announce deprecations in advance; the `Deprecation` (RFC 9745, 2025) and `Sunset` (RFC 8594, 2019) response headers let a server tell clients, in machine-readable form, that an endpoint is deprecated and when it will stop working.

Backward-compatible and breaking changes to an API

Change	Compatible?	Why
Add an optional request field or query parameter	Yes	Existing clients do not send it; the server must apply a sensible default.
Add a field to a response	Usually	Tolerant readers ignore it; strict clients that reject unknown fields will break, so document that clients must tolerate additions.
Add a new endpoint or resource	Yes	Nobody calls it yet.
Add a value to an enumeration in a response	Risky	Clients with exhaustive switch statements may fail on the unknown value; document that new values may appear.
Make an optional request field required	No	Existing clients that omit it start failing with 400.
Remove or rename a field, or change its type or units	No	Clients reading the old field get nothing, or the wrong thing.
Change the meaning of a field or status code	No	The most dangerous break: nothing fails visibly, but behaviour is wrong.
Tighten validation (shorter maximum length, stricter format)	No	Requests that used to succeed now fail.

Describing contracts: OpenAPI and AsyncAPI

A contract written in a machine-readable format can drive documentation, client generation, server stubs, mock servers, linting and tests. OpenAPI is the standard for HTTP APIs. It began as the Swagger specification, which SmartBear donated to the newly formed OpenAPI Initiative under the Linux Foundation in 2015 and which was renamed in 2016; version 3.1 (2021) aligned its schemas with JSON Schema, and version 3.2 followed in September 2025. AsyncAPI applies the same idea to message-driven APIs: it describes channels (queues and topics), the operations that send or receive on them and the message schemas; version 3.0 was released in December 2023.

Teams choose between **design-first** (write and review the contract, then implement it) and **code-first** (generate the contract from annotated code). Design-first makes the API a deliberate decision that consumers can review before anything is built, and suits public and cross-team APIs; code-first keeps contract and implementation in step with less effort, and

suits internal APIs that change quickly. In either case, keep the contract in version control, lint it in the pipeline and run a breaking-change check that compares each proposed version with the published one.

Contract testing and consumer-driven contracts

End-to-end tests that deploy every service together are slow, flaky and get slower as services multiply, yet unit tests cannot detect that a provider has changed a field a consumer relies on. Contract testing fills the gap: each side is tested in isolation against a shared description of their interactions. In consumer-driven contracts, a pattern Ian Robinson described in 2006, each consumer states the parts of the provider's API it actually uses, and the provider runs all its consumers' expectations as part of its own build. The provider then knows exactly which fields nobody reads (safe to change) and which changes would break a named consumer.

Pact, created at realestate.com.au in 2013, is the best-known tool. The consumer's tests run against a Pact mock provider and record the requests made and the responses expected into a contract file; the provider's pipeline replays those requests against the real provider and checks the responses. A broker stores contracts and verification results, so before deploying, a pipeline can ask whether this version of a service is compatible with the versions of its consumers and providers already running in production. Contract tests do not replace a few end-to-end smoke tests, and they check structure rather than business meaning, but they let services be deployed independently with confidence, which is the point of splitting them up.

WATCH OUT

Common mistakes in service communication

- Building long synchronous chains for work the user is not waiting for, so one slow service drags down the whole request.
- Retrying non-idempotent calls automatically, then discovering duplicate orders and double charges.
- Retrying 4xx errors, or retrying without backoff and jitter, which turns a brief outage into a self-inflicted overload.
- Exposing database tables as resources, so every schema change becomes an API change.
- Returning unbounded collections with no pagination, which works in testing and fails in production.
- Choosing GraphQL or gRPC for fashion, then paying for N+1 queries, query-cost limits or browser proxies the system did not need.
- Making breaking changes to an internal API because "we own both sides", while a team elsewhere was quietly calling it.

CHECK YOUR UNDERSTANDING

▼ Why does putting a queue between two services not remove all coupling between them?

It removes temporal coupling, but both services still depend on the message format and meaning, and on the broker itself, so a change to the message can still break the receiver.

▼ Why is DELETE idempotent even though a second DELETE may return 404 instead of 204?

Idempotency concerns the intended effect on the server, not the response: after one or two DELETEs the resource is gone either way.

▼ A mobile client generates a new idempotency key every time it retries. What goes wrong?

The server sees each retry as a new operation, so a lost response followed by a retry can still create a duplicate order; the key must belong to the logical operation, not the HTTP attempt.

▼ **Why can cursor pagination show a stable list when new orders arrive while the user is paging, while offset pagination may not?**

The cursor points just past a specific item, so inserts elsewhere do not shift the next page; an offset counts rows, so new rows push items across page boundaries, causing duplicates or skips.

▼ **Try it: Choose a style for each: the customer app places an order; two busy internal services exchange pricing calls; five services need to react when an order is placed. Would a mobile retry of the first be safe? Give your reasons.**

Placing an order: REST over HTTP, with a stable public contract, but POST is not idempotent, so a retry is safe only with an idempotency key. Internal pricing calls: gRPC suits frequent internal calls, with generated code and compact messages. Reacting services: publish to a topic, so each subscriber gets its own copy and the order service need not know them. A queue would give each message to only one consumer, which is wrong here.

SOURCES

- RFC 9110: HTTP Semantics (IETF, 2022): https://www.rfc-editor.org/rfc/rfc9110.html
- RFC 9457: Problem Details for HTTP APIs (IETF, 2023): https://www.rfc-editor.org/rfc/rfc9457.html
- Martin Fowler: Richardson Maturity Model: https://martinfowler.com/articles/richardsonMaturityModel.html
- gRPC documentation: Core concepts, architecture and lifecycle: https://grpc.io/docs/what-is-grpc/core-concepts/
- Ian Robinson: Consumer-Driven Contracts: A Service Evolution Pattern: https://martinfowler.com/articles/consumerDrivenContracts.html
- Apache Kafka 4.2.0 release announcement: https://kafka.apache.org/blog/2026/02/17/apache-kafka-4.2.0-release-announcement/
- RabbitMQ: Queues: https://www.rabbitmq.com/docs/queues
- RabbitMQ: Streams: https://www.rabbitmq.com/docs/streams
- RabbitMQ: Consumers: https://www.rabbitmq.com/docs/consumers
- Amazon SQS Developer Guide: queue types: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-queue-types.html
- Amazon SNS Developer Guide: fanout to Amazon SQS queues: https://docs.aws.amazon.com/sns/latest/dg/sns-sqs-as-subscriber.html

- Google Cloud Pub/Sub: Order messages: https://docs.cloud.google.com/pubsub/docs/ordering
- NATS: Core NATS: https://docs.nats.io/nats-concepts/core-nats
- NATS: JetStream: https://docs.nats.io/nats-concepts/jetstream
- RFC 9110: HTTP Semantics, section 15.3.3: https://www.rfc-editor.org/rfc/rfc9110.txt
- Microsoft Learn: Asynchronous Request-Reply pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/asynchronous-request-reply
- Stripe: Receive events in your webhook endpoint: https://docs.stripe.com/webhooks
- Standard Webhooks specification: https://raw.githubusercontent.com/standard-webhooks/standard-webhooks/main/spec/standard-webhooks.md

SECTION 6.6 OF 6.8 · ADVANCED · 30 MIN READ · LAST REVIEWED NOVEMBER 2026

Consistency models, CAP and PACELC

BEFORE YOU READ THIS

This section builds on:

- 0.5 Storing data
- 2.5 Availability and reliability
- 6.1 Why distribute

Terms it uses from earlier: ACID, Transaction, Availability, Latency, Invariant

THE IDEA IN BRIEF

Imagine a shop with three branches that each keep a copy of the stock list and phone each other with updates. If the phone line goes down and a customer asks one branch whether the last bicycle is in stock, that branch has two choices: say "I cannot be sure, please wait", or answer from its own list and risk selling a bike another branch has just sold. Even when the phones work, calling the other branches before every answer makes customers wait. Distributed systems face exactly these choices, and the right one depends on how much a wrong answer would cost.

WHAT YOU NEED TO KNOW

- A consistency model is the contract that says which stale or reordered values clients may observe; CAP's C (single-copy, linearisable) differs from ACID's C (valid state) and from isolation levels.
- From strongest to weakest: linearisable (one copy, real-time order), sequential (one agreed order), causal (causes before effects), session guarantees such as read-your-writes and monotonic reads, and eventual (convergence if writes stop).
- The CAP theorem (Brewer's 2000 conjecture, proved by Gilbert and Lynch in 2002) says that during a network partition a replicated system must choose, per request, between consistency and availability; partitions are not optional, so "pick two of three" is a misreading.
- Brewer's 2012 article recommends detecting partitions, operating in an explicit partition mode and recovering afterwards with merges and compensations, choosing differently for different operations.
- PACELC (Abadi, 2010 and 2012) adds that, else, without a partition, systems trade latency against consistency on every request.

- With N replicas, write acknowledgements W and read replies R, choosing R + W > N makes read and write sets overlap, but quorums alone are not linearisable.
- Choose consistency per use case: strong for money, unique resources and invariants; session guarantees for a user's own data; eventual for counters, feeds and positions.

What "consistent" means once data is copied

As soon as a system keeps more than one copy of a piece of data, on replicas in different zones, in caches or in other services, the copies can disagree for a while. A consistency model is the contract that says which disagreements clients may observe. It is a promise about what reads can return given the writes that have happened, and therefore a promise about which surprises application code must handle.

Beware the overloaded word. The C in ACID (Section 0.5) means a transaction moves the database from one valid state to another, respecting rules such as "a balance never goes below zero". The C in CAP means something different and narrower: every read behaves as if there were a single, up-to-date copy. Database isolation levels such as serialisable or read committed are a third family, about concurrent transactions; they are covered in transactions and isolation. This section is about the guarantees of replicated data.

Consistency models from strongest to weakest

Linearisability, defined by Maurice Herlihy and Jeannette Wing in 1990, is the strongest single-object model. Every operation appears to take effect instantaneously at some point between its start and its end, and all clients see the same order, which respects real time: once a write has completed, every read that starts afterwards sees it or something newer. The system behaves as though there were one copy. Food-delivery example: two customers try to claim the last discount code; with a linearisable store, exactly one succeeds and the other sees it gone, whichever replica they reach.

Sequential consistency, defined by Leslie Lamport in 1979 for multiprocessors, keeps the single global order and keeps each client's operations in the order that client issued them, but drops the real-time requirement. A client may read a value that is slightly out of date, as long as everyone agrees on the order of all writes. It is rarely offered by name in distributed databases but explains the gap between "one agreed order" and "one agreed order that matches the clock".

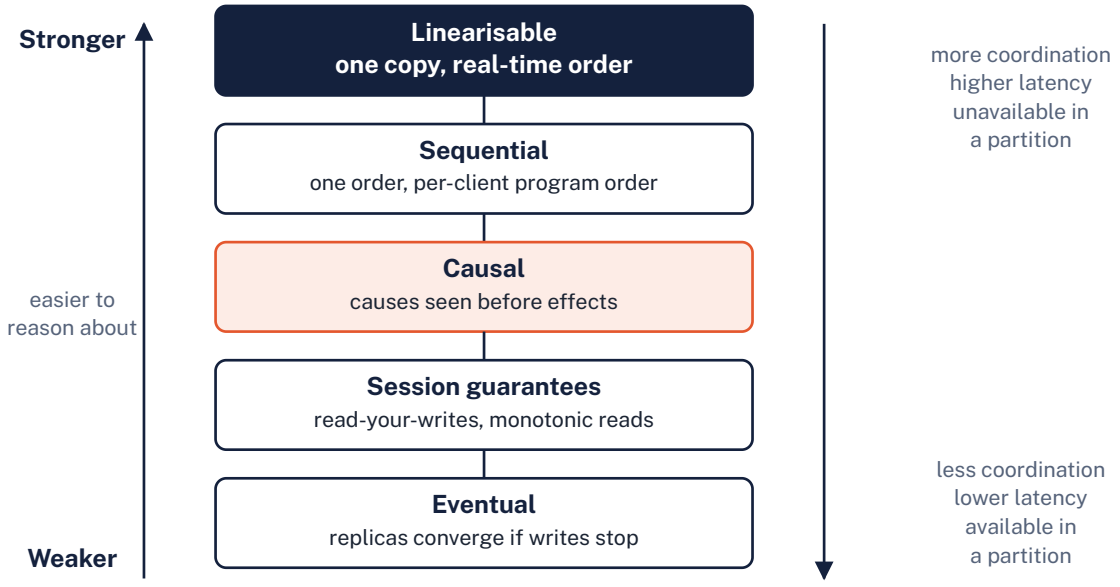
Causal consistency requires only that operations which are causally related are seen in the same order by everyone: if writing B depended on having seen A, nobody may see B without A. Concurrent, unrelated writes may be seen in different orders by different clients. Example: a restaurant replies "Yes, it is gluten-free" to a customer's question in the order chat; nobody

should ever see the answer without the question. Causal consistency is attractive because Jepsen's widely used map of consistency models classifies it as achievable with *sticky* availability: it can keep working during a partition as long as each client stays with the same replica.

