# Monolith and Microservices– Understanding the Trade-Offs – Part 1

This lesson discusses the trade-offs involved when choosing between a monolith and microservices architecture

**We'll cover the following**

So far, in this chapter, we have discussed what *monolith* is, what a *microservice* is, their pros and cons, and when to pick which. This lesson will continue our discussion on the trade-offs involved when choosing between a monolith and microservices architecture to design our application.

## Fault isolation#

When we have a microservices architecture in place it's easy for us to isolate faults and debug them. When a glitch occurs in a particular service, we just have to fix the issue in that service without the need to scan the entire codebase to locate and fix the issue. This is also known as *fault isolation*.

Even if the service goes down due to the fault, the other services are up and running. This has a minimal impact on the system.

## Development team autonomy#

In the case of a monolith architecture, if the number of developers and the teams working on a single codebase grows beyond a certain number, it may impede the productivity and the velocity of the teams.

In this scenario, things become a little tricky to manage. First off, as the size of the codebase increases, the compile-time and test runtime increases too. This is because, in a monolith architecture, the entire codebase has to be compiled as opposed to just compiling the module we work on.

A code change made, in the codebase, by any other team has a direct impact on the features we develop. It may even break the functionality of our feature. Due to this a thorough regression testing is required every time anyone pushes new code or an update to production.

Also, as the code is pushed to production, we need all the teams to stop working on the codebase until the change is pushed to production.

The code pushed by a certain team may also require approval from other teams in the organization working on the same codebase. This process is a bottleneck in the system.

On the contrary, in the case of microservices separate teams have complete ownership of their codebases. They have the complete development and deployment autonomy over their modules with separate deployment pipelines. Code management becomes easier. It becomes easier to scale individual services based on their traffic load patterns.

So, if you need to move fast, quickly launch a lot of features to the market and scale. Moving forward with microservices architecture is a good bet.

Having a microservices architecture sounds delightful but we cannot ignore the increase in the complexity in the architecture due to this. Adopting microservices has its costs.

With the microservices architecture comes the need to set up *distributed logging, monitoring, inter-service communication, service discovery, alerts, tracing, build* and *release pipelines, health checks*, and so on. You may even have to write a lot of custom tooling from scratch for yourself.

So, I think you get the idea. There are always trade-offs involved, and there is no perfect solution. We need to be crystal clear on our use case and see what architecture suits our needs best.

Let's understand this further with the help of the real-world example of a company called *Segment* that started with a monolith architecture, moved to microservices and then moved back again to the monolith architecture.

## Segment – From monolith to microservices and back again to the monolith#

Segment (https://segment.com/) is a customer data platform that originally started with a monolith and then later split it into microservices. As their business gained traction, they again decided to revert to the monolith architecture.

*Why did they do that?*

Let’s take a look.

The segment engineering team split their monolith into microservices for fault isolation and easy debugging of issues in the system.

Fault isolation with microservices helped them minimize the damage a fault caused in the system. It was confined to a certain service as opposed to impacting, even bringing down the entire system as a whole.

The original monolith architecture had low management overhead but had a single point of failure. A glitch in a certain functionality had the potential to impact the entire system.

Segment integrates data from many different data providers into their systems. As the business gained traction, they integrated more data providers into their system creating a separate microservice for every data provider. The increase in the number of microservices led to a significant increase in the complexity of their architecture, subsequently taking a toll on their productivity.

The defects with regards to microservices started increasing significantly. They had three engineers solely dedicated to getting rid of these defects to keep the system online. This operational overhead became resource-intensive and slowed down the organization immensely.

To tackle the issue, they made the decision to move back to monolith giving up on fault isolation and other nice things that the microservices architecture brought along.

They ended up with an architecture with a single code repository that they called *Centrifuge* that handled billions of messages per day delivered to multiple APIs.

Let’s dive deeper into their architecture in the next lesson.

← Back

When should you pick Microservices …

Next →

Monolith and Microservices– Understa…

✓ Mark as Completed

Report an Issue