

Software Architecture

Reading time: ~10 min · Prereqs: Lessons 01–05 · *Snapshot 2026 — concepts are stable; specific brand names move.*

Why this matters to you

When engineers say "architecture", they don't mean buildings — they mean the shape of a software system. The same building blocks (code, databases, networks) can be arranged dozens of ways, and the arrangement matters enormously. A bad architectural choice early on becomes a multi-year, multi-million-dollar headache. A good choice keeps a small team moving fast for a decade. The vocabulary in this lesson — *monolith*, *microservices*, *serverless*, *event-driven* — shows up in every architecture review, every engineering job ad, and every "why is our cloud bill so high" meeting. Once these four words click, you can read most diagrams, follow most technical debates, and ask sharper questions in the rooms where the bills get paid.

1. What "architecture" actually means in software

Architecture is *how* you arrange a system's parts once it's bigger than one file or one team. The same problems buildings have: **load** (how many users?), **traffic** (how do requests flow?), **repair** (how do you fix things at 3 AM?), **evolution** (how do you add a feature without breaking what's there?). The job of a **software architect** — often a senior engineer, sometimes a dedicated role — is to make those trade-offs explicit and write them down before the team commits to them in code. There are perhaps a dozen named architectural patterns; four of them dominate 2026 conversations, and the rest are usually variations or combinations of these. Get the four words and you can follow the room.

2. Monolith — one big app

A **monolith** is one codebase, one process, one deploy. Almost every famous startup began this way, and many still run this way at very large scale.

Pros: easy to reason about (one place to debug, one place to deploy); no network between modules (every function call is in-process — fast and reliable); one team can own the whole thing end-to-end. New engineers can read the codebase top to bottom and understand the system.

Cons: as the app grows, every change has to be rebuilt and re-tested; a bug in one corner can crash the whole process; many teams editing one codebase step on each other's pull requests. Famously, **Shopify**, **GitHub**, and **Basecamp** ran monoliths for years at extraordinary scale. The modern variation is the **modular monolith**: internally well-bounded modules with clear contracts, externally still one deploy. Many teams now see this as the sweet spot for the first several years of a product.

3. Microservices — many small apps

Microservices split the system into many small services, each owning one slice of the business (the "users" service, the "orders" service, the "payments" service) and talking to each other via APIs.

Pros: each team owns its service end-to-end and ships independently; you can scale only the hot parts of the system; different services can use different languages or databases without forcing the rest of the org to switch.

Cons (and they're big): every API call is now a network call — which can fail, time out, or be slow; operational complexity multiplies, because instead of one deploy, one monitoring target, and one log stream, you have dozens; finding the cause of a slow request now means tracing across services. The honest summary: microservices are an *organizational* solution as much as a technical one — they pay off most when you have many teams that need to ship without coordinating every release. Below maybe 50 engineers, the overhead usually outweighs the benefit.

4. Serverless — functions on demand

Serverless means you ship a function, and the cloud provider runs it on demand. Brand names as of 2026: **AWS Lambda** (the original), **Cloudflare Workers**, **Google Cloud Functions**, **Azure Functions**, **Vercel Functions** (under the hood, often Lambda).

Pros: no servers to maintain, no capacity planning, the bill is per-invocation (you pay nothing while idle), and the platform scales effortlessly to thousands of concurrent calls. **Cons: cold starts** (the first call after idle is slow); vendor lock-in (your code now leans on the platform's primitives); debugging is harder (no SSH'ing into a "server"); and at very high steady volume, serverless is often *more* expensive than just running a fixed VM. The rough rule-of-thumb: serverless is a great fit for spiky, unpredictable workloads and a poor fit for steady, high-throughput ones.