The session guarantees, described by Douglas Terry and colleagues in 1994, are weaker, per-client promises that remove the most confusing anomalies. Read-your-writes: after a customer changes the delivery address, that customer's next read shows the new address, even if other users see it later. Monotonic reads: once a client has seen the order status "out for delivery", it never later sees "preparing" because a later read hit a lagging replica. (Two further guarantees, monotonic writes and writes-follow-reads, constrain the order in which a client's writes are applied.)

Eventual consistency, popularised by Werner Vogels in 2008, is the weakest useful guarantee: if no new writes arrive, all replicas eventually converge on the same value. It says nothing about how long that takes or what may be read in the meantime, and a basic implementation that resolves conflicting writes by keeping the one with the latest timestamp, known as last write wins, silently discards the other. Stronger forms, such as strong eventual consistency with conflict-free replicated data types (CRDTs), guarantee that replicas which have received the same updates hold the same state, by using data types whose concurrent updates always merge, such as counters and sets.

The consistency ladder



Each step down removes a guarantee and some coordination; the session guarantees are per-client promises that sit between causal and plain eventual consistency.

IF YOU ARE NEW TO THIS

If you are new to this

Think of a family shopping list kept on three phones that sync whenever they have a signal. Linearisable means the list behaves as if there were one sheet of paper on the fridge. Eventual means the phones agree once everyone has had a signal for a while, but until then one person may buy milk that someone else has already bought. Everything in between is a different compromise between "always agree" and "always work, even in a tunnel".

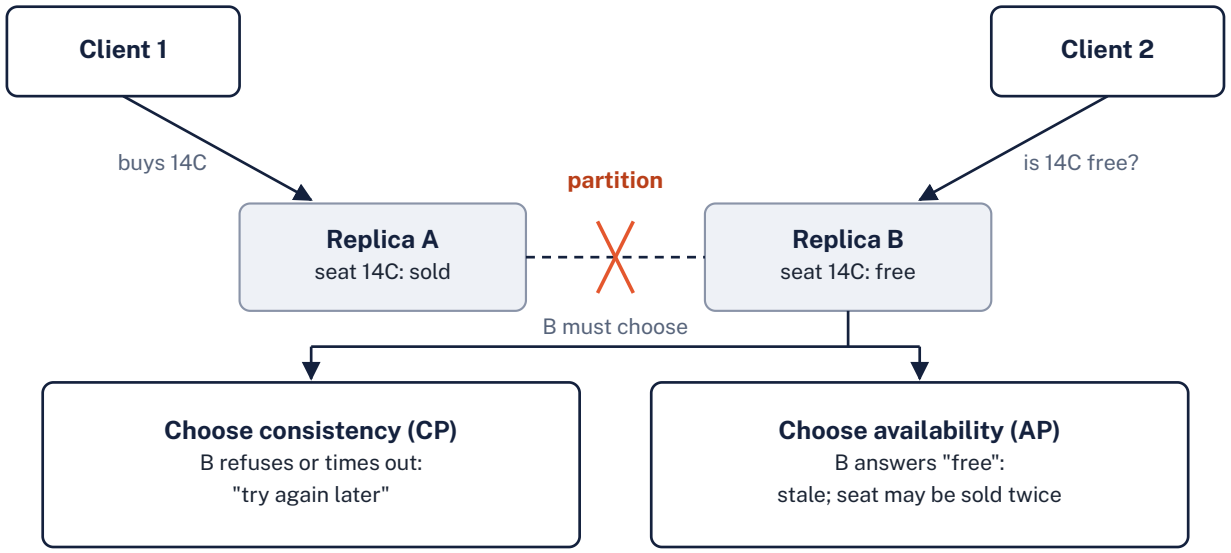
CAP: what it actually says

Eric Brewer presented the CAP idea as a conjecture in his keynote at the ACM Symposium on Principles of Distributed Computing (PODC) in 2000. In 2002 Seth Gilbert and Nancy Lynch published a formal proof, making it a theorem for their precise definitions. **Consistency** meant linearisability. **Availability** meant that every request received by a node that has not failed must result in a response. **Partition tolerance** meant the system keeps its promises even when the network loses arbitrarily many messages between nodes.

Read carefully, the CAP theorem is narrow. It says that when a network partition occurs, so that some nodes cannot reach others, a replicated system must choose for each request between consistency and availability. A node cut off from the latest writes can either refuse to answer (or wait until the partition heals), preserving consistency, or answer with what it has, preserving availability at the risk of returning stale data or accepting conflicting writes. The diagram shows the dilemma for a ticketing system selling seat 14C.

Several popular readings are wrong. **"Pick any two of three"** suggests partition tolerance is optional, but in any system that communicates over a network, partitions happen whether or not you choose them; the only real choice is how to behave when they do. A so-called "CA" system is one that has not decided what to do when a partition occurs; that is defensible only where partitions are practically impossible, such as on a single node. **"Our database is CP (or AP)"** is usually imprecise: CAP availability is a very strong property that many systems marketed as highly available do not have, and many systems called CP are not linearisable either. Martin Kleppmann argued in 2015 that databases should not be labelled CP or AP at all, because most fit neither definition and their behaviour depends on configuration. **"CAP explains our latency"** is also wrong: CAP says nothing about what happens when there is no partition, which is most of the time.

The CAP choice, made concrete



Replica B cannot learn that seat 14C has just been sold on replica A, so it must either refuse to answer or risk selling the seat twice.

CAP twelve years later

In 2012, in an IEEE Computer article titled "CAP Twelve Years Later: How the Rules Have Changed", Brewer himself explained that the two-of-three framing had misled people. Partitions are rare, so a system can offer both consistency and availability almost all of the time; the choice applies only during a partition, and only to the operations affected. The choice need not be global either: different operations and different data can make different choices.

Brewer recommended designing explicitly for partitions in three steps: **detect** the partition, enter an explicit **partition mode** in which some operations are limited (for example, accept orders but postpone refunds), and run a **recovery** process afterwards that merges state and compensates for mistakes made during the partition. He also noted that CAP's C is single-copy consistency, a stricter subset of what ACID's C protects, and that invariants such as "never oversell" may have to be repaired after the fact. This framing leads directly to the business-level compensations of sagas: an airline that oversells a flight compensates the passenger rather than refusing all bookings during a network fault.

Vector clocks and hybrid logical clocks

Merging after a partition needs to know which writes really conflict, and wall clocks cannot say (Section 6.1). A **vector clock** attaches a list of node and counter pairs to each version. If every counter in one is less than or equal to those in another, the first version is an ancestor and can go; otherwise the two are concurrent and must be merged. Amazon's 2007 Dynamo paper used

this for shopping carts: no addition is lost, though deleted items can resurface. The lists grow, so Dynamo trimmed them. A **hybrid logical clock** (Kulkarni and others, 2014) is one compact timestamp: time close to the physical clock plus a counter, so that an event that led to another always has the smaller value. It orders events but, unlike a vector clock, a smaller value does not prove a causal link. CockroachDB uses hybrid logical clocks.

PACELC: the trade-off you pay every day

Daniel Abadi pointed out in a 2010 blog post, and in a 2012 IEEE Computer article, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story", that CAP ignores the trade-off designers actually face most of the time. PACELC reads: if there is a **P**artition, choose between **A**vailability and **C**onsistency; **E**lse, in normal operation, choose between **L**atency and **C**onsistency.

The "else" half is the important addition. To make a read consistent, a replicated system must coordinate: send the write to other replicas and wait for acknowledgements, or route every read to a leader, or contact several replicas and compare. Every one of those costs network round trips, and across regions a round trip can cost tens or even hundreds of milliseconds. A system that answers from the nearest replica without coordinating is faster but may be stale. That trade-off is paid on every request, partition or not.

PACELC classifications are often quoted, such as PA/EL for systems tuned for speed and availability and PC/EC for systems that always prefer consistency. Treat them as descriptions of a configuration, not of a product: the same database can be PA/EL in one deployment and PC/EC in another, depending on settings chosen per table, per query or per request.

Consensus and coordination: leaders, leases and locks

Failover, shard maps and scheduled jobs all need several machines to agree on one answer, such as which node now leads, even while some have crashed or messages are late. That is the **consensus** problem, and a decision once reached is final. Leslie Lamport's Paxos is the classic algorithm. Raft (Ongaro and Ousterhout, 2014) offers the same fault tolerance and performance but was designed to be easier to understand. A Raft leader sends heartbeats; a follower that hears none before its randomised timeout starts an election in a new numbered *term*, and a majority of votes makes it leader. Five servers keep working with two failed. With more failures the cluster stops making progress, but it never returns a wrong result. Clusters are usually odd-sized, because etcd's documentation notes that four members tolerate no more failures than three, and it advises no more than seven, since every write must reach more machines.

Few teams implement this themselves. They use **etcd**, which uses Raft and is the backing store for all Kubernetes cluster data, or **Apache ZooKeeper**, whose election recipe makes the candidate with the lowest-numbered temporary node the leader. Both hold small amounts of coordination data, not business records.

A **lease** is a claim with a time limit: the holder must keep renewing it, and if it dies the lease expires and another node takes over. Kubernetes uses leases for node heartbeats and leader election. The danger is a holder that pauses, perhaps for garbage collection, then wakes unaware that its lease has expired and writes anyway; checking first does not help, because the pause can fall between check and write. A **fencing token** solves this: a number that rises with each grant, sent with every write, so that storage rejects any token lower than one it has already seen.

Distributed locks raise the same issue. Martin Kleppmann separates locks for efficiency, where failure means duplicated work such as a repeated email, from locks for correctness, where failure corrupts data. Redis's Redlock algorithm takes a lock on a majority of independent Redis masters. Kleppmann argued in 2016 that it depends on timing assumptions and issues no fencing tokens, so it should not protect correctness. Salvatore Sanfilippo, Redis's creator, replied that it is safe under its stated model, while accepting the point about monotonic clocks. The Redis documentation links to both; the debate is unresolved, so read them. Consensus stores need the same care: Jepsen's test of etcd 3.4.3 found that two clients could hold one lock, and advised fencing. The practical lesson is that the protected resource must defend itself, by a fencing check or a database constraint. If Dispatch runs one assignment scheduler, an occasional double run must do no harm.

Quorums: tuning consistency with arithmetic

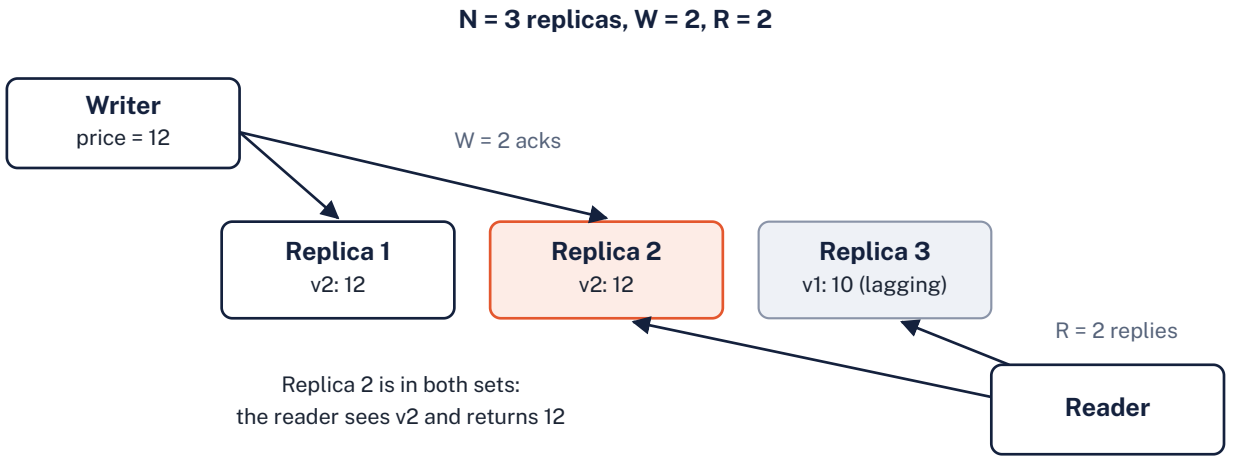
Leaderless replicated stores, in the style of Amazon's Dynamo paper (2007), let each operation choose how many replicas to involve. With *N* replicas of an item, a write is acknowledged when *W* replicas have stored it, and a read collects answers from *R* replicas and returns the newest version among them. If R + W > N, every read set overlaps every write set in at least one replica, so a read will include at least one copy of the latest acknowledged write.

