Microservices trade-offs

Daniel Barrientos Iglesias – UO270162

Raúl Núñez García – UO265205

Software Architecture, April 2021

The current mainstream discourse

The general public view on microservices seems to put them as a preferred default solution for building applications. The reasons for that are among others the modularity provided by microservices, the independent deployment of its components, which assures reliability and flexibility, and the diverse set of options and technologies available for every part of the architecture (from languages to plugins and libraries).

On the other hand, monoliths are widely seen as a thing of the past that only traditional software companies still use, and they are considered an option with too many strings attached, too cumbersome for quickly moving teams and agile methodologies and overall, not flexible enough to adapt to the newer times.

As we will see, some of those notions are not entirely accurate on the side of microservices, and they are not always a panacea. In fact, some of the typical arguments for microservices are considered widespread fallacies. In this document we are going to present the faults and benefits provided by microservices, as well as with the reasons for choosing (or not choosing) microservices over monolithic systems as the architecture for building applications, and the options provided by new hybrid models on the rise.

Next up, we are going to go deeper and comment on the main fallacies and misconceptions, starting by the scalability aspect of software solutions.

Fallacy 1 – Scalability:

The first fallacy is quite widespread, and it is that microservices are needed to tackle scaling issues. This is not entirely accurate, since with several servers and a monolithic approach we could serve a huge number of users, especially when following good practices and with the right approach. Elements like a load balancer and good design can be very helpful to scale the platforms. The corner case where microservices edge out a monolithic traditional approach would be for companies where the growth has to be very big in a small window of time, which are mainly the hyperscaler companies mentioned before. Therefore, we can surely say that we don’t need microservices to satisfy regular Enterprise scalability demands.

Fallacy 2 - Simplicity:

Another common misconception is that solutions become simpler with microservices. That might be the case when compared to monoliths that have millions of lines and very large codebases, especially if they are not maintained properly.

But the truth is that the essential complexity persists, since the application or platform to deliver stays the same, so it is impossible, in the end, to drastically reduce that essential complexity by only performing an architectural transformation. Also, we can face complex problems with microservices such as crash failures, omission failures, timing failures, response failures or byzantine failures, some of which can be avoided when using a monolithic and centralized approach.

Fallacy 3 - Reusability:

A different concept that is sometimes used in favor of microservices is that they help with reusability, with parts of different applications, especially modules, becoming suitable for multiple projects, and therefore being more affordable in the long run. While the idea is partially true, since the modularity allows for reapplying, it is important to note that extensive reusability of components is a direct threat to availability because it might introduce cascading failures as one component is used in several parts.

Fallacy 4 – Team autonomy:

Another widespread belief connected with the loose nature of microservices is that they help teams become more independent, since they are not depending on a centralized architecture. Nevertheless, the change to a new architecture doesn't solve the team coordination and management problems, since those are rooted deeper on the company's structure and working methods. Microservices might in fact be helpful in allowing more liberty to the teams and helping them work at their own pace, but not without the correct environment, company policies, structure and hierarchy.

Fallacy 5 – Improved design:

It is believed that microservices lead to better solutions in terms of design. This reasoning is based mainly on the fact that monoliths may easily end up being big balls of mud, with a lot of code that is not structured and modularized enough and, in the end, hard to maintain. On the contrary, it is supposed that microservices enforce their boundaries due to their nature, and they are easily replaceable or changed without having as many strings attached.

This reasoning not entirely accurate, and the reality is that that the source code can be organized and structured independently from the chosen runtime structure. There can also exist well-structured monoliths, it depends on the principles followed when developing the systems.

At the end of the day, the choice of an architecture is not going to improve the quality of code if the good practices and correct approaches are not present at the base of the work performed.

Fallacy 6 – Easier migrations:

Finally, microservices are hailed as the perfect solution for experimenting with different technologies, since they apparently are very flexible, and they allow for easy changes. And although it's true that migration might be better if we perform it service by service instead of migrating the whole application at once as it has to be done with monoliths, migrating a whole application that is composed of many microservices will be as expensive as migrating a whole monolith; Problems in migration come from:

- Broken IT governance processes
- Application dependencies on the OS level

And those two issues don't depend on the architecture, so there are no differences between the monolithic approach and microservices with respect to them.

