Engineer

Microservices

an architecture of small services, each deployed on its own

What it is

Microservices is an architectural style that splits one application into small services, where every service is a program of its own that runs as a separate process and can be deployed without the others. James Lewis and Martin Fowler, whose 2014 article gave the style its best-known description, sum it up as building applications as “suites of independently deployable services”. They list what such systems tend to share: services that talk over lightweight mechanisms, often an HTTP resource API; services built around business capabilities; deployment that is fully automated; a bare minimum of central management; and the freedom for each service to use its own programming language and its own storage. They are careful to add that there is no formal definition, only characteristics that most systems carrying the label show.

The word is younger than the idea. A footnote in the same article records that a workshop of software architects near Venice discussed the term in May 2011, and that the same group settled on “microservices” in May 2012. Adrian Cockcroft at Netflix had been calling the approach a fine-grained form of SOA, and Lewis and Fowler make no claim that the style is new: they find its roots at least as far back as the principles Unix was built on.

The alternative it is usually measured against is the monolith: a server-side application built and deployed as one unit. All the logic for a request runs in one process. With some care the whole application can be run and tested on one developer’s laptop, and it scales by running more copies of it behind a load balancer. The friction comes later. A small change means rebuilding and redeploying everything, a good modular structure is hard to keep over the years, and the part that needs more capacity can only be scaled together with everything else.

Lewis and Fowler define a component as a unit of software that can be replaced and upgraded on its own. A library is linked into a program and called in memory. A service is a component outside the process, reached over the network through a web request or a remote procedure call. The difference matters for coupling. In most languages, only documentation and discipline stop other code from reaching into a library's internals, while a service can be reached only through explicit remote calls, so its interface is much harder to bypass. A service may still be more than one process, for instance an application together with a database that only it uses.

The "micro" says less than it seems to. Lewis and Fowler point out that the name draws attention to size and leads to arguments over what counts as micro. The largest services they heard of followed Amazon's two-pizza team, meaning no more than a dozen people, and at the small end they saw a team of half a dozen look after half a dozen services.

Use the concept in a decision

Put "What are microservices?" to work in a real decision. We can help relate the concept to the system you are building or buying. The part users never see: where the data lives, what the rules are, and how other systems are allowed to talk to it.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

The first payoff is release on its own schedule. When an application is split into services, many changes need only one service to be redeployed, so one team can ship without waiting for the others. Lewis and Fowler are clear that this is not absolute: a change to a service's interface still has to be coordinated with the services that call it.

The second is that the software can follow the organisation, rather than fight it. In 1968 Melvin Conway argued in Datamation that organisations which design systems are constrained to produce designs that copy their own communication structures. Lewis and Fowler apply this to the usual way of splitting a large application by technical layer, with one team for the UI, one for server-side logic and one for the database. Even a simple change then becomes a project across teams, and logic ends up wherever a team happens to have access. The

microservice style turns that round. Each service covers one business capability, such as orders, stock or payments, from the UI down to storage, and belongs to a cross-functional team with the skills the whole slice needs. That team should own the service for its whole life instead of handing it over at the end of a project, and Amazon's version of the rule is that whoever builds a service also runs it in production.

The third is that the logic stays in the services. Lewis and Fowler sum up the preference as smart endpoints and dumb pipes. Services talk through HTTP requests to resource APIs, or through a lightweight message bus that does nothing more than route messages, and the deciding happens in the services at either end. That is a deliberate step away from the enterprise service bus, which routes, transforms and applies business rules in the middle of the network rather than at its ends.

Data is split along the same lines. Each service manages its own database, and different services may use different database systems if that suits their work. Eric Evans's domain-driven design gives the reasoning: a large system holds several models, and each needs an explicitly bounded context, set in terms of team organisation, of the parts of the application it governs, and of physical things such as code bases and database schemas. The sales view of a customer and the support view of a customer can then differ without anyone having to reconcile them in one table.

The costs are part of the definition, and Lewis and Fowler state them plainly. A remote call costs far more than a call in memory, so interfaces between services have to be coarser. Moving code across a service boundary is hard: interface changes must be coordinated, layers of backwards compatibility added, and testing gets more complicated. Distributed transactions are hard to implement, so services coordinate without them, accept that consistency may only be eventual, and repair mistakes with compensating operations. Any call can fail. And if the services do not fit together cleanly, the complexity has not gone away, it has moved into the connections between them, where it is harder to see. Lewis and Fowler report one argument they consider reasonable: build a monolith first, give it clean modules, and break it up only once it starts to cause trouble. They add a caveat: an interface that works well inside one process seldom makes a good interface between services.

