Understanding Microservices

A Backgrounder for Enterprise Architects

Marco Palladino

Understanding Microservices

A Backgrounder for Enterprise Architects

Marco Palladino

This eBook compares a monolithic vs microservices architectural approach to application development. It dives into the benefits and challenges of microservices and helps you determine whether a transition to microservices would be right for your organization.

Content

Introduction

Microservices is a hot trend in the technology sector. Pioneered by the likes of Netflix, Google, and Twitter, enterprises have started to ask themselves whether adopting a microservices-based architecture is right for them.

In a recent survey, 36% of CIOs said that they had already adopted microservices, with another 26% stating that they are in the tire-kicking phase. Given this rapid adoption you might expect there to be a solid understanding of microservices best practices, but there remains a lot of misinformation about what microservices are, and whether this architecture will be a good fit for your enterprise.

After all, it is simple for a startup to adopt microservices out of the gate, without the burden of legacy applications. However, for the larger enterprise with a sprawling array of different applications, each with their own history and purpose, it can be extremely daunting to start the process of moving from a monolithic architecture to a microservices based architecture.

In this eBook we will take a look at these different architectures, and present a basic education aimed at enterprise architects to help navigate the new world of microservices.

The Rise of Microservices

A decade ago Netflix, Amazon, Twitter, Google, and eBay (to name just a few) were early pioneers of the microservices architecture trend. Since then, microservices have become the preferred choice of developers for building cutting edge applications at hyper-growth companies.

As we will see, microservices have their pros and cons, but they have been adopted widely by development teams across multiple industries because they offer the following promises:

1. Agility: Componentization and distributed functionality enable application developers to iterate and deploy continuously, independent of other business units and application teams.

2. Freedom of Choice: Developers can autonomously choose their preferred frameworks which results in their being able to build and deploy functionality more quickly.

3. Resiliency: Microservices are designed for failure with redundancy and isolation in mind, which in turn makes applications more robust.

4. Efficiency: There can be significant savings for the enterprise that decouples functionality and adopts microservices.

In order to better understand whether or not a microservices-based approach to development and deployment is optimal, let's look at the differences between the more traditional monolithic architectural approach and microservices.

Monolithic vs Microservices

Monolithic and microservice oriented architectures are significantly different ways of building applications and products, each one with its own pros and cons. Until recently, with the rise of cloud applications, a monolithic architecture was the only way to build applications.

First of all monolithic architectures are easy to understand: there aren't many moving parts and everything is usually included in one codebase. That's great when developing a new application from scratch, especially when the codebase and the team working on it are both relatively small. It is also a fast way to develop a product and get it into market quickly, as there are no other dependencies to think about.

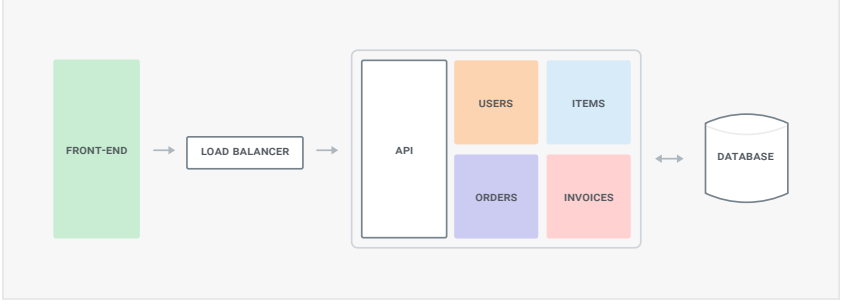


Figure 1 shows a typical monolithic application architecture

Most of the features are being delivered by the same codebase which usually communicates with the same database for storing stateful objects. In order to scale a monolithic application you introduce a frontend load-balancer which is responsible for distributing traffic to each replica of the monolithic application.

It was also common for the frontend content to be generated server-side as well until it became obvious that such an architecture would put way too much load on the backend servers, which in turn would make it harder to scale. Those were in the days when applications started to implement some functionality client-side, introducing ad-hoc backend APIs that could be consumed by the clients in order to move some of these workloads away from the servers.

It wasn't that long ago when Ajax and JSON became the de-facto standard for these client-server communications, especially for browser clients: JSON, unlike XML, is way easier to parse in Javascript, easier to generate server-side and more human-readable.

With the increasing adoption of clients, a way to onboard external partners and/or to build a developer platform was needed. APIs became the accepted way of building software with the "API first" way of thinking promoted by frameworks like Ruby on Rails.

What happens when the codebase grows?

If the codebase is small, then it is safe to say that any challenges arising from this architectural choice are also small. However, as the codebase grows, problems start to emerge.