The choice of *R* and *W* trades consistency, latency and availability. Lower values mean waiting for fewer, faster replicas and tolerating more failures, at the price of possible stale reads. Higher values mean the opposite. Writes are usually sent to all *N* replicas anyway; *W* only decides how many acknowledgements the client waits for.

Quorum overlap is necessary for fresh reads, but it is not, on its own, linearisability. Concurrent writes may be ordered by timestamps from clocks that disagree, so a later write can lose under last-write-wins. A write that failed after reaching some replicas may still be visible on them.

"Sloppy" quorums, which accept writes on substitute nodes during failures, break the overlap. And two reads that overlap a write still in progress can see new then old values unless the reader repairs lagging replicas before returning. Stores that need true compare-and-set semantics add a consensus protocol, such as the Paxos-based lightweight transactions in Apache Cassandra, for those operations only.

Why R + W > N works



The write was acknowledged by replicas 1 and 2; the read asks replicas 2 and 3; the overlap at replica 2 means the newest version is in the read set.

WORKED EXAMPLE

Worked example: choosing N, W and R for a menu-price store

The food-delivery platform stores menu prices in a leaderless store with N = 3 replicas, one per availability zone. Each replica answers in about 2 ms within its zone and about 5 ms from another zone. The quorum client waits for the W-th (or R-th) fastest reply.

W = 2, R = 2 (R + W = 4 > 3)	Overlap guaranteed. Reads and writes each wait for the second-fastest replica (about 5 ms). Both still succeed with one zone down. A common balanced default.
W = 3, R = 1 (4 > 3)	Overlap guaranteed and reads are fast (nearest replica, about 2 ms), but every write fails if any one replica is down: write availability is the product of three availabilities. Suits data written rarely and read constantly, if write failures can be retried later.
W = 1, R = 3 (4 > 3)	Writes are fast and survive two failures; every read must reach all three replicas, so reads fail if any zone is unreachable. Rarely a good trade.
W = 1, R = 1 (2, not > 3)	Fastest and most available, but a read can miss the latest write entirely. Acceptable for a popularity counter; not for a price the customer will be charged.
N = 5, W = 3, R = 3 (6 > 5)	Overlap guaranteed while tolerating two failed replicas for both reads and writes, at the cost of more storage and slightly slower quorums.

General rule: with R + W > N, writes tolerate N − W failures and reads tolerate N − R failures.

Overlap gives fresh reads only for acknowledged writes and only with reliable versioning; see the caveats above before relying on quorums for invariants such as stock levels.

For prices, the platform chose W = 2, R = 2, and charged customers using the price stored in the order at checkout, so a stale menu read could never change what was charged.

How well-known databases can be configured

Real products rarely sit at a single point on the ladder. They expose settings, and the right description is "how this deployment is configured". Four common examples follow; check the current documentation of the version you run, because defaults change between releases.

Configurable consistency in four widely used databases (behaviour as documented at the time of writing)

Database	Typical default behaviour	Settings that strengthen consistency	What to watch
PostgreSQL with streaming replication	One primary takes all writes; replication to standbys is asynchronous by default, so reads from a standby may be stale.	`synchronous_standby_names` with `synchronous_commit` (for example `remote_apply`) makes a commit wait until standbys have it; read from the primary for read-your-writes.	Synchronous replication adds commit latency, and writes stall if the required standbys are unreachable.
MongoDB replica set	Writes default to majority acknowledgement in most deployments; reads from the primary default to read concern `local`, which can return data later rolled back.	Read concern `majority` or `linearizable`; causally consistent sessions when both reads and writes use majority.	Reading from secondaries (read preference) trades freshness for latency and load.
Apache Cassandra	Consistency level is chosen per request (ONE, QUORUM, LOCAL_QUORUM, ALL and others); conflicts resolve by last-write-wins timestamps.	QUORUM (or LOCAL_QUORUM within a data centre) for both reads and writes so that R + W > replication factor; lightweight transactions for compare-and-set.	Clock skew affects last-write-wins; LOCAL_QUORUM gives no cross-data-centre guarantee.
Amazon DynamoDB	Reads are eventually consistent by default; global tables replicate between regions asynchronously by default.	`ConsistentRead=true` on tables and local secondary indexes; global tables with multi-Region strong consistency (generally available since 2025).	Strongly consistent reads are not available on global secondary indexes, consume twice the read capacity and can be slower.

Choosing consistency per use case

The most useful habit is to stop asking "is our system consistent?" and ask, for each piece of data and each operation, *what goes wrong if a client sees a stale or reordered value, and how much does it cost?* Money moving, unique resources and invariants usually justify strong consistency and its latency. Counters, feeds, recommendations and analytics usually do not.

Many user-facing anomalies disappear with just the session guarantees, which are cheap: route a user's reads to the primary for a few seconds after that user writes, or carry a version token in the session and wait until the replica has caught up with it.

Consistency needs by use case

Use case	Cost of a stale or reordered read	Sensible guarantee	Typical mechanism
Bank balance before a withdrawal	Overdraft beyond the limit; money lost	Linearisable (and serialisable transactions)	Single leader for the account, synchronous replication, or consensus
Last seat or ticket at an on-sale	Selling one seat twice; refunds and angry customers	Linearisable per seat	Conditional write (compare-and-set) on the seat record
Unique username at sign-up	Two accounts with the same name	Linearisable on the uniqueness check	Unique constraint in one database, or a consensus-backed registry
Customer's own delivery address after editing	Food sent to the old address	Read-your-writes	Read from the leader after a write, or a session version token
Order status shown to the customer	Status appears to go backwards; confusion and support calls	Monotonic reads plus read-your-writes	Sticky routing to one replica, or version tokens
Restaurant chat with the customer	An answer appears before its question	Causal	Causally consistent store, or per-conversation ordering
Shopping basket	A removed item reappears; mildly annoying	Eventual, with merge	Always accept adds; merge versions (the Dynamo paper's example)
Likes count or restaurant rating	Count is off by a few for a few seconds	Eventual	Asynchronous replication, counters or CRDTs
Courier location on the map	Dot a few seconds behind	Eventual, latest value wins	Low-latency writes to the nearest replica; old positions discarded

WATCH OUT

Common mistakes

- Saying "we chose AP" for a whole system, instead of deciding per operation and per piece of data.
- Assuming a replicated database is linearisable because its marketing says "strongly consistent", without reading what the guarantee covers (single keys? single region? indexes?).
- Adding read replicas or caches to a system and forgetting that every user-facing read path just became eventually consistent.
- Relying on last-write-wins with clocks that are not synchronised, and silently losing updates.
- Using quorums with R + W > N and believing that makes stock reservations safe without a compare-and-set.
- Treating eventual consistency as a technical detail, when it is visible to users and needs product decisions about what they see while data converges.

CHECK YOUR UNDERSTANDING

▼ What is the difference between linearisability and sequential consistency?

Both require a single order of operations that all clients agree on and that respects each client's own order; linearisability also requires that order to respect real time, so a read that starts after a write completes must see it.

▼ Why is a "CA" distributed database not a meaningful choice?

Because network partitions happen in any system that communicates over a network; a system that assumes they never occur has simply not decided what it will do when they do.

▼ With N = 5 and W = 2, what is the smallest R that guarantees read and write sets overlap?

R = 4, because R + W must exceed 5.

▼ A user edits their profile and the page reloads showing the old version, although other users seeing it late does not matter. Which guarantee is missing, and what is a cheap fix?

Read-your-writes; route that user's reads to the leader for a short time after a write, or carry a version token and wait for a replica that has caught up.

▼ **Try it: A seat-price store uses N = 5, W = 2, R = 3, and a colleague says reads always overlap the latest write. Check this and propose a fix. Would you use quorums alone to stop two fans buying one seat?**

R + W = 5, not greater than N, so a read can miss the latest write. Set W = 3 and R = 3, giving 6 > 5; both still work with two replicas down, at a slightly higher latency. Choose by use: stale prices may be tolerable briefly. Quorums alone are not linearisable, so they will not protect a unique seat. Selling one seat needs strong consistency, such as a single-leader transaction or uniqueness constraint.

SOURCES

- Eric Brewer: CAP Twelve Years Later: How the Rules Have Changed (InfoQ reprint of the IEEE Computer article): https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/
- Daniel Abadi: Problems with CAP, and Yahoo's little known NoSQL system (2010, origin of PACELC): https://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html
- Jepsen: Consistency models: https://jepsen.io/consistency/models
- Martin Kleppmann: Please stop calling databases CP or AP (2015): https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html
- Apache Cassandra documentation: Dynamo (tunable consistency and quorums): https://cassandra.apache.org/doc/latest/cassandra/architecture/dynamo.html
- Raft consensus algorithm site: https://raft.github.io/
- Ongaro and Ousterhout: In Search of an Understandable Consensus Algorithm (2014): https://raft.github.io/raft.pdf
- etcd FAQ: https://etcd.io/docs/v3.5/faq/
- Kubernetes: Operating etcd clusters: https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/
- Kubernetes: Leases: https://kubernetes.io/docs/concepts/architecture/leases/
- Apache ZooKeeper: Recipes and solutions: https://zookeeper.apache.org/doc/current/recipes.html
- Martin Kleppmann: How to do distributed locking (2016): https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html
- Salvatore Sanfilippo: Is Redlock safe?: https://antirez.com/news/101
- Redis: Distributed locks: https://redis.io/docs/latest/develop/clients/patterns/distributed-locks/
- Jepsen: etcd 3.4.3: https://jepsen.io/analyses/etcd-3.4.3
- DeCandia and others: Dynamo: Amazon's Highly Available Key-value Store (2007): https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

- Kulkarni and others: Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases: https://cse.buffalo.edu/tech-reports/2014-04.pdf
- CockroachDB: Transaction layer: https://docs.cockroachlabs.com/docs/stable/architecture/transaction-layer

SECTION 6.7 OF 6.8 · ADVANCED · 30 MIN READ · LAST REVIEWED NOVEMBER 2026

Distributed transactions, sagas and the outbox pattern

BEFORE YOU READ THIS

This section builds on:

- 0.5 Storing data
- 6.3 Microservices
- 6.4 Event-driven

Terms it uses from earlier: ACID, Database per service, Choreography, Orchestration, At-least-once delivery

THE IDEA IN BRIEF

Booking a holiday means booking a flight, a hotel and a hire car from three different companies. Nobody can make all three happen at exactly the same instant. Instead you book them one at a time, and if the hire car turns out to be unavailable, you cancel the hotel and the flight, perhaps paying a cancellation fee. That is a saga: a series of separate steps, each with a planned way to undo it. Software that spans several services works the same way, and the skill is in choosing the order of the steps and planning the cancellations before anything goes wrong.

WHAT YOU NEED TO KNOW

- Two-phase commit gives atomicity across participants through prepare and commit phases, but it blocks while the coordinator is down and holds locks across the network; XA (X/Open, 1991) standardises it.
- Across independently owned services, 2PC multiplies unavailability, holds locks remotely, couples deployments and is unsupported by most APIs and many brokers, so it is generally avoided.
- A saga (Garcia-Molina and Salem, 1987) is a sequence of local transactions, each with a compensating transaction; either all steps complete or the completed ones are compensated in reverse order.
- Compensations are new forward transactions, not rollbacks: intermediate states may have been seen, and compensations must be idempotent, retriable and able to arrive out of order.
- Sagas lack isolation; countermeasures described by Chris Richardson include semantic locks, commutative updates, pessimistic view, reread value, version files and choosing by value.

- The transactional outbox writes the business change and its message in one local transaction, and a relay (polling or change data capture such as Debezium) publishes it at least once; idempotent receivers with an inbox remove duplicates.
- Prefer one local transaction where possible, redesign boundaries when the split is accidental, and use sagas when steps can be compensated or retried.

The problem: one business action, several databases

Inside one database, a transaction makes several changes atomic: the order row, its items and the payment reference are all saved or none are (Section 0.5). Once data is split across services that each own their database, as microservices and database per service recommend, a single business action such as placing a food order touches several databases owned by different services: the order service's, the payment service's and the dispatch service's. No single local transaction can cover them all.

There are two families of answer. One is to make many resource managers act as one through a distributed atomic commit protocol, of which two-phase commit is the classic. The other is to accept that the action is a sequence of local transactions and design for the moments in between: the saga. A third answer, often the best, is to notice that the transaction boundary and the service boundary disagree and to move one of them. This section covers all three and ends with a guide for choosing.

