

Microservices

Aayush Bajaj

July 10, 2026

Contents

1	the trade, not the hype	2
1.1	what you actually buy	2
1.2	what you actually pay	2
1.3	when a modular monolith wins	2
2	decomposing a system	3
2.1	bounded contexts (ddd-lite)	3
2.2	database-per-service	3
3	communication	4
3.1	sync: rest and grpc	4
3.2	async: messaging	4
3.3	the fallacies of distributed computing	4
4	data consistency: sagas	4
4.1	the problem	4
4.2	worked order flow	5
4.3	choreography vs orchestration	5
4.4	the outbox pattern	5
5	service discovery and the gateway	6
6	resilience	6
6.1	timeouts, retries, jitter	6
6.2	circuit breaker	6
6.3	bulkhead	8
7	observability	8
8	deployment topology	9
9	see also	9

a microservice architecture decomposes one application into many independently deployable services, each owning its own data, talking over a network. the honest framing: you are trading a *code* organisation problem for a *distributed systems* problem. that trade is sometimes worth it. it is not worth it nearly as often as conference talks suggest.

*fowler's original essay remains
the best single description:
martinfowler.com/articles/microservices.html*

1 the trade, not the hype

1.1 what you actually buy

- **independent deployment:** team a ships without waiting for team b's release train. this is the real prize — everything else is secondary.
- **independent scaling:** scale the search service to 40 replicas while billing idles on 2.
- **fault isolation:** a memory leak in recommendations does not take down check-out — *if* you also build the resilience machinery below. isolation is earned, not free.
- **technology heterogeneity:** the ml service can be python while the ledger is jvm. (also a curse: n stacks to patch.)
- **organisational scaling:** conway's law working for you — service boundaries that mirror team boundaries let teams own code end to end.

1.2 what you actually pay

- every in-process function call that crosses a new service boundary becomes a network call: it can now fail independently, time out, arrive twice, or arrive late.
- transactions that were a single **BEGIN...COMMIT** become sagas (below) — you give up atomicity across boundaries and must design compensation by hand.
- refactoring across service boundaries is an order of magnitude harder than moving code between modules — the boundary is now an api contract with independent release cadences (Fowler, Martin, 2018).
- operational surface explodes: per-service ci, dashboards, alerts, on-call, versioned contracts, backwards-compatible migrations.

1.3 when a modular monolith wins

- a **modular monolith** — one deployable, strictly enforced internal module boundaries, one database with schema-per-module discipline — captures most of the design benefit at a fraction of the operational cost.
- you get cheap refactoring while the domain boundaries are still wrong (they always start wrong), real stack traces, local transactions, and one thing to deploy.
- the sane migration path is monolith-first: find the boundaries by living with them in-process, then extract the one or two services that genuinely need independent deployment or scaling — the *strangler fig* approach of routing traffic incrementally to extracted pieces.
- rule of thumb: if one team can hold the whole system in its head and deploys are not blocked on other teams, microservices solve a problem you do not have.

martinfowler.com/bliki/MonolithFirst.html and
[/bliki/StranglerFigApplication.html](http://martinfowler.com/bliki/StranglerFigApplication.html)