Reasons to use microservices

The first reason is if you need to move fast in a highly competitive and Dynamic market where speed makes the difference between success and demise.

The second reason is if you have to satisfy very disparate NFR, this is, having to cope with the union of the needs of different customers or groups of customers that want different things.

When to use microservices?

Well, as always in software architecture… it depends.

It depends on how you intend to use them; how large the organization is and if you are capable and willing to pay for those.

1. **The Good:**
   Microservices seem to be a Good choice whenever you work in short time cycles (this is, using agile methodologies) so you might take advantage of users' feedback and respond to it releasing new features. They are also a very good option if we have tons of requests to satisfy and high availability requirements, since it would be very difficult to satisfy all the requests at the same time with a monolithic approach.
2. **The Bad:**
   If you are not willing or able to do all the other things like changing the governance, the design style, the automation, the observability, the rigor of the implementation… you rather should not go for microservices because you would face the challenges of microservices without being able to leverage their benefits. In such a setting better find a simpler architectural style that suits your needs.
3. **The Ugly:**
   Microservice architecture does not offer any advantage if we work with single teams. When developing, we want to ensure team autonomy so that software artifacts do not cross team boundaries. If we have a single team, we don't have this autonomy problem so there exist simpler architectures to apply that don't have the complexity of microservices (which we are not going to take advantage of).

Alternatives

If you want to benefit from the quickly-response of the market demands and tackle the uncertainty (gain speed), you could start with DevOps instead of directly changing to microservices; As it accelerates the IT value chain without compromising quality and forces to continuously rethink the whole IT (governance, architecture…).

Alternative 1: Moduliths

A deployment module is probably the simplest runtime architecture possible.

Monoliths tend to deteriorate over time so the maintenance become a nightmare, so we could structure them correctly, obtaining well defined modules. Then we can talk about a modulith.

Moduliths are a good idea if we don't have to move fast, and we do not have any special runtime requirements in terms of very disparate Non-Functional Requirements

Moduliths' Constraints

1. Decent skills are needed to find the right module boundaries and contents, decide which functional concept to implement where and how to expose it to the outside
2. Everyone must adhere to the rules, not to violate module encapsulation, no matter if accidentally or deliberately. We should access modules only through their official interfaces

These two properties are not usually seen in most companies, ending up in a big-ball-of-mud type monolith

3. To correctly perform technology updates and have a good design, we should take care of the module coupling. The lower the coupling, the easier to update the system becomes. These updates could suppose a huge problem in monoliths.
4. The need to trace code or debug across module boundaries is symptom of a poor interface documentation. We should not read or trace the code; it should be enough with the contract and the documentation to know how to use it and what to expect from it.

Alternative 2: Microliths

Microliths are basically services designed using the independent module design principles, but that avoid calls between modules/services while processing an external request. That's why we often need some mechanism between services that allow propagating changes affecting multiple microliths. They might be a feasible option if we meet the preconditions for using microservices, but we are not willing or able to pay for them.

They are a good idea if you do not need to move fast and/or do not have multiple teams but – opposed to the modulith setting – have special runtime requirements in terms of very disparate NFRs.

Microliths' constraints

The thing with microliths is that they require the usage of some status and data reconciliation technique; For that we must satisfy 3 things:

1. All functionality needed for a use case must be implemented inside a single microlith
2. All data required to serve an external request must be in its database
3. Reusable functionality needs to be provided via libraries or some other implementation or build time modularization mechanism, not via other services

These constraints free us from dealing with fails in services calls.

As we could jump into services for fashion reasons without fulfilling the required preconditions, microliths mitigate the most harmful microservices effects by adding these and other few constraints.

References and resources used

Along with the resources provided by the lecturer, these are the references used in the creation of this document:

https://microservices.io/

https://www.martinfowler.com/articles/microservices.html

https://en.wikipedia.org/wiki/Microservices#Criticism_and_concerns

http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

https://martinfowler.com/bliki/MicroservicePremium.html

http://www.laputan.org/mud/

https://www.thoughtworks.com/insights/articles/demystifying-conways-law