11 Reasons Why You Are Going To Fail With Microservices

Shekhar Gulati Follow
Mar 2 · 12 min read ★

Over the last couple of years I have done architecture review of multiple product teams that are on their digital transformation journey. Most of the teams were building products following Microservices architecture. They had all the right intentions to use Microservices based architecture — faster development, better scalability, smaller independent teams, independent deployment, using the right technology for the job, etc. But, most often I found teams struggling with Microservices. They failed to leverage the benefits of Microservices to their advantage. In this post, I will share reasons why I think teams were struggling with Microservices.

For people new to Microservices I recommend reading Martin Fowler's article on Microservices. I like the Microservices architecture definition mentioned in the article.

> The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment

> machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

Reason 1: Management underestimate complexity of developing Microservices

I have worked with multiple customers who are very bullish on Microservices. For them Microservices is the silver bullet solution to all their problems. In my discussions I found that most teams and their management underestimate the complexity of Microservices development.

To develop Microservices developers need a productive local environment setup.

As services in your system start to increase it becomes difficult to run the subset of the application on a single machine. This especially happens when you are building applications with languages that consume relatively more memory like Java.

Below are the main points related to local development setup.

1. The first important aspect of local development is a good development machine. I find it hard that most organisations want to use all the latest and greatest technologies but they don't want to replace the poor Windows development machine. Developers are limited by their development machines. I have seen developers using VDI images or machines with poor configuration to build Microservices based systems. This reduces their productivity and they fail to apply themselves completely. A side effect of using a poor development machine is that developers don't get quick feedback. If you know you have to wait for minutes to run your integration test suite then you will rather not write more of them and increase your pain. Poor development machines promotes bad development practices.
2. Once you have got your developers reasonable machines to work. The next thing is to ensure all services use a build tool. You should be able to build the entire application on a new machine without much configuration. In my experience even with Microservices it helps to use a root build script that can build the entire application.
3. The next important point is to enable developers to run parts of the application on their system with ease. You should use multiple `docker-compose` files to spin up different services with all the ports and volume configured.
4. Next, if you are using a container orchestration tool like Kubernetes then you should invest in tools like Telepresence that makes it easy to debug applications in Kubernetes cluster.

If an organization doesn't understand the Microservices development complexity then team velocity will go down over time.

Reason 2: No process to update libraries and tools to the latest versions

In my reviews I found that a new platform has already become a legacy. Teams were not ensuring that dependencies are kept up to date or tools like databases are on the latest version. So, the modernisation effort that started two years back has months of technical debt today.

Many teams a few years back started using the Spring Cloud Netflix OSS project for Microservices. They are using container orchestration tools like Kubernetes but because they started with Netflix OSS they are not using all the features offered by Kubernetes. They are still using Eureka as the service discovery when Kubernetes comes inbuilt with service discovery. Also, with service mesh like Istio you can get rid of most of what Netflix OSS offers. This helps reduce complexity and move a lot of cross cutting concerns to the platform.

Another point to keep in mind is to keep dependency versions in sync for all the services. I was helping a client recently that used Spring Boot for building Microservices. They had built more than 20 Spring Boot services in the last two years. In their environment they were using Spring Boot versions ranging from 1.5 to 2.1. This means when someone sets up their machine they have to download multiple versions of Spring Boot. Also, they are missing many improvements made in Spring Boot since 1.5.

Our recommendation is that organization should create technical debt items for these upgrades in their backlog. These technical debt items should be discussed as part of the architecture board meetings and resolved periodically. In my last project we setup one week sprint every three months to update all dependencies to the latest version.

Also, teams should invest time in upgrading tools like databases, messaging queues, and caches to the latest version.

Reason 3: Use of shared services for local development

Because of the poor local development most teams start relying on a shared environment for key services. The first thing that goes out of the developer machine is the database. Most young developers do not realise that shared database based development is evil. Following are the main issues that I have seen with the shared database:

1. Team members have to set up a social contract of working to avoid the last writer wins problem. A developer can wipe out data written by another developer for their work. This way of working is painful and failure prone. Sooner or later this bites the team.

2. Developers fear experimentation as their work can impact some other team members. We all know that the better way to learn is experimentation and quick feedback. With shared database experimentation goes out of the door. We need experimentation to come up with database schema and to do tasks like performance tuning.

3. Another side effect is that it becomes difficult to test changes in isolation. Your integration tests become flaky. Thus further reducing development velocity.

4. Shared database has to be treated like a pet because you don't want it to have an inconsistent and unpredictable state. You might have a developer that wants to test the edge case when a table is empty but others need a table to have records.