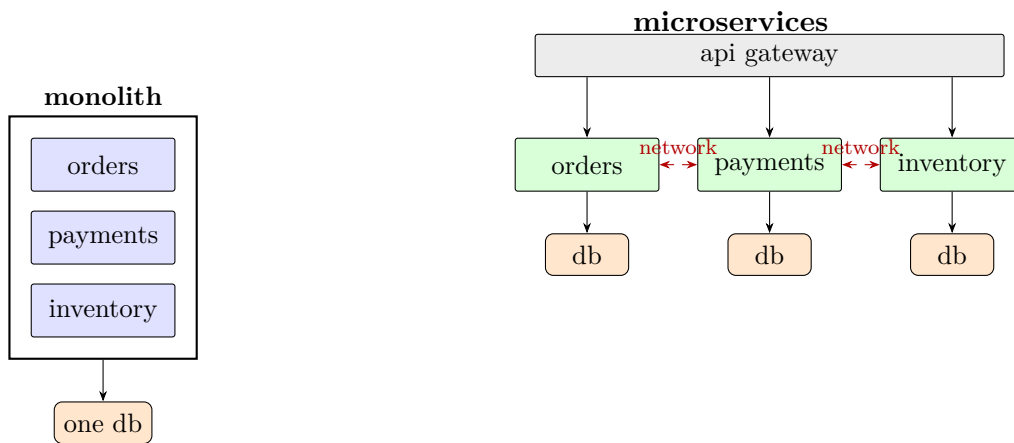


Figure 1: the same domain twice: one deployable with internal modules and one database, versus independently deployable services each owning its own store, fronted by a gateway.

2 decomposing a system

2.1 bounded contexts (ddd-lite)

the useful sliver of domain-driven design:

- a **bounded context** is a region of the system inside which a term means exactly one thing. “customer” in billing (a payment method and an abn) is a different object from “customer” in support (a ticket history) — forcing them into one canonical model couples everything to everything.
- service boundaries should follow bounded contexts, not entities. “user service, order service, product service” carved from database tables produces chatty, anaemic crud wrappers; carving by business capability (checkout, fulfilment, pricing) produces services that can actually change independently.
- heuristics for a good boundary: a change request usually lands in exactly one service; the interface is small relative to what it hides (deep module, not shallow); the team owning it can deploy without a meeting. this is coupling and cohesion at system scale — see [design principles](#).

2.2 database-per-service

- each service owns its schema; nobody else reads its tables. the database *is* the encapsulation boundary — shared databases silently re-fuse the monolith, because a schema migration now needs sign-off from every reader.
- consequence 1: no cross-service joins. data needed for a screen is composed at the gateway/bff layer, or denormalised into a read model (cqrs — see [event driven](#)).
- consequence 2: no cross-service transactions. consistency becomes eventual, and the machinery for that is the saga (below).
- reference patterns for all of this are catalogued at microservices.io.

microservices.io/patterns — richardson's pattern language; the saga, outbox and discovery entries are the ones worth memorising

3 communication

3.1 sync: rest and grpc

- **rest/http**: ubiquitous, debuggable with curl, cache-friendly. json serialisation and http/1.1 head-of-line blocking cost latency; contracts drift unless openapi schemas are enforced in ci. see [api design](#).
- **grpc**: protobuf over http/2 — binary, typed, streaming, generated clients. the natural choice for internal service-to-service calls; the cost is browser-hostility and a codegen toolchain.
- the deeper problem with sync either way: **temporal coupling**. if a needs b to be up to answer, availability multiplies: compose 5 services serially at 99.9% each and the chain delivers $0.999^5 \approx 0.995$ — you built a system less available than any part.

3.2 async: messaging

- a publishes an event to a broker and moves on; b consumes when it can. availability decouples, load spikes buffer in the queue, and new consumers attach without touching the producer — the [observer pattern](#) stretched over a network.
- costs: eventual consistency, duplicate delivery (consumers must be idempotent), harder-to-follow control flow, and a broker to operate. the full treatment lives in [event driven](#).

3.3 the fallacies of distributed computing

deutsch's eight assumptions that every distributed design quietly makes and every network eventually violates:

1. the network is reliable
2. latency is zero
3. bandwidth is infinite
4. the network is secure
5. topology doesn't change
6. there is one administrator
7. transport cost is zero
8. the network is homogeneous

peter deutsch and colleagues at sun, 1990s; still the best checklist for reviewing a service design

every resilience pattern on this page is a countermeasure to fallacy 1 or 2; service discovery exists because of fallacy 5; mtls and zero-trust because of fallacy 4. the underlying transport realities are covered in [internet networks](#).