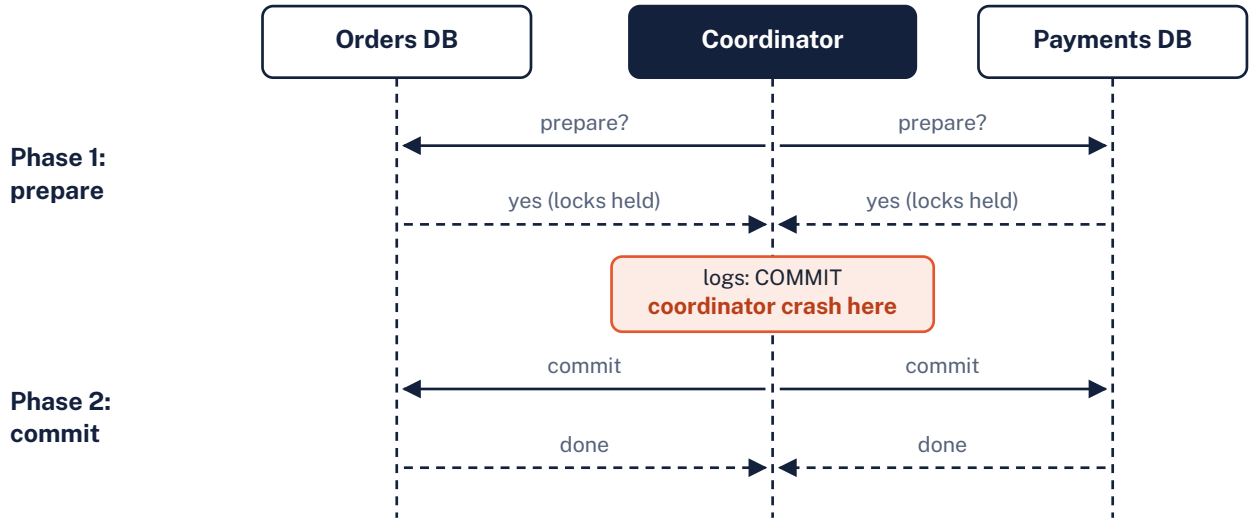
Two-phase commit and XA

Two-phase commit (2PC) is a protocol, dating from the late 1970s, that gives all-or-nothing atomicity across several participants such as two databases or a database and a message broker. A **coordinator** (the transaction manager) runs it. In **phase 1, prepare**, the coordinator asks every participant whether it can commit. A participant that says yes must first make the changes durable and keep the relevant locks, because by voting yes it gives up the right to abort on its own. If every vote is yes, the coordinator writes its decision, COMMIT, to its own log; if any vote is no or a participant does not answer, the decision is ABORT. In **phase 2, commit**, it tells every participant the outcome, and they apply it and release their locks.

The weakness is that 2PC is a **blocking** protocol. A participant that has voted yes is *in doubt*: it does not know the outcome and cannot safely guess, because another participant may already have committed. If the coordinator crashes at that moment and stays down, the in-doubt participants keep their locks until it recovers, and other transactions touching those rows wait too. Operators can force a *heuristic* decision to release the locks, at the risk of breaking atomicity.

The XA specification, published by X/Open in 1991, standardised the interface between a transaction manager and resource managers so that, for example, a Java application server can commit a database update and a message send atomically. Many relational databases and some message brokers support it.

Two-phase commit, and where it blocks



If the coordinator fails after the votes, participants that voted yes cannot decide alone and keep their locks

Participants that have voted yes hold their locks until they hear the decision; if the coordinator fails at the highlighted point, they are stuck.

Why 2PC is avoided across microservices

2PC works and is still used inside a single system's boundary. Across independently owned services it is usually a poor fit, for reasons that compound each other.

Availability multiplies down. Every participant and the coordinator must be up for any transaction to commit, so the transaction's availability is the product of theirs, the same arithmetic as a synchronous call chain (see why distribute). **Locks are held across the network** for at least two round trips, and much longer if anything is slow, which cuts throughput and raises contention. **Coupling returns**: all participants must speak the same protocol, share a transaction manager and agree on timeouts, which undermines the independent deployability that justified splitting the services. **Many participants cannot join**: most HTTP APIs, third-party services such as payment providers, many NoSQL databases and several popular brokers do not support XA. And the payment provider's API will never hold a lock for your coordinator.

The picture is different inside distributed databases. Systems such as Google Spanner run two-phase commit internally between groups of replicas that are themselves kept consistent by consensus, so a crashed coordinator is replaced rather than awaited. That makes cross-

partition transactions within one database practical; it does not make transactions across separately owned services and their different technologies practical.

Sagas: a sequence of local transactions

Hector Garcia-Molina and Kenneth Salem introduced the saga in a 1987 SIGMOD paper about *long-lived transactions*, ones that would hold database locks for hours. Their idea was to split such a transaction into a sequence of ordinary transactions $T_1, T_2 \dots T_n$, each of which commits on its own, and to pair each with a compensating transaction C_i that semantically undoes it. The system then guarantees that either the whole sequence completes, or the completed steps are followed by their compensations in reverse order: $T_1 \dots T_j, C_j \dots C_1$.

Microservice architectures adopted the idea because it fits their constraints exactly: each step is a local ACID transaction in one service's own database, and steps are linked by messages. A saga gives atomicity in the eventual, business sense (all steps happen, or the effects are undone), but not isolation: between steps, other transactions can see the intermediate state.

It helps to classify the steps, as Chris Richardson does in *Microservices Patterns* (2018). **Compensatable transactions** can be undone by a compensation. The **pivot transaction** is the go/no-go point: once it commits, the saga will run to completion. **Retriable transactions** come after the pivot and must be designed so that they eventually succeed if retried. Placing steps that are likely to fail before the pivot, and steps that cannot be undone after it, is a key design decision.

Choreography or orchestration

A saga's steps must be coordinated somehow, and the two options mirror the choreography and orchestration discussed in event-driven architecture.

In a **choreography-based saga** there is no coordinator. Each service listens for events and reacts by doing its local transaction and publishing a new event: the order service publishes `OrderCreated`, the payment service reacts and publishes `PaymentAuthorised`, the dispatch service reacts to that, and so on; failure events trigger compensations in the services that listen for them. It is simple for two or three steps and keeps services loosely coupled. As steps grow, the flow exists only implicitly, scattered across services, so it is hard to see, test and change; and cyclic event dependencies between services creep in.

In an orchestration-based saga a single orchestrator, usually in the service that owns the business process, sends commands to participants and reacts to their replies. The orchestrator is a persistent state machine: it records which step it is on in its own database, so it can resume after a crash. The flow is explicit and testable in one place, and participants need not know

about each other. The risk is that the orchestrator accumulates business logic that belongs in the participants. Workflow engines such as Temporal, Camunda and AWS Step Functions provide durable orchestration as a platform, at the cost of another dependency. For more than about three steps, or for any flow with branching and timeouts, most practitioners prefer orchestration.

Durable execution: Temporal and Step Functions

Durable execution means that a workflow, once started, runs to completion despite crashes. In Temporal, a workflow is ordinary code; what happens is stored in an event history, and after a crash another worker replays the code against that history to rebuild its state and carry on. Workflow code must therefore be deterministic, and outside work goes into *activities*, which can be retried and so should tolerate repetition. AWS Step Functions instead defines a state machine in Amazon States Language; Standard workflows can run for up to a year and Express ones for five minutes. An order saga's wait for a restaurant to accept can then be a plain timer. The costs are a new critical dependency and care when changing the code of workflows still running. Check the current limits.

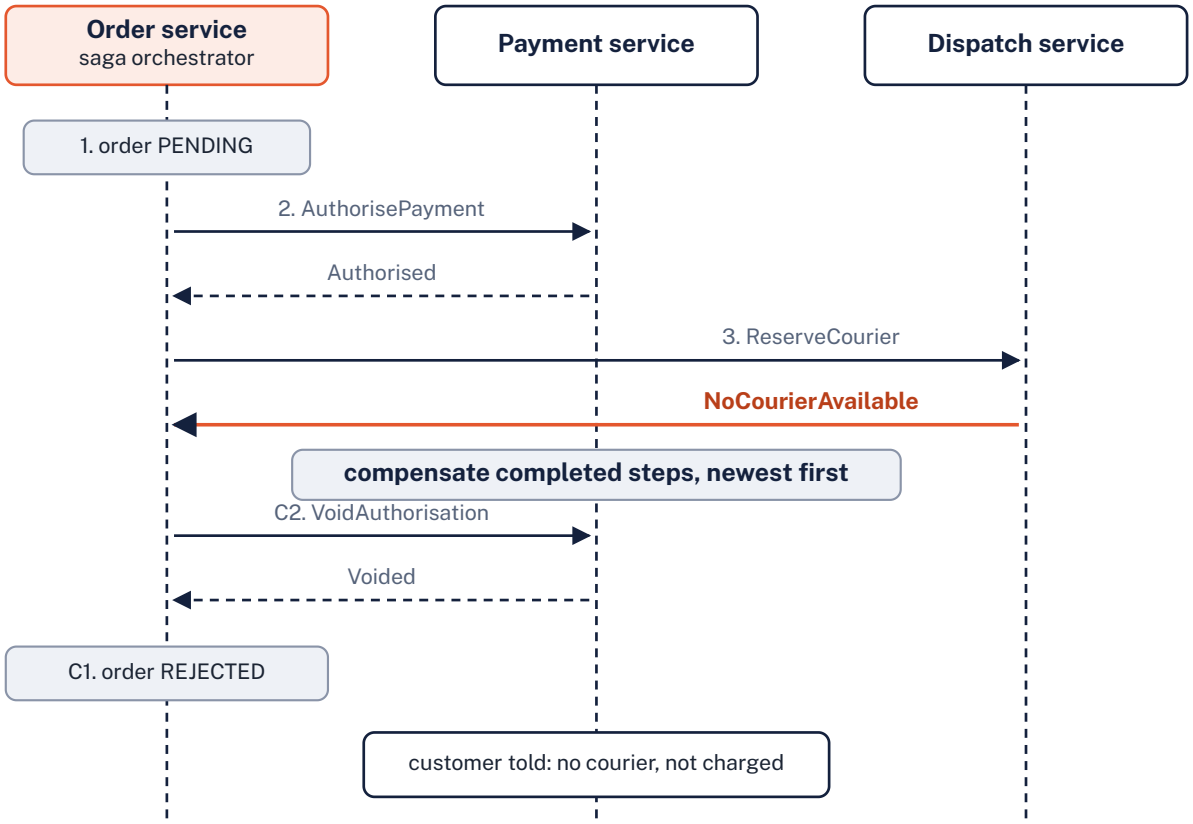
Worked example: the order saga

The food-delivery platform's order saga is orchestrated by the order service. The order of steps is deliberate: the payment authorisation is a cheap hold that is easy to void, the courier reservation is the step most likely to fail at busy times, and confirming to the restaurant is the pivot, because once a kitchen has started cooking the order cannot simply be undone.

The order saga, step by step

Step	Service and local transaction	Kind	Compensation if a later step fails
1. Create order	Order service inserts the order with state `PENDING` (a semantic lock).	Compensatable	C1: set state to `REJECTED` and record the reason. The row is kept for audit, not deleted.
2. Authorise payment	Payment service asks the card provider to place a hold for the total, using the order ID as idempotency key.	Compensatable	C2: void the authorisation, releasing the hold. The customer may briefly see a pending amount on their banking app.
3. Reserve courier	Dispatch service reserves a nearby courier for the pickup window.	Compensatable	C3: release the courier so they can take other jobs.
4. Confirm order	Order service sets state to `CONFIRMED` and sends the ticket to the restaurant.	Pivot	None: from here the saga must go forward. Problems later are handled by new business processes (refunds, vouchers).
5. Notify customer; capture payment on pickup	Notification service sends "Order confirmed"; payment service captures the held amount when the courier collects the food.	Retriable	None: these steps are retried, idempotently, until they succeed or are escalated to a person.

The order saga when no courier is available



Step 3 fails, so the orchestrator runs the compensations for steps 2 and 1, newest first, and the customer is told before anything was cooked or charged.

Compensation is not rollback

A database rollback makes it as if the transaction never happened: nobody else ever saw its changes. A compensation is a new, forward transaction that semantically counteracts an earlier one that *did* happen and may have been seen. In the order saga, the customer's bank may already show a pending hold, and an email may have said "payment authorised". The compensation cannot unsend the email; it can only void the hold and send another message. Some actions cannot be compensated at all, such as food already cooked or a notification already read; those belong after the pivot, or need a business compensation such as a voucher or a refund.

Compensations therefore need the same care as forward steps. They must be **idempotent**, because the message that triggers them may be delivered twice. They must be **retriable until they succeed**, because a compensation that fails permanently leaves the system in a state nobody designed; if they truly cannot succeed, they must raise an alert for a person to resolve. They may need to be **commutative with the forward step**: if a `VoidAuthorisation` arrives before the `AuthorisePayment` it undoes (possible after timeouts and retries), the payment

service should remember the void and refuse the late authorisation rather than fail. And they should be visible to the business: "rejected, no courier" is information the operations team wants on a dashboard.

