```
@InProceedings{10.1007/978-3-319-74433-9_3,
  author="Kalske, Miika and M{\"a}kitalo, Niko and Mikkonen, Tommi",
  editor="Garrig{\'o}s, Irene and Wimmer, Manuel",
  title="Challenges When Moving from Monolith to Microservice Architecture",
  booktitle="Current Trends in Web Engineering",
  year="2018",
  publisher="Springer International Publishing",
  address="Cham",
  pages="32--47",
  isbn="978-3-319-74433-9"
}
```

Challenges When Moving from Monolith to Microservice Architecture

Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen

University of Helsinki,
Department of Computer Science,
Helsinki, FINLAND
{miika.kalske,niko.makitalo,tommi.mikkonen}@helsinki.fi

Abstract. One of the more recent avenues towards more flexible installations and execution is the transition from monolithic architecture to microservice architecture. In such architecture, where microservices can be more liberally updated, relocated, and replaced, building liquid software also becomes simpler, as adaptation and deployment of code is easier than when using a monolithic architecture where almost everything is connected. In this paper, we study this type of transition. The objective is to identify the reasons why the companies decide to make such transition, and identify the challenges that companies may face during this transition. Our method is a survey based on different publications and case studies conducted about these architectural transitions from monolithic architecture to microservices. Our findings reveal that typical reasons moving towards microservice architecture are complexity, scalability and code ownership. The challenges, on the other hand, can be separated to architectural challenges and organizational challenges. The conclusion is that when a software company grows big enough in size and starts facing problems regarding the size of the codebase, that is when microservices can be a good way to handle the complexity and size. Even though the transition provides its own challenges, these challenges can be easier to solve than the challenges that monolithic architecture presents to company.

1 Introduction

One of the more recent avenues towards more flexible installations and execution is the transition from monolithic architecture to microservice architecture. The motivation for this transition comes from the fact that constantly maintaining a monolithic architecture has resulted in difficulties in keeping up in pace with new development approaches such as DevOps, calling for deployment several times a day. In contrast, microservices offer a more flexible option, where individual services comply with the single responsibility principle (SRP) [1], and they can therefore be scaled and deployed independently [2]. Although partially overlooked in a recent paper addressing liquid software design space [3], such architecture clearly supports building liquid software, as more liberally updated, relocated, and replaced than their traditional, usually monolithic counterparts.

In this paper, we study the reasons why the companies decide to make the transition from monolithic architectures to microservices, and identify the challenges that companies may face during this transition. The study is based on different publications and case studies conducted about these architectural transitions from monolithic architecture to microservices.

The rest of the paper is structured as follows. Section 2 discusses the background of the paper. Section 3 compares monolithic and microservice based architectures from several different viewpoints. Section 4 introduces challenges encountered in the transition from a monolithic architecture to microservices. Towards the end of the paper, Section 5 draws some final conclusions.

2 Background and Motivation

Microservices are small services that comply with the single responsibility principle (SRP) [1]. Each service is focused only on one functionality. This kind of approach makes it clear where the boundaries between different services are and where code changes should go. Consequently, microservices are by nature loosely coupled [4]. Loose coupling gives developers a chance to make independent changes to services without affecting the rest of the codebase. Because microservices are not tied to each other, they can be scaled and deployed independently [2]. Such architecture is also a key enabler for building liquid software, as more liberally updated, relocated, and replaced than their traditional, usually monolithic counterparts.

All these qualities make microservices desirable option for existing monolithic applications. Scaling of monolithic application is always harder than scaling microservices because one has to scale the whole application and deploy the whole codebase instead of scaling the part of the application that demands more resources [6]. The current development of cloud services make automatic scaling of resources very easy and cost-efficient. Microservices make most out of this automatic scaling. When developing and deploying a big monolithic applications companies cannot take the full advantage of these functionalities.

As applications grow in size during the many years of development, it becomes harder to maintain and make changes to them [5]. It is possible to maintain and develop the monolithic software but eventually it becomes obvious that changes to the architecture of the whole application has to be made. This kind of trend was first noticed in companies which have a lot of traffic, many developers and large codebase. Companies such as Amazon[7], Netflix[8], LinkedIn[9], SoundCloud[10] and many more have made the transition to microservice architecture because their existing monolithic application was too hard to maintain, develop and scale.