4 data consistency: sagas

4.1 the problem

placing an order must (a) reserve stock, (b) charge the card, (c) create the shipment. in the monolith this is one acid transaction. across three services there is no

distributed COMMIT worth using — so instead: a **saga** is a sequence of local transactions, each publishing an event that triggers the next, with a **compensating transaction** per step to semantically undo it if a later step fails.

two-phase commit exists but couples availability of all participants and holds locks across the network — the cure is worse than the disease at service scale

4.2 worked order flow

forward path, and what happens when payment is declined:

- **t1** order service: create order in state **PENDING**, emit **OrderCreated**
- **t2** inventory service: reserve stock, emit **StockReserved**
- **t3** payment service: charge card — **fails**, emit **PaymentFailed**
- **c2** inventory service: release the reservation (compensates t2)
- **c1** order service: mark order **CANCELLED** (compensates t1)

compensation is *semantic* undo, not rollback: you cannot un-send an email, so the compensating action is sending the “sorry, cancelled” email. order the steps so the hardest-to-compensate action (charging money, shipping goods) happens last.

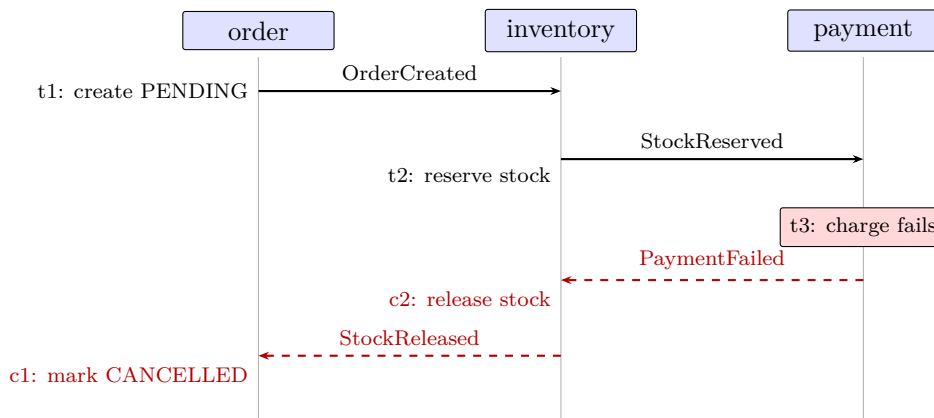


Figure 2: order saga, choreography style: solid arrows are the forward local transactions, dashed red arrows the compensations that run when payment fails.

4.3 choreography vs orchestration

style	flow logic lives	shines when	rots when
choreography	in the events, no owner	2–4 steps, stable flow	flow smeared across services
orchestration	in one orchestrator	long, branching flows	orchestrator turns mini-monolith

- **choreography**: each service reacts to events; nobody owns the flow. finding out “what state is this saga in?” requires archaeology across consumers.
- **orchestration**: a saga orchestrator commands each step and tracks state explicitly — the **mediator pattern**; choreography is observer all the way down. pick per-saga, not per-company.

4.4 the outbox pattern

the lurking bug in every saga: a service commits its local transaction, then crashes *before* publishing the event — db and message stream now disagree. writing to the db and the broker is a distributed write, and you cannot make it atomic directly. the fix:

- **transactional outbox**: within the local transaction, also insert the event into an **outbox** table — one acid commit covers both.
- a relay process (polling, or tailing the write-ahead log — “change data capture”) reads the outbox and publishes to the broker, marking rows sent.
- delivery becomes at-least-once; consumers deduplicate by event id. exactly-once *processing* is built from at-least-once delivery plus idempotency, never assumed from the transport.

5 service discovery and the gateway

- **discovery**: instances are ephemeral (autoscaling, redeploys), so callers cannot hardcode addresses. either client-side (consult a registry like consul/eureka, pick an instance, load-balance in the client) or server-side (dns/virtual ip in front; kubernetes services do this — the registry is the platform). server-side via the platform is the modern default; run a registry yourself only when you must.
- **api gateway**: single entry point that routes to services and centralises the cross-cutting stack — tls termination, authn, rate limiting, request logging. it is the façade pattern at network scale (Gamma, Erich and Helm, Richard and Johnson, Ralph and Vlissides, John, 1994), and pairs with **bff** (backend-for-frontend): one gateway shaped per client type, so the mobile app is not stitching six calls together on a flaky radio.
- the gateway is on every request path: keep it dumb (routing + policy), never let business logic leak in, and make it highly available before anything else.

