Monolith vs Microservices

Complete Interview Guide

Index

01

The Core Difference

The interview answer, and the example used throughout this guide

The Interview Answer

What's the difference between Monolith and Microservices?"

A monolith is a single application containing all of a system's functionality in one codebase and one deployment. Microservices split that same functionality into separate, independently deployable services.

The real difference isn't just 'one app vs many' — it's that microservices let you scale and deploy each part independently, at the cost of added operational complexity: network communication between services, more moving parts to monitor, and the need to handle partial failures.

The Running Example

This guide uses one example throughout: a food delivery app made up of three parts — Orders, Payments, and Restaurants — the same example most interviewers will expect you to reach for.

```
Orders              Payments            Restaurants
Placing and         Handling            Managing restaurant
tracking orders     transactions        listings & menus

Right now, all three live inside the same application.
```

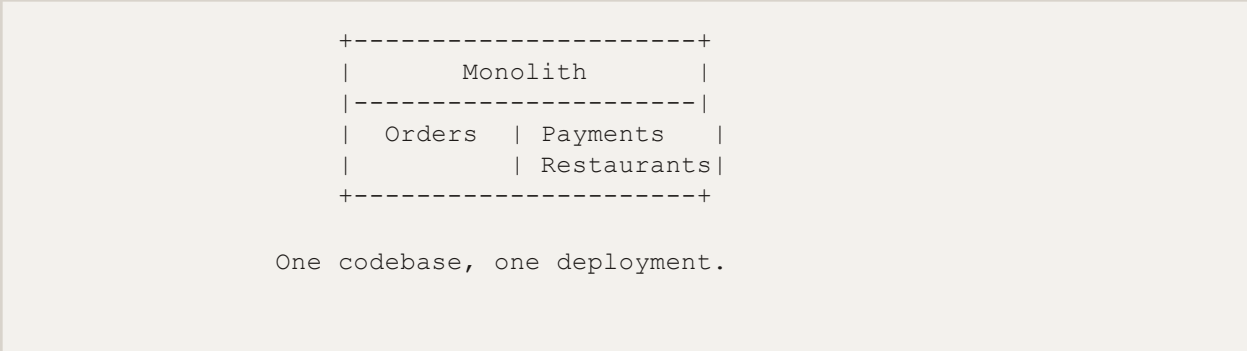
02

What is a Monolith?

One codebase, one deployment, everything together

Definition

A monolith is an application built and deployed as a single unit. Every feature — Orders, Payments, Restaurants, and anything else — lives in the same codebase, runs in the same process, and is deployed together as one artifact.



What "Simplicity" Actually Means Here

In practice

One repository to check out and run. One build pipeline. One deployment to monitor. A function call between Orders and Payments is just a regular in-process function call — no network involved, no serialization, nothing to retry if it fails partway through. For a small team or an early-stage product, this is a genuine advantage, not just a lack of sophistication.

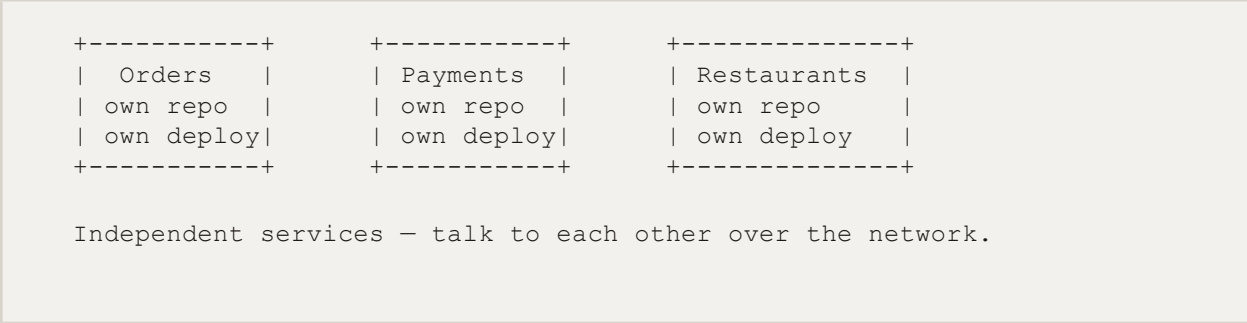
03

What is Microservices?

Independent services, independently deployed

Definition

Microservices is an architectural style where a system is split into separate, independently deployable services, each owning a specific piece of functionality. Orders, Payments, and Restaurants each become their own service, with their own codebase, their own deployment pipeline, and typically their own database.