Monolithic applications have their downsides when the application codebase grows big and the changes have to be made rapidly. Fine-grained scaling is also impossible with monolith, because the whole application needs to be deployed every time. However, when teams start developing new a application, there are business requirements to develop new features fast in the beginning so the company can survive. Monolithic applications make it simple to develop, deploy, test and scale application when the size of the codebase is relatively small [11]. Most of the applications have monolithic architecture because of these reasons. Monolithic approach is enough in the beginning and it is possible that the size of the codebase and need for fine-grained scaling is never needed. Which means that it is better to stay with the monolith and avoid the technical and organizational challenges that microservice architecture comes with. There are also differing opinions. It can be argued that the refactoring of the existing monolith is too demanding task and instead the organization should spend more time at the start of the process to evaluate the architecture and functionalities that are required [12]. It is much easier to introduce accidental tight coupling in a monolithic than in microservices. Breaking up these tight couplings can be hard and require a lot of time and understanding of the application.

Whether the decision is to start with microservices or monolith that later will be refactored towards microservices, there are multiple technical challenges that needs to be solved in order to use microservices. Microservices add more distribution which adds more points of failure. This brings up many questions such as how to handle failures, how services communicate between each other, how transactions are handled and so on [1]. If monolithic application for some reason stops running in production, it is very fast to recognize that because nothing is working. With microservices, if one service stops responding other services still work and these kind of error situations needs to be handled properly. Communication between microservices is one of the big questions to get right. Getting the communication wrong can lead to a situation where microservices lose their autonomy and thus the main benefits of the whole approach can diminish [1]. On top of that, the communication between multiple microservices can introduce performance issues if the services are too fine-grained [34]. It seems to be an agreed consensus that the complexity should be within services instead of messaging pipes [2]. One challenge is also to handle the orchestration of the microservices in production. Luckily in the last few years many new tools for supporting this have been made such as Kubernetes [32] and Mesos [33] to name two.

In most of the cases, the need for architectural change from monolithic to microservices is realized when the codebase and the size of company has grown big. In these cases, there are new refactoring and organizational challenges on top of the existing ones that come with the microservice architecture. Refactoring of the existing software can be a daunting task. In order to successfully make the refactoring, a good test coverage is required. Otherwise, there is a chance that during the introduction of microservices new bugs might end up in the existing functionalities. It might also be hard to find and define which parts of the existing software should be split up to microservices and what are good candidates for microservices. One of the methods is to find seams from the existing software [1]. A seam is a part of the code that can be isolated and work alone in separation from the rest of the codebase [13]. Finding the seams requires good knowledge about the business use cases. However, this knowledge should already be in the organization. Either in the codebase, if the monolith has good modular architecture or even if the modules are not well defined then at least the use cases should be pretty clear.

On top of the technical challenges, adopting microservice architecture requires organizational changes [1]. The organization needs to adapt accordingly to the new architecture. Every team needs to take ownership of the services or service that they own. This means developing, testing, deploying and taking care of the service in production. DevOps culture needs to be adopted as teams now need to deploy, monitor and address issues also in production [14]. Teams have to set up their own continuous integration (CI) and continuous delivery (CD) pipelines. Microservices also require the composition of the teams to change. It is very typical that in organizations where monolithic applications are built, that the developers, quality assurance and operations work as separate teams. An organization which uses microservices, these horizontal borders needs to be broken and teams should be vertical in the sense that every team should consist of people from development, quality assurance and operations. Melvin Conway's paper *How Do Committees Invent* states the following: "Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure." [15]. In other words, organizations that build microservices also need to adapt their communication structure to this new style of architecture. Otherwise, the conflict between organization structure and the structure of architecture design will cause problems.

3 Comparison of monolithic and microservice architecture

Monolithic architecture is the standard way to start application development because it is more straightforward. A monolith application is developed and deployed as a single unit containing all the needed parts. A typical monolith application consists of UI-layer, business logic layer and data access layer which communicates with the database as can be seen from the Figure 1 on page 5. Monolithic architecture is a good way to start development because it makes the initial development faster than with microservices [16].

However, as the codebase grows in size, the problems of monolithic architecture increase [11]. The new features and modifications of old features are harder to implement because the developer has to find the correct place to apply these changes. It takes a long time to get familiar with the big codebase. This means that it takes time for new developers to get up to speed as they feel lost and cannot find the correct place to apply changes. With big monolith codebase refactoring changes can reflect many parts of the software. This might lead to that developers fear to make big refactoring tasks, because their changes effect numerous places and testing that everything still works is a big task. This can even result in situations where refactoring is ignored because it is too risky. Which will lead to code that is not clean. Because developers are not familiar with the codebase, it is very likely that the code duplication level raises as it is almost impossible to find existing code which would already do the same thing. Also, it is very likely that the modularity of the codebase goes down as the codebase grows, as there are no hard module boundaries [11].