5. Only shared databases have all the data that is required for the system to work. Team members over time lose the traceability of the changes so no one knows how they replicate the same set up on their machine. The only way is to take the complete database dump and work with it.

6. Difficult to work when you are not connected to the network. This typically happens when you have a long commute or you are on a flight.

Database is just one example of a shared service but it can also be messaging queue, centralized cache like Redis or any other services that a service can mutate.

The best way to solve this is to make it easy for developers to run databases on their machine (as docker container) and invest in creating SQL scripts to setup schema and initial master data. These SQL scripts should be kept in version control and maintained just like any other piece of code.

Reason 4: Lack of visibility in their version control hosting platform

I was working with a client that had more than 1000 repositories in their version control system. They were using the Gitlab version control platform. They had 5 products and each product was made up of multiple Microservices. The first question I asked them was to help us understand which all services and their respective code repositories were part of product A. Their chief architect had to spent a day figuring out all the repositories that made the product A. After spend a day still she was not sure if she has covered all the services.

The best way to solve this is to group your Microservices in some way from the start so that you always have visibility on your product ecosystem. Gitlab gives a way to create a group and then create project repositories inside it. Github does not have group functionality so you can either use topics or naming conventions to achieve it.

> I personally prefer mono repos as I find them really convenient. Most developers that I meet consider it as an anti-pattern. I agree with Dan Lua post where he mentioned following benefits of mono repo
>
> ** Simplified organization*
>
> ** Simplified dependencies*
>
> ** Tooling*
>
> ** Cross project change*

Reason 5: No clear definition of a service

Most teams are not aware of what should be considered a service. There is a lot of confusion and fud around what actually constitutes a single Microservices. Let's take an example that your application has a plugin-like architecture where you will be

integrating with multiple third-party services. Should each integration be a Microservice? I have seen multiple teams going the path of creating one Microservice per integration. This becomes unmanageable very soon as the number of integrations rises. These services typically are too small that they running them as a separate process add more overhead.

I think it is better to have few large services than to have too many small services. I will start by creating a service that models an entire department within a business organization. This is in accordance with DDD as well. I break a domain into subdomains and bounded contexts. A bounded context represents a department within a company such as finance and marketing. You might think that this could lead to large Microservices and you will be right. But, in my experience it is always easier to refactor a monolithic to Microservices than doing vice-versa. As you gain more knowledge you can move towards fine-grained microservices that represent smaller concerns. You can apply Single Responsibility Principle to understand if your Microservice is becoming too big and doing too many things. You can then break it down into smaller independent services. No service should directly talk to the database of another service. They should only communicate through the published contracts. You can read more about decompose by subdomain pattern mentioned on Microservices.io website.

I also follow the advice mentioned in the backendlore document. This advice can help limit service to service communication which is the number one reason for low performance in Microservice based systems.

> If two pieces of information are dependent on each other, they should belong to a single server. In other words, the natural boundaries for a service should be the natural boundaries of its data.

Reason 6: No clear strategy on code reuse

I was working with a client that had copied four Java files related to a specific concern in all their Java based Microservices. So, if a bug is found in that code based it needs to be applied everywhere. We all know under time pressure we will miss out applying our change to one or more services. This will waste more time and increase frustration.

It is not that development teams don't know the right thing. But, the way organisations are structured people always default to easy and error-prone way of doing things.

The right way is to use an artifact manager like Bintray or Nexus and publish dependencies there. Then, each Microservice should depend on that library. You need to build tooling so that when a new version of library is published all Microservices should be updated and redeployed.

Using Microservices does not mean you should not use the best practices that have worked for us so far. You need to invest in tooling to make it easy to upgrade Microservices so that a human does not have to do this.

> Using Microservices without proper tooling and automation is a recipe for disaster.

Reason 7: Polyglot programming

I find teams using multiple programming languages, multiple databases, multiple caches in the name of best tool for the job. This all works in the initial phase of the project but as your product is in production these choices start to show their real colour. Reasons like we were building Java Spring Boot applications but we realised Java consumes more memory and performance was poor so we decided to switch to Node.js. In my last assignment, I explained to the team that their reasoning is weak.