What "Independent Scaling and Deployment" Actually Means

In practice

Orders can be redeployed ten times a day without touching Payments or Restaurants at all. Orders can run on twenty instances while Payments runs on two. Different services can even be written in different languages, since each is its own independent process communicating over the network rather than through in-process function calls.

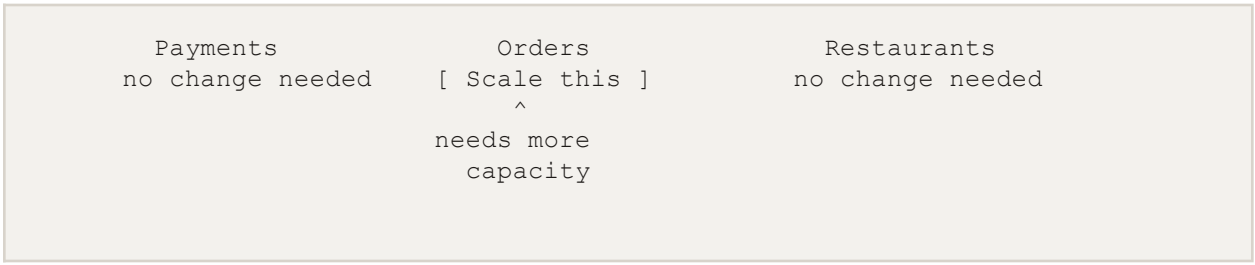
04

The Scaling Problem

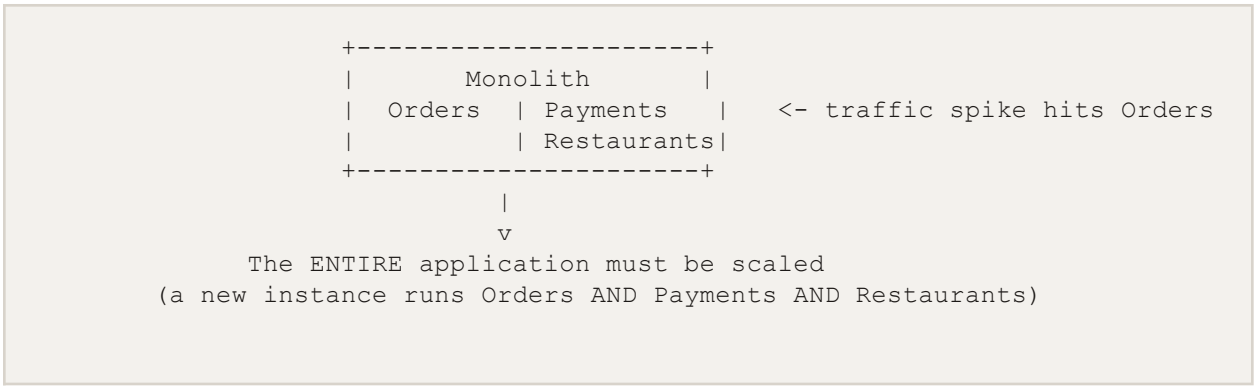
The exact example most interviewers expect

The Scenario

Orders traffic suddenly increases 20x — a flash sale, a marketing push, a viral moment. The goal is simple: give Orders more capacity. Nothing else needs to change.



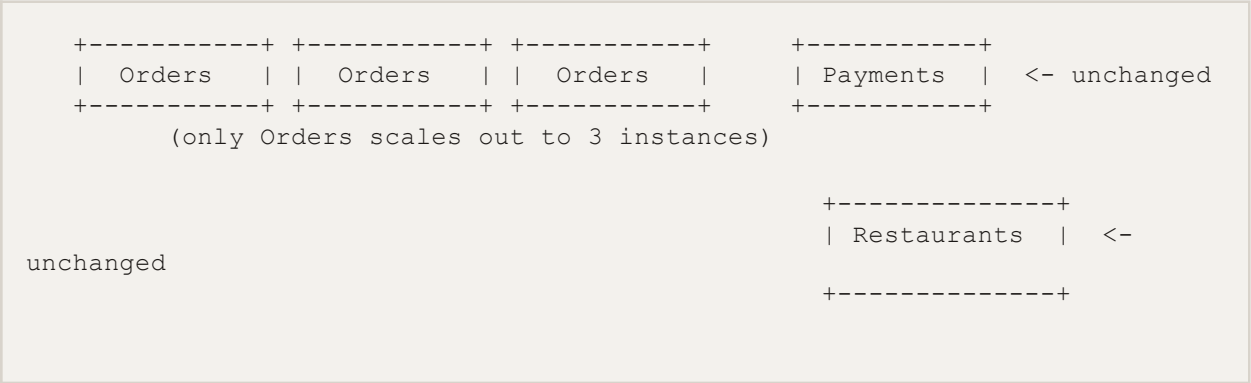
In a Monolith



Why this happens

Since all three parts run in the same process and are deployed as one unit, there is no way to add capacity to just Orders. Scaling means running another full copy of the entire application — Payments and Restaurants get scaled along for the ride, even though neither needed it. This wastes compute and makes capacity planning harder to reason about.