Monolithic applications are a great way to start off a new application and test it in the market, because there are a few moving parts and the initial team is likely to be a small one.Iterations are relatively fast since with simple blue-green deployment developers can ship improvements to the end users continuously.

But when the codebase grows developing software becomes increasingly challenging. Even small changes require complete redeployments of the entire application, and if something goes wrong it can often affect the entire application.

In this scenario, what used to be an advantage becomes also a disadvantage: the business logic is bundled together and developers cannot properly isolate and compartmentalize the changes as the codebase gets bigger and bigger.

A growing codebase presents its own challenges: keeping clean code abstractions, documented code and good practices can help reduce tech debt over time, but good code management is continuously pressured by a growing team. New team members joining the team, trying to learn and push their own code to production can crack the foundations of monolithic systems over time and frustrate the management because features become getting increasingly hard to build, or seem to fail as soon as they are put in production.

The root cause of these problems is that it is very hard to keep a large codebase organized and clean enough over time, and that there is no way to guarantee code and business logic isolation, or compartmentalized feature versioning, while having different team members simultaneously contributing to different areas of the same codebase which ultimately ends up being deployed all at once in our environments.

Microservices

Therefore, it is easy to understand that a solution to these challenges is to have smaller applications, rather than a single large application.

The requirements are also clear: these components will need to be able to be built independently, will need to be deployed independently and most importantly each team member that works on a component shouldn't necessarily have to know the nuances of other components, since any implementation detail will be taken care by the specific team working on that specific feature.

Because each component is oblivious to the implementation of other components, there needs to be a way to work with interfaces, or APIs. And since developers need to be able to route and version requests, to deploy and scale them independently, these interfaces will be consumed over a network. This is when "components" become "services", and because each one of them will be doing one specific thing very well they therefore tend to become much smaller and this is why they are called "microservices".

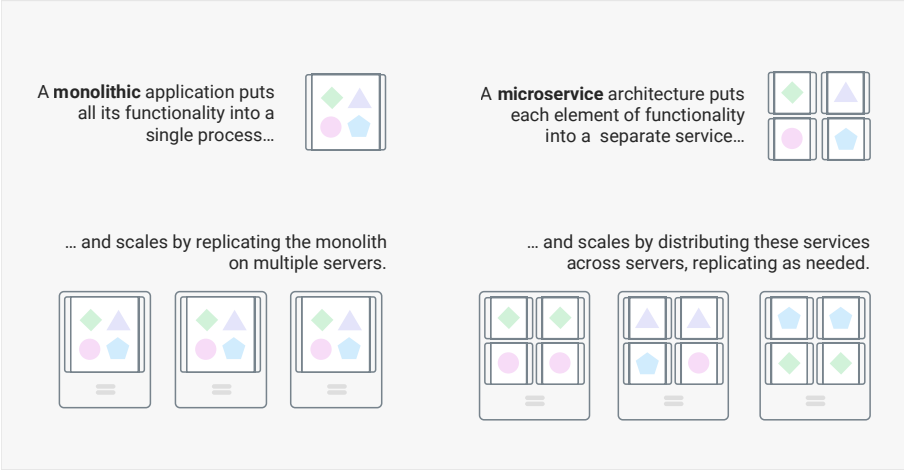


Figure 2 shows the differences between monolithic and microservice architectures

Pros and cons of a monolithic approach

As we have seen, monolithic and microservice architectures are two fundamentally different ways of designing and building our software, each one with its own pros and cons. In this section we will summarize these advantages and disadvantages.

Let's start by taking a look at monolithic architectures. The biggest benefit is having a few moving parts, which for small codebases also means dealing with more simplicity. Most commonly the application runs behind a load-balancer, and everytime a developer makes a change they are updating the main codebase and will then re-deploy the entire application. Scaling the application can be done horizontally by adding more nodes behind the load-balancer.

Simplicity, for small codebases	Faster early development speed	Easy testing	IDE support
Not ideal for growing codebases	Slower iterations in the long term	Harder to innovate	Steep code learning curve

Teams can effectively iterate and deploy updates rapidly. They can decide to deploy one or more times a day no matter what at fixed schedules - as the team keeps iterating and improving the codebase they can continuously deliver those changes to the end-user with blue-green deployments or canary releases.

