# Why Microservices Fail: An Experience Report

Eberhard Wolff[1]

innoQ Deutschland GmbH, Berlin, Germany
eberhard.wolff@innoq.com

**Abstract**

Microservices are now widely used. However, not all projects successfully apply microservices principles. The presented overview of typical failure scenarios shows how to use microservices more effectively, and to come up with novel architecture approaches. To support future projects, approaches to mitigate the problems are shown.

## 1 Microservices: Definition and Benefits

Microservices offer technological benefits like e.g. decoupled development, decoupled scalability, and easier continuous delivery. There are also organizational benefits: If each microservice implements a separate part of the domain, this leads to decoupled modules and enables teams to work independently.

However, microservices also pose challenges. Data isn't consistent across microservices, and microservice use a lot of new technologies. Also the effort for operations is higher. Finally, the system is turned into a distributed system so it much more likely that some part of the system fails.

This paper presents an overview of typical failure scenarios. It is based on the experience of the author from reviewing and consulting several microservices projects. The goal of such reviews is usually to understand the main challenges and risks or to understand a failure.

## 2 Failures

**Technology Overkill** Microservice projects often use lots of new technologies including platforms like Kubernetes and tools for monitoring, tracing and logging. Often new programming languages and frameworks are also introduced. It is hard to get all of these complex technologies to work.

However, not all technologies need to be introduced at once. Just like microservices should be introduce stepwise so should the technologies.

While this is a quite common pattern, it does not necessarily make the project fail. However, still excessive effort is spent on matters that do not really help the project.

**No Independent Teams** A good microservices architecture allows teams to make independent decisions about technologies and domain architecture. Often the policies and culture in the organization still require decisions by people higher up in the hierarchy. In that case, an important benefit of microservices is not realized.

So teams must be empowered to make their own decisions. That often means to rethink the organization and how it treats its employees.

This problem is quite common because the focus of microservices seems to be on the architecture while in fact independent development also needs a different approach to the organization.

**Flaky System** With a large number of microservices and a complex infrastructure, it is much more likely that some part of the microservices architecture fails. If that part causes other parts to fail, the system will easily become unavailable.

So microservices must be resilient. This means that the system can tolerate some unavailable microservices. There are some technological measure to enable resilience but often the domain logic also has to handle failures of some microservices.

Usually this problem is well understood. With modern frameworks and technologies like services meshes a technical solution is easily available.

**Synchronous Calls** Synchronous call are easy to understand because they are just like method calls in calls. However, if the called microservice is not available, this problem must be handled. Otherwise the system becomes *flaky* (see above). Also performance suffers because calls through the network are quite slow and have a high latency. So often synchronous calls are the actual root cause of other problems.

The solution is asynchronous calls. However, that changes not just the communication protocol but also the architecture fundamentally.

There are many successful synchronous systems so this problem is common but its impact is probably not too high. Models like REST encourage synchronous communication. Also programmer are used to synchronous control flow and find it easier to design synchronous systems.

**Common Data Model** A common data model implements business objects such as "customer" or "item" once for all microservices to use. This leads to strong coupling and in some cases it means all microservices must be deployed together. Microservices architectures that rely on a common event model and event sourcing often suffer from such problems.

Domain-driven Design proposes data models for a bounded context as a solution to this problem.

While the relation between bounded context and microservices are usually well understood, the definition of a bounded context and how to achieve true independence are harder to grasp. Sometimes a common data model happens by accident. If it is implemented in a shared artifact, it might even make it impossible to deploy the microservices separately. In that case the system has become a deployment monolith.

**Entity Microservice** An entity microservice provides access to a type of business objects. Entity microservices implement a *common data model* (see above). Usually access to entity services is *synchronous* (see above). When an entity services fails, it is hard to make clients resilient so the system is *flaky* (see above).

The solution is to split the system into bounded contexts by functionalities and provide each of them with a domain model including all entities.

This problem is usually easy to spot in an architecture diagram and can be avoided after a review. It is often the result of apply object-oriented design to a distributed system.

**Unfit Operations** Compared to a deployment monolith there are many more microservices that operations must deploy and operate. The operation approaches are usually not a good fit for such a high number of systems.

Because operations is such a challenge, this problem can make it impossible to use microservices. However, besides organizational remedies there are also public and private cloud offerings that simplify dealing with the multitude of microservices.

If operations is a separate unit and the development team cannot really manage the team, then this problem might make it impossible to succeed with microservices.

**Bad Structure** Microservices are just a different type of modules. So a badly structured system that as implemented with microservices are not better changeable. Actually it might be harder to change them. In a badly structured system, a change will not be contained in one module. So several modules must be changed. However, with a deployment monolith all modules are deployed together. With microservices, deployment of the microservices needs to be coordinated and decoupled. So it is harder to get the change into production.

This misunderstanding is quite common. It can result from a migration that keeps the structure of the system and just implements the modules differently.

# 3 Conclusion and Outlook

Most of the problems of microservices are now well understood and this presentation shows approaches to solve or mitigate them. It is still important to realize that microservices represent a trade-off. Microservices should only be used if the benefits outweigh the problems in the concrete scenario.

Most of the challenges can be solved. However, the solution is not just technical but also on the organizational level and on the architectural level, in particular the domain architecture. It is therefore important to focus on these fundamental issues and not just introduce microservices and hope that will solve most problems, which unfortunately a lot of organizations are doing.

The problems presented in this paper and their priority is based on personal experience. So further research about how common these problems are and what the impact would be very beneficial.