1. **Node.js has better performance than Java.** Node.js typically performs better if you have workload that is IO based. Java beats node.js on any compute intensive workload. You can have better performance with Java for IO workloads by using reactive paradigm. Spring Boot Reactor performs equivalent to Node.js for IO workloads.
2. **Node.js consume less memory than Java.** This is partially true as Node.js application typically use less memory than Java. Java Spring Boot applications are not as bad as most people think. I ran a load test on one of the Spring Boot Java Microservice and memory consumption remained less than 1 GB. You can optimise Java memory utilisation by OpenJ9 JVM, limiting dependencies on your class path, and by tuning default JVM parameters. Also, there are new alternatives to Spring Boot in Java like Micronaut and Quarkus that consumes memory equivalent to Node.js.
3. **Node.js is more productive than Java.** This depends on the developer writing code. Java with static typing and static analysis tools can help find problems early in the development lifecycle.

Most of the time it all depends on the context. If your developers lack maturity you will develop poor products irrespective of the programming language you use.

I recommend that an organisation publishes a list of languages that teams can use. I think 2–3 is a good number. Also, list down reasons why one language should be used over the other.

There are multiple reasons you should think before you choose a language:

1. How easy is it to find mature enterprise software developers easily?
2. How easy is it to retrain developers to new technology? We found Java developers can relatively easily learn Golang.
3. How easy can developers outside the initial team can contribute, transfer, and maintain code written by others?
4. How mature is the ecosystem in terms of tooling and libraries?

This is not just limited to programming languages. This applies to databases as well. If you already have MongoDB in your system then why do you want to use ArangoDB in your ecosystem? They both are primarily document databases.

> Always think about the maintenance and operation aspect of using multiple technologies.

Reason 8: People dependency

This is not specific to Microservices but it becomes more rampant in Microservices ecosystem. The reason is that most teams are focussed on their specific services so they don't understand the complete ecosystem. In my work with different customers, I found that there are only a bunch of architects who know the overall picture. But, the problem with these architects is that they are not active in day to day activities so their influence on development is limited.

I think the best is to ensure all teams have one representative part of the architecture group so that they can align their teams with the overall architecture team roadmap and goals. To become a mature organization, you need to invest in setting up a lightweight governance.

Reason 9: Lack of documentation

Most organisations that we interacted in the last couple of years struggle with documentation. Most developers and architects either don't write documentation or the documentation they write is not useful. Even if they want to write they don't know how they should document their architecture.

At minimum we should document following:

1. Design documents
2. Context and container diagrams in C4 model
3. Keeping track of key architecture decisions in the form of architecture decision records
4. Developer onboarding guide

I recommend all the documentation be maintained in a version control system.

Reason 10: Feature over platform maturity

I have briefly touched upon this reason in other points but I think it deserves to be mentioned as a top level reason. Microservices are more complex than your traditional monolithic application because you are building a distributed system with many moving parts. Most developers do not yet comprehend different failure modes of the system. Most Microservices are built with a happy path in mind. So, if your management only wants to focus on features sooner than later you are going to fail. Features built on a weak platform fail to deliver value.

Organisations need to get into the platform mentality. Platform mentality does not only mean using containers and Kubernetes. They are part of the solution but not the complete solution in themselves. You need to think about distributed tracing, observability, chaos testing, function calls vs network calls, secure service to service

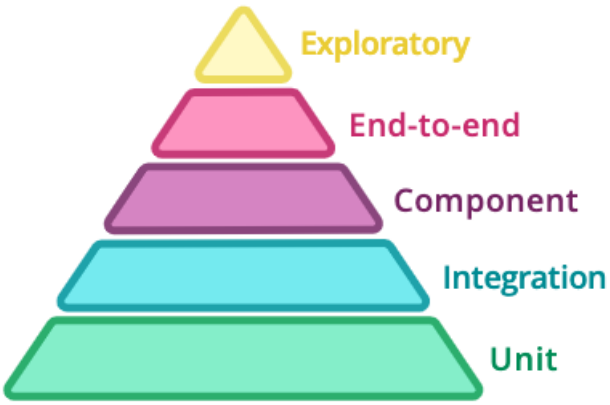
communication, debuggability, etc. This requires serious effort and investment in building the right platform and tooling team.

> *If you are a startup with limited resources my suggestion will be to rethink your Microservices strategy. Please understand what you are getting into.*

Reason 11: Lack of automated testing

Most teams know how critical automation testing is in the overall quality of the product but still they fail to do it. Microservice architectures provide more options for where and how to test. If you don't do thorough automation testing then you are going to fail badly.

I will not write much on this point as it has been covered by many other people on the web. The below diagram that I took from the Microservices testing article published on Martin Fowler website talks about the test pyramid of Microservices based systems.



Microservices Test Pyramid

Thanks to CNCF Padwan and Rahul Sharma.

Microservices Software Engineering Software Architecture