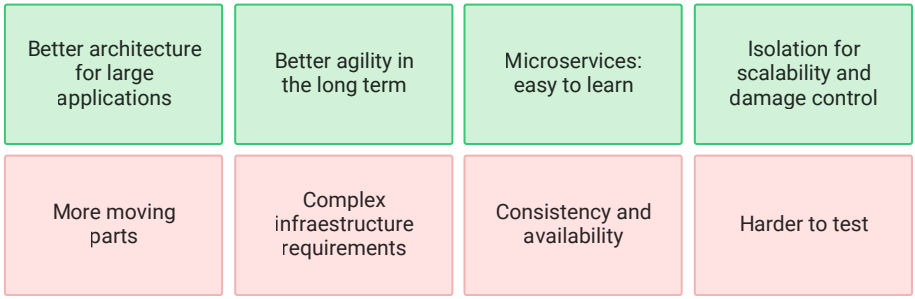
Having a few moving parts makes it easy to implement integration tests in a monolithic application because everything is in the same place and the team doesn't need to run many dependencies to fully test features - most likely a database is the only dependency. Traditionally IDE support has been built for monolithic applications, so for example in Eclipse a developer can run and test the entire application in one click.

All of these benefits disappear quickly as the application grows more complex. A growing codebase is harder to deal with, especially for new hires, since we now have to understand the big picture all at once making the learning curve steeper than it should be. Even when the code is being properly managed (with good engineering practices in place and a good overall engineering culture), deploying a large code base becomes slower and slower since we have to make sure that everything holds together and double check that our changes are not unexpectedly affecting other areas of the codebase. Experimenting with new technologies such as new databases or languages becomes almost impossible since developers cannot properly isolate the experiments and risk affecting the entire application. Because of these reasons monolithic applications are harder and harder to innovate against as they grow in size. Even a small change to the codebase requires a full redeployment of the entire application, and that's also true if you build libraries to decouple the codebase - every change to the library will still require a full deployment.

On a separate, but related note, negative factors in building or shipping a product eventually also translate to frustration for both the management and the engineering team, as release cycles become longer and the rate of innovation slows. The overall experience for the team is that any new change or features becomes exponentially harder to implement and release. Team morale and momentum is a key factor in developing a great product, and if the architectural choices start to negatively affect how the team works and ships the software that should be a red flag worth investigating.

Pros and cons of Microservices

The true spirit of microservices oriented architectures is to delegate different areas of the business logic to separate services that can be created, deployed and scaled independently.



When building an application it is possible to identify different areas or boundaries that deal with separate business logic concerns. For example in an e-commerce product there are features that deal with order management, invoice management, user management, inventory and so on. In a microservice oriented application an architect may start decoupling each one of these functions as a separate service so that, for example, updating the invoice generation business logic only requires re-deploying that specific area of the application without impacting all the other functions.

By having different services for each one of these functions teams can now decide independently when to deploy and scale their own services. These services will be communicating with each other via an interface, like an HTTP/RPC API, ignoring the actual implementation details. As long as the interface can be consumed from another service, the team can build microservices in different languages, leveraging different datastores, and experiment, within the context of a service, with new technologies such as a new database, a new framework or a new language.

Microservices therefore should also enable a more gradual learning curve for new hires since, as the name implies, they should be relatively small and dedicated to performing a few tasks. Last, but not least, microservices have built in isolation and scalability built into the architecture. If one service goes down the other services should still be up and running without bringing down the entire application. If a service suddenly experiences an increase or decrease in requests, it can be scaled up or down horizontally without impacting the other services.

Likewise, if one service is compromised, it can be isolated and taken offline without impacting the overall application, or having it infect the rest of the application.

In microservice oriented architectures large teams disappear in favor of smaller 'pizza-teams'. This comes from the idea that you can feed that team with a single large pizza. This tends to lead to a team size of 7-8 developers. Each team is now responsible for building, shipping and scaling the services they maintain independently from each other.

Of course, these benefits come with additional costs that architects need to be aware of. Microservices introduce a lot more moving parts that have to work together, like an orchestra, to deliver the final result. The networking requirements become a lot more significant since the team needs to make sure that all of these services are properly up and running, and monitoring becomes harder and more complicated than monitoring a single application. As data moves around each service it is also crucial to ensure consistency of states and availability of the data - usually eventually consistent implementations are taken into consideration. While it's easier to test a specific service, it becomes harder to test the full picture since all of these services have to be up and running.

Conclusion

In this eBook we have looked at the differences in monolithic and microservice architectures with the goal of helping architects understand if a microservice-based architecture is right for their application. We have also looked at the advantages and disadvantages of both approaches. Next we recommend checking out our eBook "Blowing up the Monolith" which is available for download here.

Konghq.com

Kong Inc.
contact@konghq.com

251 Post St, 2nd Floor
San Francisco, CA 94108
USA