Living without isolation: anomalies and countermeasures

Because each step commits on its own, sagas lack the I in ACID, and concurrent sagas can interfere. Richardson describes three resulting anomalies. A **lost update** occurs when one saga overwrites a change made by another without reading it, such as a cancellation saga marking an order cancelled and the order saga then setting it confirmed. A **dirty read** occurs when a saga reads data another saga has changed but may yet compensate, such as a credit-limit check that counts a hold that is about to be voided. A **fuzzy or non-repeatable read** occurs when two steps of one saga read the same data and get different values because another saga changed it in between.

He also catalogues countermeasures, design techniques that reduce the anomalies, drawing on earlier research on transactions across multiple databases. The most widely used are these.

Countermeasures for the lack of isolation in sagas (after Richardson)

Countermeasure	Idea	Food-delivery example
Semantic lock	A compensatable step sets an application-level flag, usually a `*_PENDING` state, telling other transactions that the record is in flux; they wait, fail or treat it specially.	While an order is `PENDING`, a cancellation request is queued or refused with "order is being processed", not applied halfway through.
Commutative updates	Design updates so that they give the same result in any order, removing lost updates.	The courier's earnings ledger records credits and debits as separate entries rather than overwriting a balance.
Pessimistic view	Reorder steps to minimise the business risk of a dirty read, putting risky updates into retriable steps after the pivot.	Increase a restaurant's "available capacity" only in a retriable step after cancellation is certain, never in a step that might be compensated.
Reread value	Before updating, re-read the record and check it has not changed since it was read (optimistic locking); if it has, abort and possibly restart the saga.	Before confirming, the orchestrator checks the order's version is still the one it created; if a cancellation changed it, it compensates instead.
Version file	Record the operations applied to a record so that operations arriving out of order can be reordered or recognised.	The payment service records that a void arrived for an authorisation it has not yet seen, and rejects the late authorisation.
By value	Choose the coordination mechanism per request by its business risk.	Ordinary orders use the saga; a high-value corporate catering order might go through a stricter, slower process.

IF YOU ARE NEW TO THIS

Step by step: the moment between saving and sending

The idea that needs the most care here is the dual-write problem and how the transactional outbox solves it. Each saga step saves a change in its own database and sends a message so that the next step can start. The trouble lies in the moment between the two.

- The order service saves order 1042 in its database with the state PENDING. Next it should send an OrderCreated message to the broker, so that the payment service can begin.
- Suppose the server crashes after saving and before sending. The order exists, but the message never went, so the payment service never hears of it and order 1042 waits for ever.
- Doing it the other way round is no better: if the service sends first and the save then fails, the payment service starts work on an order that does not exist.
- Writing to two places in separate steps like this is called a dual write: there is always a moment when one has happened and the other has not.
- The transactional outbox removes that moment. The service saves the order and the outgoing message, as a row in an outbox table, in one database transaction, so both are saved or neither is. A separate program, the relay, then reads the outbox and sends each message to the broker.
- If the relay crashes after sending a message but before noting that it did, it sends that message again, so every receiver must cope with duplicates, as the end of this section explains.
- **The lesson:** without two-phase commit, which this section advises against, a database and a broker cannot be updated as one. Instead, save the message with the change, send it afterwards, and accept the occasional duplicate.

Publishing reliably: the transactional outbox

Every saga step does two things: it updates its own database and it sends a message (an event or a reply). Doing them separately creates the dual-write problem. If the service commits and then crashes before publishing, the rest of the saga never hears about the step. If it publishes first and the commit then fails, other services act on something that never happened. Wrapping the database and the broker in 2PC is exactly what we wanted to avoid.

The transactional outbox solves it with one local transaction. The service writes the business change and the outgoing message, as a row in an `outbox` table, in the same database transaction; either both commit or neither does. A separate **relay** then reads the outbox and publishes each message to the broker. Two relay designs are common. A **polling publisher** queries the outbox table for unpublished rows every so often and marks them sent: simple, but it adds load and latency. **Transaction log tailing** uses change data capture (CDC): a tool reads the database's own replication log (the PostgreSQL write-ahead log or the MySQL binary log) and turns new outbox rows into messages. Debezium, an open-source CDC platform usually run with Kafka Connect, is the best-known example and includes an outbox event router that routes each row to a topic based on a column such as the aggregate type.

The relay can crash after publishing and before recording that it did, so it will sometimes publish a message twice. The outbox therefore gives at-least-once delivery, and consumers must cope with duplicates. Publishing order is preserved per aggregate if the relay publishes rows in commit order and uses the aggregate ID as the message key.

```
BEGIN;

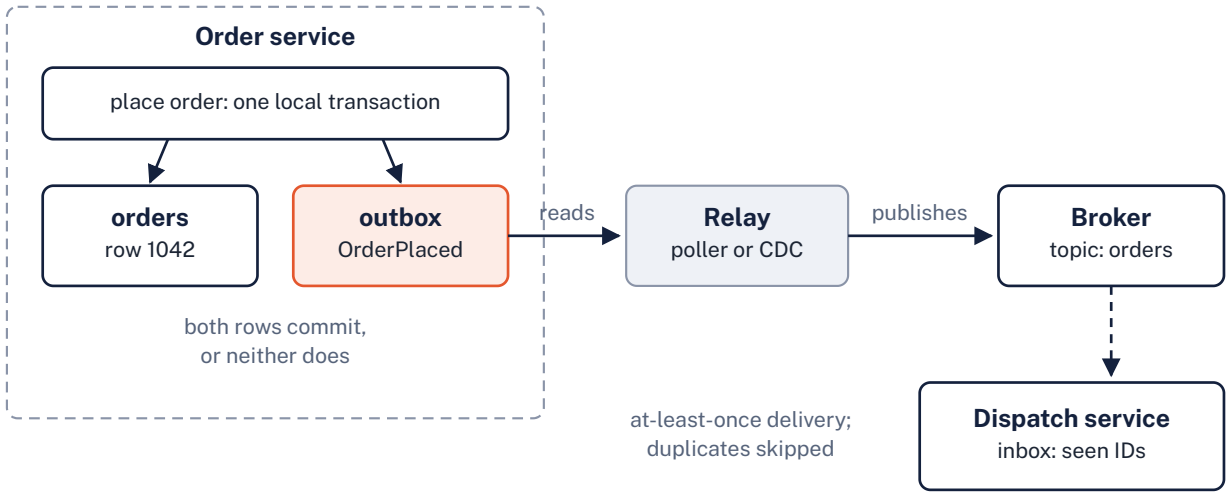
INSERT INTO orders (id, customer_id, state, total_pence)
VALUES ('1042', 'c-77', 'PENDING', 2350);

INSERT INTO outbox (id, aggregate_type, aggregate_id, type, payload)
VALUES ('e-9f1c', 'order', '1042', 'OrderCreated',
        '{"orderId":"1042","customerId":"c-77","totalPence":2350}');

COMMIT;
-- A relay (poller or CDC such as Debezium) later publishes e-9f1c to the broker.
```

Placing an order with a transactional outbox: the order and its event commit together, in one local transaction.

Outbox on the way out, inbox on the way in



The outbox removes the dual write on the sending side; the inbox, or any duplicate check, makes the receiving side safe under at-least-once delivery.

Receiving reliably: idempotent receivers and the inbox

Because the outbox, broker retries and saga retries all produce duplicates, every participant must be an idempotent receiver, the name Gregor Hohpe and Bobby Woolf gave the pattern in *Enterprise Integration Patterns* (2003). Some operations are naturally idempotent ("set order 1042 to CONFIRMED"); others need a duplicate check. The usual implementation is an **inbox**: a table of processed message IDs, written in the same local transaction as the business change, so that "process the message" and "remember that it was processed" commit together. A redelivered message finds its ID already present and is acknowledged without doing anything. The consumer-side code appears in event-driven architecture.

Together, outbox, at-least-once delivery and idempotent receivers give the practical result people usually mean by "exactly once": each business effect happens once, even though messages may travel more than once.

DECISION GUIDE

Decision guide: one transaction, a saga, or new boundaries?

Use this when a business action needs to change data in more than one place.

Q1 **Is all the data that must change together owned by one service and stored in one database?**

- **Yes** Use one local ACID transaction. It is the simplest and strongest option; do not distribute it.
- **No** Go to the next question.

Q2 **Do the parts that must change together nearly always change together, belong to one team and share a bounded context, so that the split is accidental?**

- **Yes** Redesign the boundary: merge the services, or move the data, so the action becomes one local transaction. Sagas are not a fix for a wrong split.
- **No** Go to the next question.

Q3 **Must every participant see all changes at the same instant, with no intermediate state ever visible (for example, a ledger that must always balance)?**

- **Yes** Keep those records in one database, or use a distributed database that provides cross-partition transactions internally. Use 2PC only inside one system's boundary, with participants you control.
- **No** Go to the next question.

Q4 **Can every step either be compensated, or be placed after a pivot and retried until it succeeds?**

- **Yes** Use a saga: orchestrated if it has more than about three steps or any branching, choreographed if it is short and stable. Add an outbox, idempotent receivers and countermeasures such as semantic locks.
- **No** Reorder the steps or change the business process (for example, reserve before charging, or add a manual review) until it can; otherwise reconsider the boundaries.

WATCH OUT

Common mistakes

- Publishing an event after committing, without an outbox, and losing events whenever a service restarts at the wrong moment.
- Writing compensations as "delete the row", destroying the audit trail and confusing anyone who already saw it.
- Forgetting that compensations can fail or arrive before the step they undo.
- Placing an irreversible step, such as sending the order to the kitchen or charging the card, before a step that commonly fails.
- Using choreography for a long, branching flow until nobody can say what happens when step 5 fails.
- Using sagas to paper over a service split that cuts through one consistency boundary; the cheaper fix is often to move the boundary.

CHECK YOUR UNDERSTANDING

▼ Why can a participant in two-phase commit not simply commit on its own if the coordinator disappears after it voted yes?

Because it does not know the decision: another participant may have voted no, so the coordinator may have decided to abort; guessing could break atomicity.

▼ In the order saga, why is confirming the order to the restaurant the pivot rather than the first step?

Once the kitchen starts cooking, the step cannot be undone, so it must come after the steps that might fail (payment authorisation, courier reservation), which can be compensated cheaply.

▼ What problem does the transactional outbox solve, and what guarantee does it give?

It removes the dual write between the database and the broker by committing the change and the message together; the relay then delivers the message at least once, so duplicates are possible.

▼ Name two ways a compensation differs from a database rollback.

It is a new transaction that runs after the original committed, and others may already have seen or acted on the original; it may also be only a semantic undo (a refund, a void) rather than restoring the exact previous state.

▼ Try it: A ticket saga has steps: authorise payment, hold seats, issue ticket, email fan. Seats turn out to be gone at step two. What happens? Which step is the pivot, and why is a compensation not a rollback?

The saga compensates in reverse: it voids the payment authorisation, and the fan is told the seats have gone. Issuing the ticket is the pivot, since once a ticket exists it cannot simply vanish, and later steps such as the email must be retried until they succeed. A compensation is a new forward transaction, not a rollback: others may have seen the hold, so it must be idempotent and safe to retry.

SOURCES

- Garcia-Molina and Salem: Sagas (Princeton technical report, 1987): https://www.cs.princeton.edu/research/techreps/598
- Chris Richardson: Pattern: Saga (microservices.io): https://microservices.io/patterns/data/saga.html
- Chris Richardson: Pattern: Transactional outbox (microservices.io): https://microservices.io/patterns/data/transactional-outbox.html
- Microsoft Azure Architecture Center: Saga design pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/saga
- Debezium documentation: Outbox event router: https://debezium.io/documentation/reference/transformations/outbox-event-router.html
- Temporal: Understanding Temporal: https://docs.temporal.io/evaluate/understanding-temporal
- Temporal: Workflows: https://docs.temporal.io/workflows
- AWS Step Functions: Standard and Express workflows: https://docs.aws.amazon.com/step-functions/latest/dg/concepts-standard-vs-express.html

SECTION 6.8 OF 6.8 · INTERMEDIATE · 28 MIN READ · LAST REVIEWED NOVEMBER 2026

Serverless and space-based architectures

BEFORE YOU READ THIS

This section builds on:

- 0.7 Servers and the cloud
- 6.1 Why distribute
- 6.4 Event-driven

Terms it uses from earlier: Managed service, Idempotent, Partial failure, Eventual consistency, Read replica

THE IDEA IN BRIEF

Think of two ways to cope with customers. A taxi rank charges only for each journey: when nobody needs a ride you pay nothing, but on a busy night each trip costs more than owning a car would, and the first taxi may take a moment to arrive. That is serverless. Now picture a stadium box office before a huge concert: instead of every clerk queueing to check one central ledger, each counter keeps its own copy of which seats are free and sends its sales to the back office later. That is space-based architecture, built for the rush.