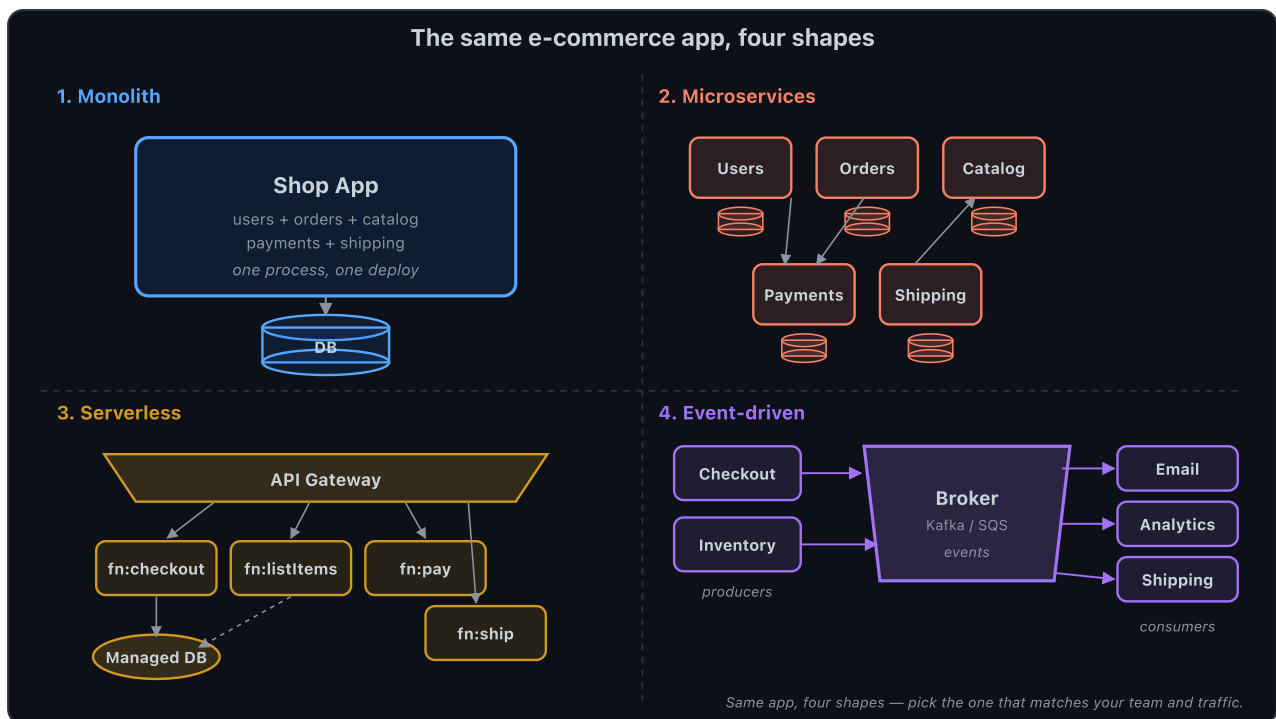
5. Event-driven — publish-subscribe

In an **event-driven** system, components don't call each other directly. Instead, when something happens, the originating component publishes an **event** (e.g., `OrderPlaced`, `PaymentReceived`) to a **broker** — **Kafka, RabbitMQ, AWS SQS / SNS / Kinesis, NATS, Redis Streams**. Other components subscribe to events they care about and react.

Pros: components are decoupled (the publisher doesn't know or care who's listening); you get a natural audit trail in the event log; you can add new consumers without changing the producer. **Cons:** tracing a single user action across N event handlers is hard; *eventual consistency* is the default (the world doesn't update instantly everywhere); ordering and exactly-once delivery are surprisingly tricky to get right. Most large e-commerce, fintech, and logistics systems use events for at least *part* of the workload — rarely the whole thing.

6. The same app, four ways

Here is the same e-commerce application drawn in each of the four shapes. Same boxes, same job, very different arrangements.



7. The anti-pattern: distributed monolith

The most common architectural mistake of the past decade: split the monolith into "microservices" but require all of them to deploy together. This happens when API contracts between services are too brittle, when services share a database, or when services were split by *technical layer* ("the auth microservice", "the email microservice") instead of by *business domain* ("the orders service"). You now have the operational cost of microservices and the development friction of a monolith — the worst of both. The fix is usually long and political. Practical guidance from most senior architects in 2026: **start with a monolith. Split only when team size or scale forces the issue.**

What people argue about

- **"Microservices everywhere" vs "monolith first".** The pendulum has currently swung toward monolith-first, after a decade of microservices fashion. Amazon Prime Video famously moved one service *off* microservices and cut its bill by 90%; the post made the rounds for a reason.
- **Serverless at scale.** Two passionate camps. One: "scales effortlessly, no ops, focus on the product". The other: "your bill scales effortlessly too, and

you can't optimize what you don't run". Both sides have real war stories — the right answer depends on the workload shape.

- **Event-driven by default.** Some teams (especially in finance and logistics) make events the default for everything and treat synchronous calls as the exception. Others say events introduce eventual-consistency bugs that simple synchronous calls would never have. Both are right inside their context.

Reading on

- **Building Microservices** by Sam Newman — the canonical book on the topic; balanced and honest about the costs as well as the benefits.
- **Monolith First** — Martin Fowler's short essay, martinfowler.com/bliki/MonolithFirst.html. Five minutes; the single most-cited argument for starting simple.
- **AWS Well-Architected Framework** — free; one short chapter per architectural concern (reliability, performance, cost, security, operations). A good vocabulary expander even if you're not on AWS.

LAST-REVIEWED: 2026-05-25