6 resilience

the network will fail (fallacy 1). resilience patterns bound the blast radius.

6.1 timeouts, retries, jitter

- **every** network call gets a timeout. “no default in your http client” is not a decision — it is an unbounded queue of hung threads waiting to happen.
- retry only idempotent operations, only on transient errors, with **exponential backoff plus jitter**: attempt k sleeps $\text{rand}(0, \min(\text{cap}, b \cdot 2^k))$. the randomness matters: deterministic backoff synchronises clients into retry waves that re-kill the recovering service.
- retries multiply load downstream — a retry storm three layers deep turns one failure into $3^3 = 27$ requests. budget retries (e.g. a fraction of live traffic) and never retry a circuit-open error.

the aws architecture blog post ‘exponential backoff and jitter’ has the simulations: full jitter wins

6.2 circuit breaker

stop calling a service that is down; fail fast and give it room to recover.

- **closed** (normal): calls pass through; count consecutive failures.
- **open** (tripped): after N consecutive failures, fail immediately without calling — protects the caller’s thread pool and the callee’s recovery.

- **half-open** (probing): after a recovery timeout, let one probe through. success closes the breaker; failure re-opens it.

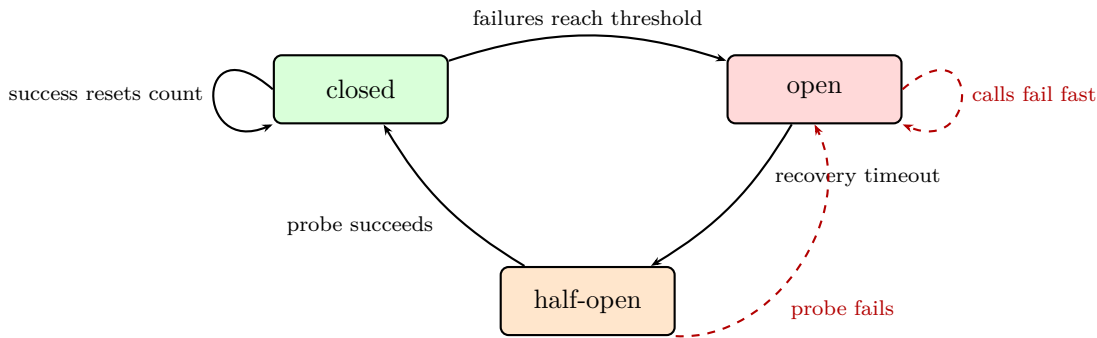


Figure 3: circuit-breaker state machine: failures trip it open, a recovery timeout admits a single probe, and the probe's outcome decides between closed and open.

a minimal implementation — the **state pattern** earning its keep — driven with a fake clock against an upstream that dies at $t = 0$ and recovers at $t = 9$:

```

1  import enum, time
2
3  class State(enum.Enum):
4      CLOSED = "closed"; OPEN = "open"; HALF_OPEN = "half-open"
5
6  class CircuitOpenError(Exception):
7      pass
8
9  class CircuitBreaker:
10     def __init__(self, threshold=3, recovery_s=5.0, clock=time.monotonic):
11         self.threshold, self.recovery_s, self.clock = threshold, recovery_s,
12         ↪ clock
13         self.state, self.failures, self.opened_at = State.CLOSED, 0, None
14
15     def call(self, fn, *args):
16         if self.state is State.OPEN:
17             if self.clock() - self.opened_at >= self.recovery_s:
18                 self.state = State.HALF_OPEN # allow one probe through
19             else:
20                 raise CircuitOpenError("failing fast")
21         try:
22             result = fn(*args)
23         except Exception:
24             self._on_failure()
25             raise
26         self._on_success()
27         return result
28
29     def _on_failure(self):
30         if self.state is State.HALF_OPEN: # probe failed: re-open
31             self._trip()
32         else:
33             self.failures += 1
34             if self.failures >= self.threshold:
35                 self._trip()
36
37     def _on_success(self):
38         self.state, self.failures = State.CLOSED, 0
39
40     def _trip(self):
41         self.state, self.opened_at = State.OPEN, self.clock()