WHAT YOU NEED TO KNOW

- Serverless combines functions as a service, where the provider runs event-triggered code and scales it from zero, with backend as a service, where managed services replace server-side components; AWS Lambda (announced 2014) made FaaS mainstream.
- Functions are stateless and short-lived, billed per invocation and per GB-second, subject to cold starts and hard limits on time, memory and payload, and often triggered at least once, so handlers must be idempotent.
- All durable state lives in managed services, so a serverless system is a distributed system by construction and is tightly coupled to its provider unless business logic is kept behind ports.
- Per-request pricing is cheaper than always-on capacity below a crossover volume and more expensive above it; the crossover depends on duration, memory, traffic shape and people costs.
- Space-based architecture serves requests from processing units with replicated in-memory data grids, managed by virtualised middleware, and updates the database asynchronously through data pumps and writers, with data readers to warm the grid.

- Its name comes from the tuple space of Gelernter's Linda (1985); its main risks are data collisions between replicas, memory limits, database lag, cost and complexity.
- Use space-based architecture only for extreme, spiky load that caching, replicas and waiting rooms cannot handle, and route or partition contended data so it cannot collide.

Two answers to the question of capacity

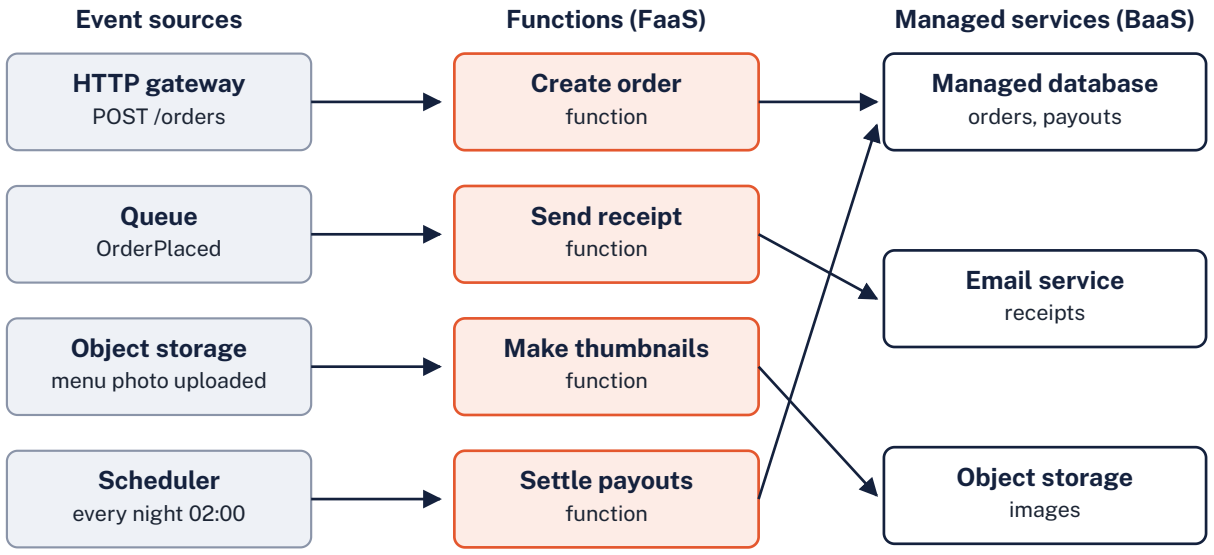
Every style in this chapter so far assumes that someone runs servers and sizes them for the load. This section covers two styles that answer the capacity question in opposite ways. Serverless architecture hands servers to a cloud provider entirely and pays per use, which suits work that is small, event-driven and often idle. Space-based architecture keeps everything a busy request needs in memory, spread over many identical units, so that the database is no longer on the hot path, which suits extreme and sudden peaks such as a ticket on-sale. Both are specialised tools; both are often chosen for the wrong reasons.

Serverless: functions and managed back ends

Serverless does not mean there are no servers; it means the team does not provision, patch, size or pay for idle servers. Mike Roberts' widely cited 2018 article separates two ingredients. Functions as a service (FaaS) runs small pieces of the team's own code in response to events, in containers that the provider creates, scales and destroys, and that may live for only one invocation. AWS Lambda, announced in November 2014 and generally available in April 2015, made the model mainstream; Azure Functions, Google Cloud Run functions and Cloudflare Workers are other examples. Backend as a service (BaaS) replaces server-side components with fully managed third-party services that the client or function calls directly: authentication (Auth0, Amazon Cognito), databases (Firebase, DynamoDB), object storage, email and queues.

A serverless application is usually a combination: functions hold the business logic that is specific to the organisation, and everything else, including all durable state, lives in managed services. The provider handles capacity, availability across zones and the operating system. The team still owns its code, its data, its configuration and its security settings, as the shared responsibility model in servers and the cloud explains.

Events trigger functions; state lives elsewhere



Each function instance is stateless and short-lived; all durable state lives in managed services

In a food-delivery platform, four kinds of event trigger four functions, which read and write managed services rather than keeping anything themselves.

How functions behave: the characteristics that shape design

Event triggers. Functions run in response to events: an HTTP request through a gateway, a message on a queue or stream, a file landing in object storage, a database change, or a schedule. Triggers that invoke the function asynchronously, such as queues and storage notifications, typically deliver *at least once* and retry on failure, so handlers must be idempotent, exactly as in event-driven architecture.

Scale to zero and scale out. With no traffic, no instances run and nothing is billed for compute. As events arrive, the platform starts more instances in parallel, up to an account or function concurrency limit; on AWS Lambda the default regional limit is 1,000 concurrent executions at the time of writing, raisable on request. Rapid scale-out is a feature and a hazard: a thousand concurrent functions can open a thousand database connections and overwhelm a relational database designed for a few hundred, which is why connection proxies and per-function concurrency caps exist.

Per-invocation billing. Cost is usually a small charge per request plus a charge for memory multiplied by execution time (gigabyte-seconds). Code that waits idly on a slow downstream API is therefore paying to wait.

Cold starts. The first request to a new instance must wait while the platform starts a container, loads the runtime and runs initialisation code: from tens of milliseconds to several seconds, depending on language, package size and network set-up. Instances are reused for later requests while warm, so cold starts hit infrequent traffic and sudden bursts hardest. Mitigations include smaller deployment packages, lighter runtimes, initialising outside the handler, and paying to keep instances pre-initialised (AWS calls this provisioned concurrency), which partly reintroduces always-on cost.

Limits. Functions run under hard limits on execution time, memory, payload size and temporary disk; for AWS Lambda at the time of writing, 15 minutes, 128 MB to 10,240 MB of memory (with CPU allocated in proportion) and 6 MB for a synchronous request or response. Long jobs must be split into steps, and large payloads passed by reference to object storage.

Statelessness. Nothing in an instance's memory or local disk is guaranteed to survive to the next invocation. Caching in memory between calls is an optimisation that may or may not hit, never a place for data that matters.

IF YOU ARE NEW TO THIS

Step by step: a function that forgets

The step most people trip over here is that a serverless function cannot rely on keeping anything from one run to the next, so every piece of information that matters must live somewhere else. A small mistake shows why.

- The food-delivery team writes a function that runs each time an order is placed. To show a live count of today's orders, a developer keeps a counter in the function's memory and adds one each time.
- On a quiet morning it seems to work: one copy of the function handles the orders one after another, and the count goes 1, 2, 3.
- At lunchtime orders surge, and the platform starts 50 copies of the function at once, each with its own memory. Each copy keeps its own separate count, so none of them knows the real total.
- When traffic falls, the platform quietly removes the copies it no longer needs, and their counts disappear with them. Tomorrow's first order may start from 1 again, or from whatever a surviving copy happens to remember.
- The fix is to keep the count in a managed service that every copy shares, such as a database or a cache, and to use that service's own 'add one' command, so that two copies adding at once cannot overwrite each other.
- **The lesson:** write every function to be stateless. Treat anything in its memory as a scrap that may vanish at any moment, and decide where each piece of information lives before writing the function, which is what the next part of this section is about.

Where the state goes

Because functions keep nothing, architecture work shifts to choosing where each kind of state lives. Durable business data goes to a managed database, often one that handles many short-lived connections well (key-value or document stores, or a relational database behind a connection proxy). Large objects go to object storage, passed between functions by key. Short-lived shared state, such as rate-limit counters, goes to a managed cache. The progress of a multi-step process goes to a workflow service (AWS Step Functions, Azure Durable Functions or similar), which acts as a durable saga orchestrator, as described in sagas. User sessions go into tokens held by the client or into a session store.

The consequence is that a serverless system is a distributed system by construction, with all the concerns of this chapter: network latency between every function and its data, partial failure, duplicate events, eventual consistency and the need for distributed tracing. A single

business action may cross a gateway, three functions, a queue and two databases. Observability (see observability) is not optional.

Vendor coupling: the price of not running servers

Serverless couples a system to its provider more tightly than renting virtual machines or running containers. Event formats, trigger configuration, identity and permission models, workflow definitions and the managed services themselves are all provider-specific. Moving a mature serverless application to another cloud is closer to a rewrite than a migration.

Practitioners disagree about how much this matters. One view holds that lock-in is the price of a large reduction in operational work and that most organisations never move clouds, so optimising for portability wastes effort. The other holds that pricing changes, regulatory demands or acquisitions do force moves, and that deep coupling removes negotiating power. A middle path is to accept coupling at the edges and contain it in the core: keep business logic in plain modules behind ports, with thin function handlers acting as adapters (the hexagonal architecture approach), keep infrastructure as code, and record the coupling as an explicit decision. Cost and lock-in are discussed further in FinOps and lock-in.

WORKED EXAMPLE

Worked example: when does always-on become cheaper?

The food-delivery platform's restaurant-menu API could run as functions or on two always-on virtual machines. **All prices are illustrative round numbers chosen for the arithmetic, not current list prices of any provider.** Each request runs for an average of 200 ms with 0.5 GB of memory, so it uses 0.1 GB-seconds.

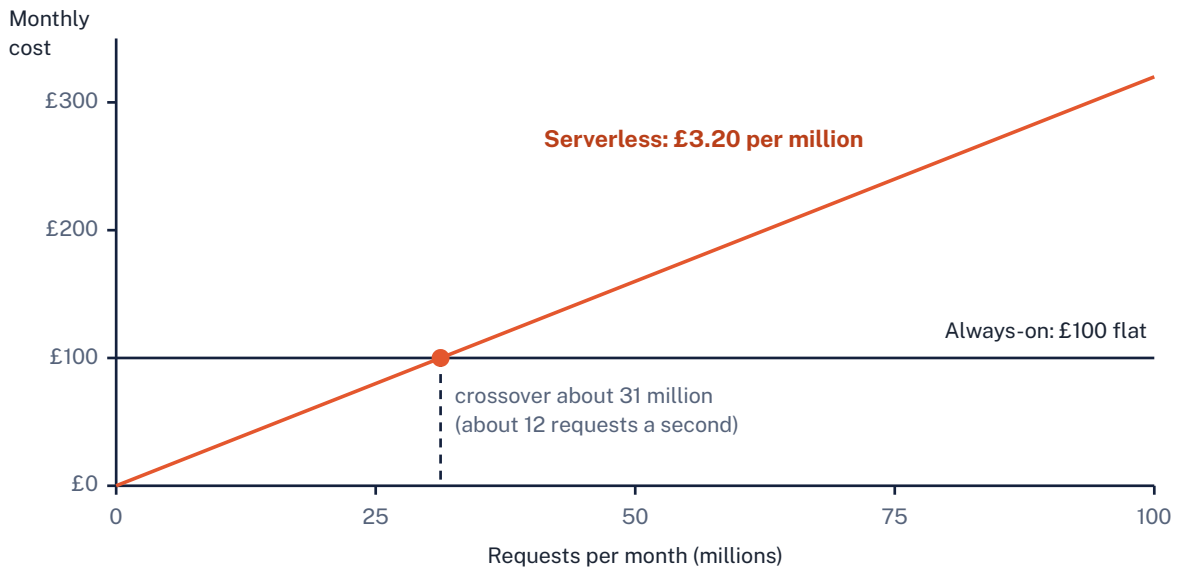
Illustrative serverless prices	Compute £0.00002 per GB-second; £0.20 per million invocations; £1.00 per million requests through the HTTP gateway.
Serverless cost per million requests	Compute: 1,000,000 × 0.1 GB-s × £0.00002 = £2.00. Invocations £0.20. Gateway £1.00. Total £3.20 per million.
Illustrative always-on cost	Two small VMs at £40 a month each (two for redundancy) plus a load balancer at £20 = £100 a month, sized for peaks of about 300 requests a second.
1 million requests a month (0.4 a second on average)	Serverless £3.20; always-on £100. Serverless is about 30 times cheaper.
10 million a month (about 4 a second)	Serverless £32; always-on £100.
Crossover	£100 ÷ £3.20 ≈ 31 million requests a month, about 12 requests a second on average (31.25 million ÷ 2.59 million seconds in 30 days).
100 million a month (about 39 a second)	Serverless £320; always-on still £100, because two VMs still have headroom.
Effect of halving duration to 100 ms	Compute falls to £1.00, total £2.20 per million, and the crossover moves to about 45 million. Optimisation pays directly in serverless.