In Microservices



> **Why this works**
>
> Because Orders is its own independently deployable service, it can be scaled out on its own — more instances, more resources — without touching Payments or Restaurants at all. This is the concrete, practical benefit that "independent scaling" refers to.

05

The Catch: Added Complexity

What splitting into services actually costs

Turning in-process function calls into network calls between separate services isn't free. This is the tradeoff the interview answer needs to name explicitly — not just "microservices are more complex," but specifically why.

```
MONOLITH:  Orders code calls Payments code directly
           (in-process function call — fast, always available)

MICROSERVICES:  Orders service calls Payments service over HTTP/gRPC
           (network call — can be slow, can time out, can fail)
```

What This Introduces

Concretely, you now have to handle

Network latency — a call that used to be near-instant now has real, variable latency added to it.
Timeouts — what happens if Payments doesn't respond in time?
Retries — should Orders retry the call, and is it safe to retry a payment?
Serialization — data has to be encoded (usually JSON) and decoded on both ends.
Service discovery — how does Orders know where Payments currently is?

06

Monitoring & Observability

Debugging one process vs debugging a distributed system

In a Monolith

Debugging is comparatively simple

One log file (or one aggregated stream from one process). A stack trace for a failed request shows the entire call path, start to finish, because it never left the process.

In Microservices

A single request can now cross many services

One user action ("place an order") might touch Orders, then Payments, then Restaurants, each running as a separate process with its own logs. Figuring out where something went wrong means correlating logs across all of them — which is why distributed tracing (tools that follow one request across every service it touches) becomes essential, not optional, once you have more than a couple of services.

The practical takeaway

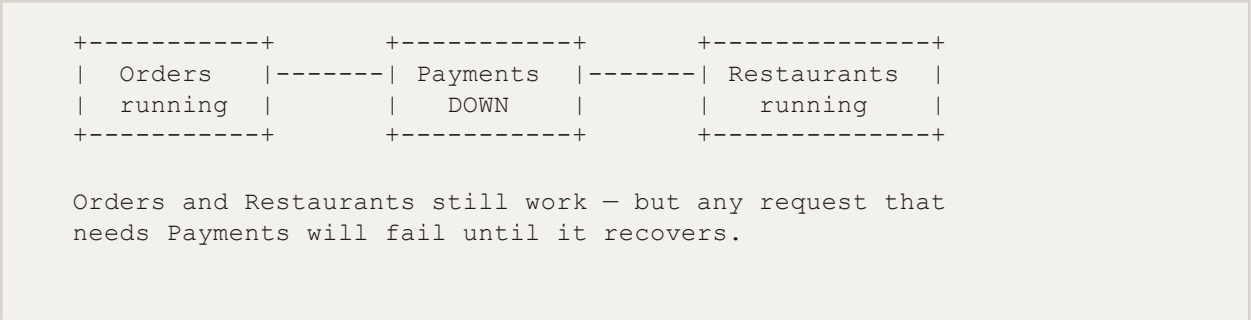
Teams adopting microservices need to invest in centralized logging and distributed tracing from early on — this tooling isn't a nice-to-have, it's what makes microservices debuggable at all once the service count grows past a handful.

07

Partial Failures

One service can go down while others keep working

In a monolith, if the process crashes, the entire application goes down — a single, all-or-nothing failure. In microservices, failure gets more nuanced: individual services can fail independently, while the rest of the system keeps running.



Handling It

Common patterns

Retries with backoff — for calls that might succeed if tried again shortly.

Circuit breakers — stop hammering a service that's already failing, and fail fast instead of piling up more load on it. Fallbacks — show cached or default data instead of failing the whole request when a non-critical service is unavailable.

Timeouts — never wait indefinitely for a service that may never respond.

Why a monolith never needs this vocabulary

None of this exists as a concept in a monolith, because there's nothing to partially fail — the whole process is either up or down. This entire category of engineering work is something microservices introduces, not something it removes.