What it looks like in practice

The note Selling the last ticket exactly once reaches for a service only when one database is no longer enough, and then states the rule that holds the design together: every item has exactly one owner of its stock, an inventory service, and nothing else decides. That service is sharded by tenant and item, and the largest tenants get cells of their own. Only the reservation of stock has to be synchronous. Payment, the order and the receipt are separate systems, and no transaction spans them. The note rejects two-phase commit, because it blocks while any participant is slow and payment providers do not take part in it, and uses a saga instead: a sequence of local steps, each with a step that undoes it.

Each step writes its message to an outbox table in the same transaction as the change it announces. In PostgreSQL, a data-modifying statement inside `WITH` can pass its result to an insert, so the inventory service can reserve a unit and record the message in one statement:

```
-- inventory service, its own database
WITH held AS (
    UPDATE inventory
    SET available = available - 1
    WHERE tenant_id = 't-7' AND item_id = 'item-881'
      AND available >= 1
    RETURNING item_id
)
INSERT INTO outbox (order_id, item_id, event, units)
SELECT 'A-17', item_id, 'StockReserved', 1 FROM held;
-- sold out: no row matches, so no message is written
```

Change data capture then publishes the outbox rows, and every consumer is idempotent, keyed by the order, because messages arrive at least once. In this sketch no other service touches the `inventory` table: the others learn about stock from these messages, or by asking the inventory service.

The same note also shows where the split belongs in the order of work. In its list of what to do first, giving inventory its own service and its own owner comes fourth, after measuring, after turning the write into one conditional statement and after shedding load with a waiting room, a gate and a cached sold-out answer. The cheaper fixes come first.

The portfolio case A maritime platform built in clean layers records the founder's architecture contribution under NextMariner and shows the other side of the choice. It was a young product whose requirements were going to move a lot, and the aim was that changing a business rule should not mean touching the frontend, the API and the database at once. It settled on three layers, each with one job: a Next.js frontend for presentation, a deliberately thin Go API that routes, validates, filters for security and orchestrates, and all business logic in PostgreSQL functions. The boundaries were written down and enforced in review. Lewis and Fowler note that modular lines inside one application take a great deal of discipline to hold, and that the more explicit separation of services makes team boundaries easier to keep clear. This platform kept the discipline without splitting its business logic into separate services.

Where it goes wrong

Chains of synchronous calls multiply downtime. When answering one request means the first service calls a second, which calls a third, the request succeeds only while all three are up, and it waits for each of them in turn. Lewis and Fowler call this the multiplicative effect of downtime, and cite the Guardian's rule of one synchronous call per user request. Keep the synchronous path as short as the note keeps it, where only the reservation waits for an answer and everything after it is a message. Give each remaining call a timeout and a circuit breaker, the patterns Lewis and Fowler point to in the book Release It!, so that a failing service is cut off instead of waited on.

Services that always change together. A split along the wrong lines produces services that cannot be released alone: every feature touches three of them, and they ship in lockstep. Lewis and Fowler give the test: if you keep changing two services together, they should be merged. A related failure is turning method calls into remote calls one for one, which leaves services chatting over the network and performing badly. Draw boundaries around business capabilities and bounded contexts, make interfaces coarse, and design each service to tolerate changes in the services it calls, so a supplier can grow without breaking its consumers; Lewis and Fowler treat versioning as a last resort.

Two services share one database. When the order service reads and writes the inventory tables directly, the schema becomes an interface that nobody owns, neither team can change it alone, and stock gains a second writer that the inventory service's checks never see. Give every piece of data one owner, as the

note gives stock one, and have other services ask that owner or read its messages. When an update spans services, use a saga with compensating steps rather than a distributed transaction.

Nobody can follow a request. A request that crosses several services leaves a fragment of its story in each one's logs. Lewis and Fowler give real-time monitoring a central place, with technical numbers such as requests per second to the database watched side by side with business numbers such as orders per minute. They also warn that services reacting to each other's events produce emergent behaviour that monitoring has to catch quickly. Trace every request. In OpenTelemetry's model, a trace is the path of one request through the system, built from spans that share a trace id and point to their parent, and context propagation is what lets spans from different services be joined into one trace. Pass the trace context on every call and with every message, including those that go through the outbox, or the trace ends at the first asynchronous hop.

Further reading

- Microservices — James Lewis and Martin Fowler, 2014: the nine characteristics, the history of the name, the sidebars on size and synchronous calls, and the case for caution.
- How Do Committees Invent? — Melvin Conway's 1968 paper from Datamation, where a system's structure copies the communication structure of the organisation that designed it.
- Domain-Driven Design Reference — Eric Evans's summary of the patterns, including bounded context and the maps of how contexts relate.
- OpenTelemetry: Traces — spans, trace ids, parent spans and context propagation, the pieces that make one request visible across services.
- PostgreSQL: WITH Queries — data-modifying statements in `WITH`, which let one statement change a row and record the change elsewhere.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What are microservices?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/microservices/