Below the crossover, serverless is cheaper because idle time is free; above it, a steadily busy service pays more for the same work. Bursty traffic with long idle periods favours serverless even at higher volumes.

The raw comparison leaves out the largest cost for many teams: people. Patching, scaling and on-call for VMs can easily cost more than the difference, which pushes the real crossover higher. Data transfer, logging and observability charges apply to both and are often larger than either compute bill.

Because the result depends on duration, memory, traffic shape and current prices, redo the calculation with real prices and measured traffic before deciding, and again as traffic grows.

The crossover, drawn



With these illustrative prices, serverless cost grows by £3.20 per million requests and meets the flat £100 always-on cost at about 31 million requests a month.

When serverless fits, and when it does not

Serverless fits **spiky, unpredictable or low-volume** workloads, where paying for idle servers is wasteful; **event-driven glue**, such as reacting to uploads, queue messages and database changes; **scheduled jobs** and back-office automation; **prototypes and new products**, where the team should spend its time on the product rather than on infrastructure; and teams without platform or operations capacity.

It fits poorly with **steady, high-volume** traffic, where per-request pricing overtakes always-on capacity (as in the worked example); **long-running or stateful** processes, such as video transcoding beyond the time limit or WebSocket servers holding many connections; **strict, low tail-latency** requirements that cold starts would breach; workloads needing special hardware or fine control of the runtime; and fine-grained designs in which many functions call each other synchronously, which add network hops, cost and failure points at every step. Chapter 1 described how Amazon Prime Video, in 2023, moved one monitoring service from distributed serverless steps to a single process and reported an infrastructure cost cut of over 90% (why architecture matters): not a verdict on serverless, but a reminder that fit matters more than fashion.

Space-based architecture: taking the database off the hot path

Most architectures scale the web and application tiers easily but eventually hit the database, which every request touches. Adding servers in front only moves the queue to the database. Space-based architecture, as described by Mark Richards and Neal Ford in *Fundamentals of Software Architecture*, removes the database from the synchronous path altogether. Each request is handled entirely in memory by one of many identical processing units, and the database is updated asynchronously, behind the scenes.

The name comes from the tuple space, a shared associative memory introduced by David Gelernter's Linda coordination language in 1985: processes communicate by writing tuples into a shared space and reading or taking matching tuples out, without knowing about each other. Sun's JavaSpaces brought the idea to Java in the late 1990s, and commercial platforms built distributed, replicated spaces on it, which evolved into today's in-memory data grids such as Hazelcast, Apache Ignite and Oracle Coherence.

The parts of a space-based system

1. **Processing unit**
   A deployable unit containing the application logic and an in-memory data grid holding the data that logic needs, plus a replication engine. Units are identical, so adding one adds capacity. In the ticketing example, each unit holds the seat map and current holds for the events it serves.

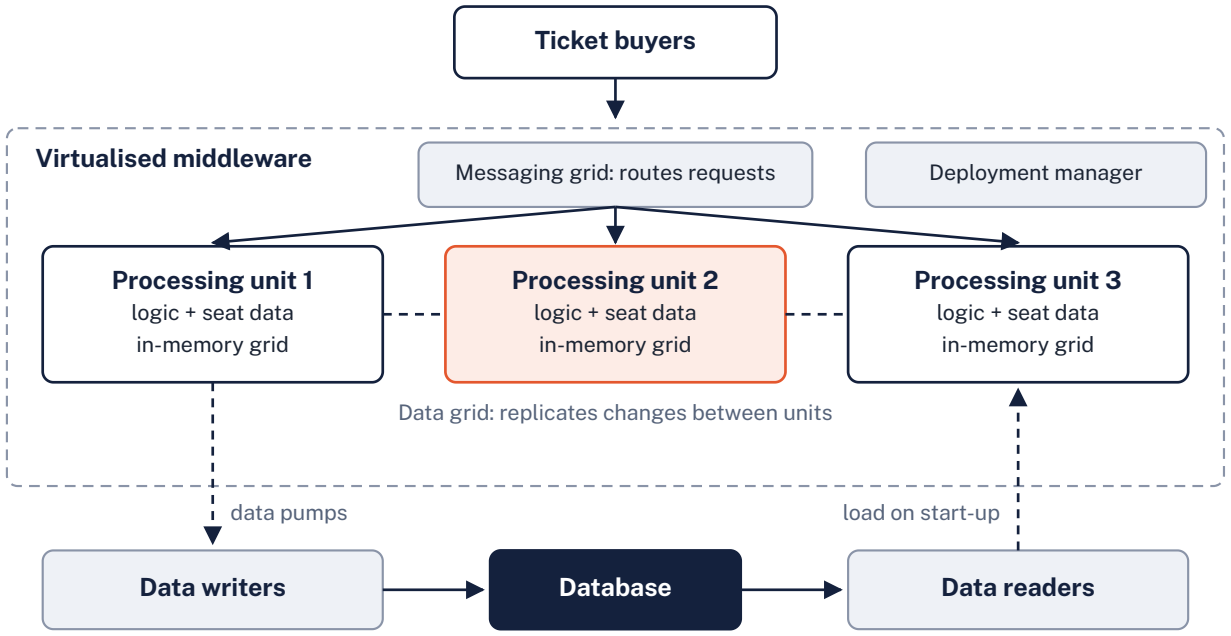
2. **Virtualised middleware**
   The infrastructure that manages the units. The **messaging grid** receives requests and routes each to an available unit. The **data grid** replicates changes between units' in-memory copies. The optional **processing grid** coordinates requests that need several kinds of unit. The **deployment manager** starts and stops units as load rises and falls.

3. **Data pumps**
   Asynchronous channels, usually persistent message queues, that carry each change made in a unit to the database side. If the database is slow or down, changes wait in the queue; buyers are unaffected.

4 **Data writers**
Components that consume the pumps' messages and write the changes to the database. They can be scaled and paused independently of the units.

5 **Data readers**
Components that load data from the database into the units when the grid starts cold, for example after every unit has been restarted or redeployed. They are not used in normal request processing.

Space-based architecture for a ticket on-sale



Requests are served from memory by identical processing units; the database is updated asynchronously through data pumps and writers, and read only to warm the grid.

Data collisions and other costs

Holding replicated copies in many units brings the consistency problems of the previous sections into memory. With a **replicated** cache, each unit has a full copy and changes are copied asynchronously, which gives fast local reads, but two units can update the same item before replication catches up. Richards and Ford call this a data collision and estimate its rate as the number of units multiplied by the square of the update rate, divided by the number of items, multiplied by the replication latency. With 5 units, 20 updates a second, 50,000 items and 100 ms latency, that is 5 × 400 ÷ 50,000 × 0.1 = 0.004 a second, about 115 collisions in an eight-hour day; cutting latency to 1 ms brings it to about one.

For data that must never collide, such as the last seat in a row, the alternatives are a **distributed** cache, in which each item lives in one partition and every unit reads and writes it there (consistent, but every access is a network call and the grid's partitions must stay available), or routing all requests for one event or section to the same unit, so that only one unit ever changes those seats. Most real designs combine the two: partition the hot, contended data by key and replicate read-mostly data such as venue maps and prices.

Other costs are significant. Data held in memory is limited by memory size and must be rebuilt after a full restart. The database is eventually consistent with the grid, so reporting and other systems see changes late, and losing queued changes is a real risk that the pumps' durability must cover. Testing at realistic peak load is expensive and hard. And the style is complex and costly to build: Richards and Ford rate it low for simplicity and testability and high for cost. It is worth it only when the load genuinely cannot be served by caching, read replicas and a well-tuned database.

WORKED EXAMPLE

Worked example: a stadium on-sale

A stadium tour goes on sale at 10:00. About 400,000 people are waiting; for the first ten minutes, demand is hundreds of times the normal rate, then falls away. The ticketing system must not sell a seat twice and must not fall over.

Before the on-sale	Data readers load the seat maps for each show into the grid. The deployment manager starts enough processing units for the forecast peak.
Routing	The messaging grid routes requests by show and section, so all holds for section 112 of Saturday's show are handled by the same unit. Seat holds never collide.
A buyer selects two seats	The unit marks them HELD for this buyer for 8 minutes, in memory, in microseconds, and returns. Seat maps for other buyers update through replication.
The hold becomes a sale	The payment step runs (an external call, with an idempotency key); on success the unit marks the seats SOLD and a data pump queues the sale.
The database	Data writers write sales at a steady rate the database can sustain. During the peak the queue grows; it drains a few minutes after demand falls. No buyer request waits for the database.
A hold expires	The unit releases the seats in memory; the pump queues a release record.

Many ticketing systems put a virtual waiting room in front of the sale as well, admitting buyers at a controlled rate. It is far simpler and often enough on its own; space-based architecture is for when the admitted load itself is still too high for a database-centred design.

Routing by section is the design decision that turns the collision problem from "rare but possible" into "impossible" for the data that matters most.

Characteristics compared (qualitative, for typical designs)

Characteristic	Serverless (FaaS and BaaS)	Space-based
Elasticity	Very high, automatic, including scale to zero	Very high for spikes, but units take time to start and load data
Performance	Good when warm; cold starts and network hops add latency	Very high: requests served from memory
Cost profile	Pay per use; excellent when idle or spiky, expensive when steadily busy	High: many units, large memory, complex middleware, always on
Simplicity	Simple per function; the whole system can become hard to follow	Low: replication, collisions, pumps and cold-start loading
Consistency	Whatever the managed data stores provide	In-memory replication is eventual unless contended data is partitioned; database lags behind
Testability	Units are easy to test; integration needs cloud environments	Hard: realistic peak testing is expensive
Operational effort	Low for infrastructure; high for observability and limits	High
Portability	Low: provider-specific triggers and services	Medium: depends on the data grid product

Checklist: before choosing serverless or space-based

Answer these with measured numbers where you can.

- [ ] **Traffic shape**
  Is load spiky, idle much of the time or steady? Serverless rewards idle time; space-based rewards extreme peaks; steady load favours ordinary always-on services.
- [ ] **Cost crossover**
  Have you calculated per-request cost against always-on cost with real prices, durations and traffic, including gateway, data transfer and people?
- [ ] **Latency budget**
  Can the slowest acceptable response absorb a cold start, or the network hop to wherever the state lives?
- [ ] **Limits**
  Do any jobs exceed execution-time, memory or payload limits? Can they be split into steps or passed by reference?
- [ ] **State and connections**
  Where does each kind of state live, and can the data stores cope with sudden bursts of concurrent connections?
- [ ] **Idempotency**
  Can every handler safely run twice for the same event?
- [ ] **Contention**
  For space-based designs, which data is contended, and will it be partitioned or routed so it cannot collide?
- [ ] **Durability of the lag**
  If the grid or pumps fail, what is lost? Are pumps persistent, and is the database's lag acceptable to reporting and other systems?
- [ ] **Simpler alternatives**
  Would caching, read replicas, a queue in front of the database or a virtual waiting room meet the need with less complexity?
- [ ] **Lock-in decision**
  Is the coupling to the provider or grid product recorded as a conscious decision, with business logic kept behind ports?

CHECK YOUR UNDERSTANDING

▼ Why must a function triggered by a queue be idempotent?

Because asynchronous triggers typically deliver at least once and retry on failure, so the same event can invoke the function more than once.

▼ Using the illustrative prices in the worked example, roughly how much would 50 million requests a month cost on serverless, and which option is cheaper?

50 × £3.20 = £160 a month, so the £100 always-on option is cheaper on raw infrastructure cost, before counting people and other charges.

▼ In space-based architecture, what do data readers do, and when?

They load data from the database into the processing units' in-memory grids when the grid starts cold, such as after a full restart or redeployment; they are not used for normal requests.

▼ How does routing all requests for one stadium section to one processing unit help?

Only that unit ever changes those seats, so two units cannot update the same seat before replication catches up, eliminating data collisions for that data.

▼ Try it: Illustrative prices: compute £0.00002 per GB-second, £0.20 per million invocations, £1 per million gateway requests. Each request takes 300 ms at 1 GB. Two always-on servers cost £288 a month. Find the crossover. What would you pick at 5 million requests?

Each request uses 0.3 GB-seconds, or £6 per million, so the total is £7.20 per million. Crossover: 288 / 7.20 = 40 million requests a month. At 5 million, serverless costs £36 against £288, so choose functions. Traffic shape and people costs also matter, and cold starts may rule it out if latency is tight. If volume nears 40 million, the sums reverse, so keep logic behind ports for an easier move.

SOURCES

- Mike Roberts: Serverless Architectures (martinfowler.com, 2018): https://martinfowler.com/articles/serverless.html
- AWS: Introducing AWS Lambda (announcement, 13 November 2014): https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/
- AWS Lambda Developer Guide: Lambda quotas: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

- Adrian Colyer: Generative Communication in Linda (summary of Gelernter, 1985): https://blog.acolyer.org/2015/02/17/generative-communication-in-linda/

Further reading

- *Fundamentals of Software Architecture (2nd edition)*, Mark Richards and Neal Ford (2025), O'Reilly Media. Clear chapters on service-based, event-driven, space-based and microservices styles, with characteristics ratings, and the extended list of distributed-computing fallacies.
- *Software Architecture: The Hard Parts*, Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani (2021), O'Reilly Media. Trade-off analysis for distributed systems: architecture quanta, service granularity (disintegrators and integrators), data ownership and distributed workflows.
- *Building Microservices (2nd edition)*, Sam Newman (2021), O'Reilly Media. The standard practitioner's guide to microservices, built around independent deployability, with balanced advice on when not to use them.
- *Microservices*, James Lewis and Martin Fowler (2014), martinfowler.com. The article that defined the term for most of the industry, with its nine common characteristics.
  https://martinfowler.com/articles/microservices.html
- *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Gregor Hohpe and Bobby Woolf (2003), Addison-Wesley. The vocabulary of messaging (channels, event and command messages, idempotent receivers, dead-letter channels) that every event-driven design still uses.
  https://www.enterpriseintegrationpatterns.com/
- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. The best explanation of replication, consistency, logs, stream processing and the trouble with distributed systems.
  https://martin.kleppmann.com/2026/03/24/designing-data-intensive-applications-2e.html
- *Building Event-Driven Microservices (2nd edition)*, Adam Bellemare (2025), O'Reilly Media. Event streams as the backbone of a service architecture: event design, schemas, ownership and stream processing.

Glossary

API gateway
A single entry point in front of services that handles cross-cutting concerns such as TLS, authentication, rate limiting and routing.

AsyncAPI
A machine-readable specification for describing message-driven APIs: the channels, the operations that send or receive on them and the message schemas.

Asynchronous communication
An interaction in which the sender hands a message to an intermediary, such as a message broker, and carries on without waiting; the receiver processes it when ready.

At-least-once delivery
A guarantee that every message is processed, possibly more than once; duplicates must be handled by the receiver.

Backend as a service (BaaS)
Fully managed third-party services, such as authentication, databases, storage and messaging, that replace server-side components an organisation would otherwise build and run.

Backend for frontend (BFF)
A thin back end built for one kind of client, such as the mobile app, which calls services and shapes their data for that client.

Backward-compatible change
A change to an interface or schema that existing consumers can survive without modification, such as adding an optional field. Removing or renaming a field is a breaking change.

Broker topology
An event-driven topology with no central coordinator: processors react to events and publish further events, so the workflow emerges from their reactions.

Canonical data model
A single enterprise-wide representation of each business entity, into and out of which all messages are translated. Rarely fits every part of a business well.

CAP theorem
The result, conjectured by Eric Brewer in 2000 and proved by Gilbert and Lynch in 2002, that during a network partition a replicated system cannot be both linearisable and available to every request.

Cascading failure
A failure that spreads: one slow or failed component makes its callers wait and exhaust their own resources, so they fail too, and so on up the chain.

Causal consistency
A consistency model in which operations that are causally related (one could have influenced the other) are seen in the same order by everyone; unrelated concurrent operations may be seen in different orders.

Change data capture (CDC)
Reading a database's own change or replication log to stream every committed insert, update and delete to other systems. Debezium is a widely used open-source CDC platform.

Choreography
Coordination of a multi-step process by participants reacting to each other's events, with no single component in charge.

Cold start
The extra delay when a serverless platform must create and initialise a new function instance before handling a request, from tens of milliseconds to several seconds.

Command
A message asking one specific handler to do something, such as AuthorisePayment. Unlike an event, it can be refused.

Compensating transaction
A new transaction that semantically counteracts the effect of an earlier, already committed one, such as voiding a card authorisation. Unlike a rollback, the original may already have been seen.

Consistency model
The contract a data store offers about which values reads may return given the writes that have happened, and therefore which stale or reordered results clients must be prepared for.

Consumer group
A set of consumers that share the work of reading a topic: each partition is read by one member, and each group keeps its own offsets.

Consumer-driven contract
A contract assembled from what each consumer actually uses of a provider's API; the provider runs all its consumers' expectations in its own build so it knows which changes would break whom.

Contract testing
Testing each side of an integration in isolation against a shared description of their interactions, instead of deploying both together for end-to-end tests.

CQRS
Command query responsibility segregation: using separate models, and often separate stores, for updating data and for reading it.

Cursor pagination
Paging through a collection by passing an opaque token that marks the last item seen, rather than a numeric offset. It stays fast and stable when items are added during paging.

Data collision
In a replicated in-memory cache, two units updating the same item before replication has propagated the first update, so one update overwrites the other.

Database per service
The rule that each service owns its data and no other service reads or writes it directly; others use the owner's API or its events.

Distributed system
A set of programs running on different computers that cooperate by sending messages over a network so that users see one service.

Domain service
In service-based architecture, a separately deployed service that covers a whole business area, such as ordering or payments, from API to data access.

Dual write
Updating two systems, such as a database and a message broker, in separate operations; a crash between them leaves the two inconsistent.

Enterprise service bus (ESB)
Middleware at the centre of classic SOA that routes, transforms and mediates messages between services and often runs business processes as well.

Event
An immutable record of something that has happened, named in the past tense, such as OrderPlaced. Any number of receivers may react to it.

Event notification
An event that only signals a change, often carrying just an identifier, so receivers that need details must ask the publisher.

Event sourcing
Storing every change to an entity as an event, so the event log is the source of truth and current state is derived by replaying it.

Event-carried state transfer
Events that carry enough data for receivers to keep their own copy, so they need not call the publisher.

Event-driven architecture
A style in which components communicate by publishing events about things that have happened and reacting to events published by others, usually through a broker.

Eventual consistency
A guarantee that copies of data that have stopped being updated will converge to the same value after some time, while readers may see stale values in between.

Fallacies of distributed computing

Eight false assumptions about networks listed at Sun Microsystems by L Peter Deutsch and colleagues, with James Gosling adding the eighth: for example that the network is reliable and that latency is zero.

GraphQL

A query language and runtime for APIs, released by Facebook in 2015, in which the client sends a query naming exactly the fields it wants from a typed schema and receives a response of the same shape.

gRPC

A remote procedure call framework, open-sourced by Google in 2015, that sends Protocol Buffers messages over HTTP/2 and generates typed client and server code. It supports streaming in either or both directions.

Hypermedia controls (HATEOAS)

Links in an API response that tell the client which actions it can take next, so the client follows links rather than building URLs itself. Level 3 of the Richardson maturity model.

Idempotency

The property of an operation that has the same effect whether it is performed once or several times, which makes retries safe. Usually achieved with a unique request or event identifier that the receiver remembers.

Idempotency key

A unique value a client sends with a non-idempotent request, such as POST, so that the server can recognise a retry of the same operation and return the original result instead of doing the work twice.

Idempotent consumer

A message receiver that records which messages it has processed, in the same transaction as their effects, so that duplicates change nothing.

In-memory data grid

Software that holds data in the memory of many machines as one logical store, with partitioning and replication, for very fast access. Hazelcast, Apache Ignite and Oracle Coherence are examples.

Independent deployability

The ability to change and release one service to production without changing, testing or releasing any other service at the same time.

Last write wins

A conflict-resolution rule that keeps the write with the latest timestamp and discards the others. Simple, but concurrent updates are silently lost and clock differences can pick the wrong winner.

Linearisability
The strongest single-object consistency model: every operation appears to happen at one instant between its start and end, in an order all clients share and that respects real time, as if there were one copy.

Mediator topology
An event-driven topology in which a mediator receives an initiating event and directs the workflow by sending commands to processors in a defined order.

Message
Any data sent from one program to another through a channel; it may carry an event, a command, a query or a reply.

Microservices
A style in which an application is built as small services, each in its own process, built around a business capability, independently deployable and communicating through lightweight mechanisms.

Monotonic reads
A guarantee that once a client has seen a value, its later reads never return an older one, so data never appears to go back in time.

N+1 query problem
A performance fault in which code makes one query to fetch a list of N items and then one more query per item, giving N+1 round trips where one or two would do.

Network partition
A network fault that splits a system into groups of nodes that cannot communicate with each other, although each group may still be running and reachable by some clients.

Offset
The sequential position of an event within a partition; consumers record the offset they have reached so they can resume after a restart.

OpenAPI
A standard, machine-readable format for describing HTTP APIs (paths, operations, parameters, schemas), formerly the Swagger specification. It drives documentation, code generation and testing.

Orchestration
Coordination of a multi-step process by one component that tells each participant what to do and decides what happens next.

PACELC
Daniel Abadi's extension of CAP: if there is a Partition, trade Availability against Consistency; Else, in normal operation, trade Latency against Consistency.

Partial failure
A situation in which some parts of a distributed system work while others have failed, are slow or cannot be reached, often without the working parts being able to tell which.

Partition (of a topic)
In Kafka and similar logs, one ordered, append-only part of a topic. Events with the same key go to the same partition; order is guaranteed only within it.

Problem details (RFC 9457)
A standard JSON format for HTTP API errors, with members type, title, status, detail and instance, served as application/problem+json. RFC 9457 (2023) replaced RFC 7807.

Protocol Buffers
A language-neutral, compact binary format for structured data, defined in .proto files in which each field has a permanent number. Used by gRPC and many messaging systems.

Publish-subscribe
A messaging pattern in which a publisher sends a message to a topic and every subscriber receives its own copy, without the publisher knowing who they are.

Queue (point-to-point channel)
A messaging channel in which each message is delivered to exactly one of the consumers reading from it, so several instances can share the work.

Quorum (R + W > N)
In a store with N replicas, a write waits for W acknowledgements and a read for R replies; if R + W > N, every read set overlaps every write set in at least one replica.

Read-your-writes
A guarantee that after a client writes a value, that client's own later reads see the write or something newer, even if other clients do not yet.

Saga
A business transaction implemented as a sequence of local transactions, each paired with a compensating transaction, so that either all steps complete or completed steps are semantically undone. Introduced by Garcia-Molina and Salem in 1987.

Saga countermeasure
A design technique that reduces anomalies caused by sagas' lack of isolation, such as a semantic lock (a PENDING state), commutative updates or re-reading a value before updating it.

Saga orchestrator
A component that coordinates a saga by sending commands to participants, tracking progress in a persistent state machine and triggering compensations on failure.

Sequential consistency
A consistency model in which all clients see operations in one agreed order that preserves each client's own order, but that order need not match real time.

Service discovery
A mechanism, such as a registry or DNS, through which clients find the current network addresses of healthy service instances.

Service mesh
An infrastructure layer of network proxies, centrally configured, that provides service-to-service encryption, retries, timeouts, traffic control and telemetry.

Service-based architecture
A distributed style with a few coarse-grained, separately deployed domain services (typically 4 to 12) that often share one database; named by Mark Richards and Neal Ford.

Service-oriented architecture (SOA)
An approach, dominant in enterprises from the early 2000s, that exposes business capabilities as shared services with explicit contracts, typically connected through an enterprise service bus.

Session guarantees
Per-client consistency promises, described by Terry and colleagues in 1994: read-your-writes, monotonic reads, monotonic writes and writes-follow-reads.

Space-based architecture
An architectural style that serves requests from many identical processing units holding data in replicated in-memory grids, updating the database asynchronously so that it is off the request path.

Synchronous communication
An interaction in which the caller sends a request and waits for the reply before continuing, like a telephone call. Both sides must be available at the same moment.

Transactional outbox
A pattern in which a service writes outgoing messages to an outbox table in the same local transaction as its business change; a relay later publishes them, giving at-least-once delivery without a distributed transaction.

Tuple space
A shared associative memory into which processes write tuples and from which they read or take matching tuples, without knowing about each other. Introduced by David Gelernter's Linda language (1985).

Two-phase commit (2PC)
A protocol that makes several participants commit or abort together: a coordinator first asks all to prepare and vote, then tells all the outcome. Participants that voted yes block if the coordinator fails.

XA

The X/Open standard (1991) interface between a transaction manager and resource managers such as databases and message brokers, used to run two-phase commit across them.