08

Side-by-Side Comparison

Every dimension, compared directly

Aspect	Monolith	Microservices
Codebase	One, shared	Separate per service
Deployment	One unit, all together	Independent per service
Scaling	Whole app scales together	Scale only what needs it
Inter-part communication	In-process function calls	Network calls (HTTP/gRPC)
Failure mode	All-or-nothing	Partial — one service can fail alone
Debugging	One process, one log stream	Distributed tracing across services
Operational overhead	Low	Higher — more moving parts
Team structure fit	Small, single team	Multiple teams owning separate services
Best for	Early-stage products, small teams	Large systems with independent scaling needs

09

When to Use Which

Neither one is "always better"

Use a Monolith when...

- The team is small and one codebase is easy to reason about
- The product is early-stage and requirements are still changing fast
- Different parts of the system don't have wildly different scaling needs
- Operational simplicity matters more than independent deployability

Use Microservices when...

- Different parts of the system have clearly different scaling needs
- Multiple teams need to own and deploy their parts independently
- The system has grown large enough that one codebase slows everyone down
- The organization can support the added operational overhead

The point your interview answer should end on

Microservices are not a strictly "more advanced" or "better" choice — they're a tradeoff.

Many well-known systems run successfully as monoliths for years before ever needing to split. Choosing microservices before the scaling or team-structure problem actually exists just adds operational cost for no real benefit yet.

10

Common Mistakes

What teams get wrong when adopting microservices

Do This

✓ Start with a monolith unless you already know you need independent scaling

✓ Invest in distributed tracing before service count grows large

✓ Design for partial failure explicitly (timeouts, retries, fallbacks)

✓ Split services along real ownership/team boundaries, not arbitrarily

Avoid This

✗ Adopting microservices to look more sophisticated, without a scaling need

✗ Splitting services before understanding where the real boundaries are

✗ Ignoring network failure modes that don't exist in a monolith

✗ Underestimating the operational cost of running many services

11

Interview Cheat Sheet

Complete Q&A to explain Monolith vs Microservices confidently

Core Concept Questions

Q1

What is the difference between a monolith and microservices?

A monolith is a single application with all functionality in one codebase and one deployment. Microservices split that functionality into separate, independently deployable services. The key practical difference is independent scaling and deployment, at the cost of network communication overhead, more complex monitoring, and the need to handle partial failures.

Q2

Give a concrete example of why independent scaling matters.

In a food delivery app with Orders, Payments, and Restaurants, if Orders traffic spikes 20x, a monolith forces the entire application to scale together — Payments and Restaurants get scaled along even though they didn't need it. With microservices, only the Orders service scales out, leaving the others untouched.

Q3

What communication changes when you move from a monolith to microservices?

In-process function calls become network calls (typically HTTP or gRPC) between separate services. This introduces latency, the possibility of timeouts, the need for retries, and serialization overhead — none of which exist when one function directly calls another within the same process.

Complexity & Failure Questions

Q4

What is a partial failure, and why doesn't it exist in a monolith?

A partial failure is when one service in a distributed system goes down while the rest keep running normally — e.g. Payments failing while Orders and Restaurants stay up. A monolith runs as a single process, so it's either entirely up or entirely down; there's no 'partial' state possible.

Q5

How would you handle a service that's temporarily unavailable?

Common patterns include retries with backoff for transient failures, circuit breakers to stop overwhelming a struggling service, timeouts so a caller never waits indefinitely, and fallbacks — like showing cached data — for non-critical dependencies.

Q6 **Why is debugging harder in microservices?**

A single user request can cross multiple services, each with its own process and its own logs. Reconstructing what happened means correlating logs across all of them, which is why distributed tracing tools become necessary — in a monolith, one stack trace shows the entire request path already.

Decision & Scenario Questions

Q7 **Are microservices always better than a monolith?**

No. Microservices are a tradeoff, not a strict upgrade. They add real operational complexity — network calls, distributed monitoring, partial failure handling — that's only worth it once you actually have a scaling or team-structure problem that a monolith can't solve. Many systems run successfully as monoliths for years.

Q8 **A startup with a 4-person engineering team asks whether to build microservices from day one. What would you advise?**

Start with a monolith. A small team doesn't have the independent-team-ownership problem microservices solves, and the added operational overhead — service discovery, distributed tracing, network failure handling — would slow them down without a corresponding benefit at that scale.

Q9 **How do you decide where to split services when moving to microservices?**

Split along real ownership and scaling boundaries — parts of the system with genuinely different scaling needs, or that different teams need to own and deploy independently. Splitting arbitrarily, without a clear boundary, tends to create services that still depend heavily on each other, which brings the network overhead of microservices without the independence benefit.