```

```

42 # --- drive it with a fake clock and a service that dies then recovers ---
43 now = [0.0]
44 cb = CircuitBreaker(threshold=3, recovery_s=5.0, clock=lambda: now[0])
45 healthy = [False]
46
47 def flaky():
48     if not healthy[0]:
49         raise ConnectionError("upstream dead")
50     return "200 ok"
51
52 script = [(t, "call") for t in (0, 1, 2, 3, 4, 7)] + [(9, "recover"), (12,
53     ↪ "call"), (13, "call")]
54 for t, action in script:
55     now[0] = float(t)
56     if action == "recover":
57         healthy[0] = True
58         print(f"t={t:>2} upstream recovers")
59         continue
60     try:
61         r = cb.call(flaky)
62         print(f"t={t:>2} call -> {r:<12} state={cb.state.value}")
63     except CircuitOpenError:
64         print(f"t={t:>2} call -> fast fail state={cb.state.value}")
65     except ConnectionError:
66         print(f"t={t:>2} call -> upstream err state={cb.state.value}")

```

```

t= 0  call -> upstream err state=closed
t= 1  call -> upstream err state=closed
t= 2  call -> upstream err state=open
t= 3  call -> fast fail state=open
t= 4  call -> fast fail state=open
t= 7  call -> upstream err state=open
t= 9  upstream recovers
t=12  call -> 200 ok state=closed
t=13  call -> 200 ok state=closed

```

read the trace: the third failure at $t = 2$ trips the breaker; $t = 3, 4$ fail fast without touching the upstream; at $t = 7$ the recovery timeout has elapsed so a probe goes through, fails, and re-opens the breaker (resetting the timer); the $t = 12$ probe succeeds and the breaker closes.

6.3 bulkhead

- partition resources so one failing dependency cannot exhaust them all: separate connection pools / thread pools / semaphores per downstream, named after ship compartments.
- without it: service *c* hangs, all 200 of your worker threads end up blocked waiting on *c*, and now *you* are down for requests that never needed *c*. with a 20-thread bulkhead for *c*, the other 180 keep serving.

7 observability

- with one process you read a stack trace; with thirty services you reconstruct a story from fragments. the fix is **correlation**: the gateway assigns a request id, every service logs it and propagates it on every outgoing call (the `w3c traceparent` header), so one grep reassembles the request's whole life.
- **distributed tracing** (opentelemetry; jaeger/zipkin lineage): each unit of work

is a **span** (name, start, duration, parent), spans share a **trace id**, and the ui renders the tree as a flame graph across services — where the latency actually went, which calls fanned out serially that should have been parallel.

- the three pillars: structured logs (searchable, carrying trace ids), metrics (cheap aggregates for alerting — rate, errors, duration per endpoint), traces (expensive, sampled, for debugging). instrument at the platform layer once, not per-service by hand.

8 deployment topology

- one service instance per **container**, grouped and scheduled by an orchestrator — mechanics in **containers**.
- **sidecar** pattern: a per-instance helper container handles mtls, retries and telemetry; a fleet of sidecars plus a control plane is a **service mesh** (istio/linkerd lineage) — the resilience section above, moved out of application code into infrastructure. adopt when the polyglot service count makes per-language client libraries untenable, not before.
- rollout strategies: **rolling** (replace instances gradually), **blue-green** (two full environments, flip the router), **canary** (small traffic slice to the new version, watch error rates, ramp). canary plus good metrics is the only one that catches the bugs your tests missed — see **testing** for what tests do catch.
- schema changes deploy in **expand/contract** phases: add the new column/contract, migrate readers, remove the old — never a breaking change in one step, because you cannot deploy thirty services atomically.

9 see also

- **event driven** — the async half of this page, done properly
- **api design** — the contracts these services expose
- **containers** — the unit of deployment
- **design patterns** — façade, mediator, observer and state, all reappearing at network scale
- **design principles** — coupling and cohesion, the whole game

references

Fowler, Martin (2018). *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

Gamma, Erich and Helm, Richard and Johnson, Ralph and Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional.