Microservices architecture

Verified 2026-08-06

Boundaries, data ownership, sagas, resilience, tracing and contracts — plus an honest account of what the style costs.

Contents

Mental model

A microservice is not defined by its size — it is defined by being **independently deployable**. You adopt the style to stop teams blocking each other at release time, and you pay for that autonomy by turning every former function call into a network call that can be slow, duplicated, or lost. Almost everything below is a consequence of that one trade. Fowler's own guidance is the honest starting point: "don't even consider microservices unless you have a system that's too complex to manage as a monolith".

Service boundaries

Split by	What you get
Business capability	Most changes stay in one service
Technical layer	Every feature touches every service
Database entity	Chatty calls, no clear owner

Lewis and Fowler describe the style as "an approach to developing a single application as a suite of small services, each running in its own process", organized "around business capability". The working test of a boundary is not line count: it is whether a typical feature can ship without touching a second service. Conway's Law from 1968 still decides the argument — "any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure" — so a boundary that cuts across team lines loses to the org chart every time.

```
By capability — one team owns each
  orders/    place, cancel, reprice
  shipping/  label, track, return

By layer — every feature touches all
  web/  logic/  data/
```

> **Gotcha:** A wrong boundary costs far more than a wrong class. “Any refactoring of functionality between services is much harder than it is in a monolith” — moving a method is a compile error, moving a capability is a migration, a deploy, and a data backfill.

Data ownership

Isolation	Verdict
Private tables per service	Weakest form that works
Schema per service	The usual default
Database server per service	Strongest, most operations
One shared schema	Not microservices

The rule is one line: “keep each microservice’s persistent data private to that service and accessible only via its API”. This is the constraint teams quietly skip, and skipping it is what separates microservices from a distributed monolith. The cost is real and worth stating plainly — cross-service joins and cross-service transactions both stop being free.

Queries spanning services get answered by API composition or a CQRS read model, never by reading another service's tables.

```
-- Distributed monolith: the orders service
-- reaching into the customers service.
SELECT o.id, c.name
  FROM orders o
  JOIN customers c ON c.id = o.customer_id;
```

> **Warning:** A shared database gives you network latency *and* deployment coupling, with none of the autonomy you bought them for. If two services must be released together, they are one service wearing two hats.

Communication styles

Style	Coupling	Reach for it
Sync request/reply	Callee must be up now	User-facing reads
Async event	Publisher knows no one	State changed
Async command	Named recipient, no reply	Do this work

Favour "smart endpoints and dumb pipes": keep logic in the services and the transport boring, because a broker that routes on business rules becomes the component nobody dares edit. Every synchronous hop multiplies availability and adds its latency to the caller's, so a chain of four 99.9% services tops out around 99.6% and is as slow as its worst

link. Events invert the dependency — the publisher does not know who listens — which is why emitting `order.placed` ages better than calling the shipping service directly.

```
Sync chain:  A -> B -> C -> D
  uptime   0.999 ^ 4  = 99.6%
  latency  sum of every hop

Event:  A -> [order.placed] -> B, C, D
  A still serves when B is down
```

> **Gotcha:** `await` does not make a call asynchronous in the architectural sense. An awaited HTTP request is still a synchronous hop with the same availability math. What makes a call async is that the caller does not need the answer to finish its own work.

Sagas and consistency

Approach	How it runs	Watch for
Choreography	Events trigger next step	No one sees the whole flow
Orchestration	A coordinator drives steps	Logic piles up in it

There is no distributed transaction across services, so a write spanning several of them becomes a saga — “a sequence of local transactions”,

each committing locally and emitting an event that triggers the next. Nothing rolls back for you: failure is handled by **compensating transactions** you design and implement yourself. Sagas also give up isolation, so two in flight can observe each other's half-finished state; the pattern expects deliberate countermeasures rather than pretending the window does not exist.

```
order.created    -> reserve stock
stock.reserved   -> charge card
payment.failed   -> compensate:
  release stock, cancel order
```

> **Gotcha:** Committing to your database and publishing the event are two systems, and a crash between them loses the event and stalls the saga forever. Write the message to an outbox table inside the same transaction and relay it from there.

Designing for failure

Guard	Prevents
Timeout	Unbounded waiting on a hung callee
Bounded retry, jittered	One blip becoming a stampede
Circuit breaker	Hammering a service already down
Bulkhead	One slow dependency eating every thread
Fallback	A degraded answer instead of a 500

“Applications need to be designed so that they can tolerate the failure of services. Any service call could fail due to unavailability.” The defaults are against you: many HTTP clients ship with no timeout at all, and a call without one is an unbounded resource leak under load. A circuit breaker trips once failures cross a threshold, after which “all further calls to the circuit breaker return with an error, without the protected call being made at all” — then a trial call decides whether to close it again.

```
closed    --failures over limit--> open
open      --cooldown elapsed----> half-open
half-open --trial succeeds------> closed
half-open --trial fails---------> open
```

> **Warning:** A timeout tells you the call did not answer, never that it did not happen. Retrying a non-idempotent request turns one timeout into two charges — retry only what is safe to repeat, and give everything else an idempotency key.

Observability

Signal	Answers
Trace	Where did this one request spend its time
Metric	Is the system healthy right now
Log	What exactly happened inside this step

A stack trace stops at the process boundary, so in a distributed system the request has to carry its own identity. W3C Trace Context standardizes that as the `traceparent` header: a version, a trace id of 32 hex characters, a parent span id of 16, and flags — propagated unchanged through every hop. Wire this up before anything else — without a shared trace id, correlating ten services' logs is manual archaeology, and with one it is a single query.

```
traceparent: 00-<trace-id>-<span-id>-01
             ^      32        16      ^
          version   hex       hex   flags


One trace id, every hop of one request
```

> **Tip:** Return the trace id in error responses and support replies. It turns "checkout was slow yesterday" into an exact lookup across every service that touched the request.

Contracts and deployment

Change	Safe to ship alone
Add an optional field	Yes
Add a new endpoint	Yes
Remove or rename a field	No
Tighten a validation rule	No
Change a field's type	No

Independent deployability survives exactly as long as your contracts stay backward compatible — the moment producer and consumer must ship together, you have lost the property you restructured for. Make breaking changes in two phases: **expand**, adding the new field while still writing the old one, then **contract**, removing the old field once nothing reads it. Consumer-driven contract tests turn “nothing reads it” from a belief into something CI verifies.

```
Expand    write total AND total_cents
          consumers migrate at will
Contract  stop writing total
          only when readers hit zero
```

> **Gotcha:** A version number in the URL does not make a breaking change safe — it makes two versions you run, test, and patch indefinitely. Prefer additive change; spend a new version only on a redesign you are willing to operate twice.

When not to

Signal	Reading
One team, one product	Stay a monolith
Releases must be coordinated	Distributed monolith
No automated deploy or alerting	Not ready yet
Teams blocked by each other	A real reason to split

Fowler is unambiguous: “the majority of software systems should be built as a single monolithic application”, and the premium is paid up front and forever — automated deploys, monitoring, tracing, on-call, and eventual consistency leaking into the domain model. His empirical claim carries more weight than any argument: “almost all the successful microservice stories have started with a monolith that got too big and was broken up”, while greenfield microservice systems “ended up in serious trouble”. Build the modular monolith, let the seams prove themselves, then extract one at a time.

```
monolith -> modular monolith -> extract
            clear modules,      one seam
            single deploy       at a time
```

> **Warning:** Microservices do not repair a struggling monolith. A codebase nobody understands becomes a distributed system nobody understands, with the network added. Fix the module boundaries first — that work is required either way.

Further reading

- Microservices — Lewis and Fowler
- Microservice Premium
- MonolithFirst
- Microservice patterns — Richardson

- W3C Trace Context