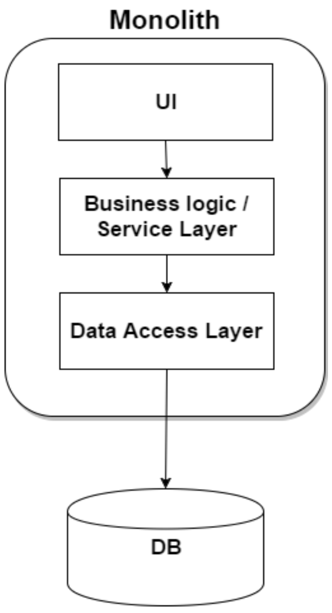

Fig. 1. Typical monolithic application consisting of three layers.

There are also other reasons to split up the monolith than just the size of the codebase. One of the reasons is team structure [1]. If the teams are located in different geographical areas and their communication is very slow, it makes sense that they take ownership of different parts of the code. This eases up the development as now the teams in different geographical areas do not have conflicts about modifying parts of the software. The team that has the ownership of a service can decide what happens inside that service and the other teams only have to care about the interface of the service. With this kind of approach the communication doesn't have to be as fast and as fine-grained as before. Also, if a team has ownership of a service it is more likely that the code stays cleaner and technical issues are solved faster as there is no one else to blame about the quality of the code. This division also means that if the team wants to publish a new version of the service to production it is a lot faster and easier, because the team doesn't have to communicate and coordinate the changes with other teams so much.

In general, it can be said that if the following requirements apply to the organization, then monolithic approach might be the correct choice: the number of the teams is low, the codebase is relatively small and it will stay that way for the coming years, the teams are close to each other geographically and their communication is easy. On top of these, the complexity of the domain plays a big factor and it is not easy to say when to use monolithic approach instead of microservices or vice versa. It is possible to have a monolithic that has good modularity and clean code but it requires more work to keep that modularity when working with monolith instead of microservices as microservices provide the modularity in their nature and make it harder to break the modularity [16].

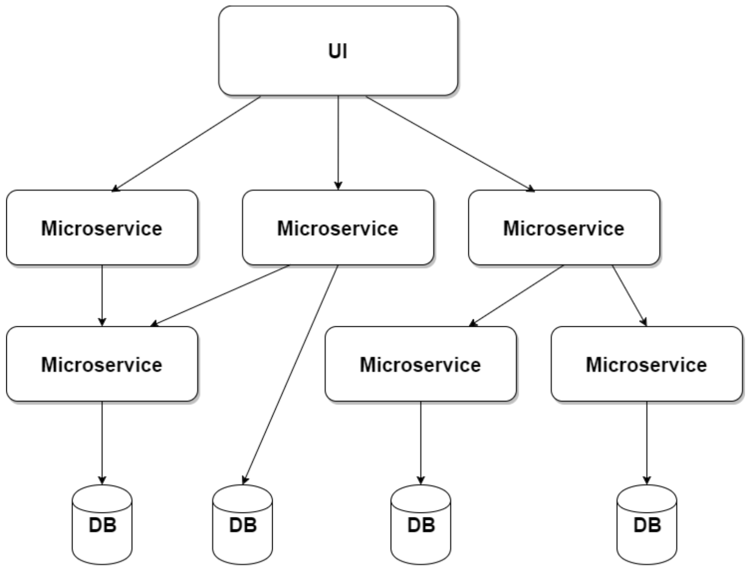

Fig. 2. Example of microservice architecture.

A typical monolithic application can be seen in the Figure 1. A layered architecture with three layers is very common especially in enterprise applications. As we can see, there is usually only one database which means that if some of the data would require better scaling, this kind of architecture does not support it. This limits the choices that the teams can make to support the business requirements and scaling requirements. For example, most of the data would fit fine in a relational database management system, but some parts of the application data require greater scalability and performance which for example Cassandra would provide. One database also means that schema changes have to be coordinated between multiple teams which slows the development.

In Figure 2 on page 6 we can see that with microservices it is possible to select the database engine per microservice. This kind of pattern is called database per service [17]. This gives teams more freedom to select their tools. Designing and scaling of the database are easier when the database consists of fewer tables and the microservice has full control of the data and the schema of the database. Some of the microservices can even be without database, if they for example write to disk. From Figure 2 we can also see that microservices usually communicate with each other and the UI can request data from multiple services.

Table 1 contains comparison of these two architecture styles. As we can see, both of them have pros and cons. As a conclusion about the table we can notice that microservice architecture style becomes attractive when we are working with big codebase. With smaller projects the technical challenges that microservices bring out might not have enough time to pay back. Also, if the DevOps skills of the team are lacking then it might be better to stick with monolith in the beginning.

Category	Monolith	Microservices
Time to market	Fast in the beginning, slower later as codebase grows.	Slower in the beginning because of the technical challenges that microservices have. Faster later
Refactoring	Hard to do, as changes can affect multiple places.	Easier and safe because changes are contained inside the microservice.
Deployment	The whole monolith has to be deployed always.	Can be deployed in small parts, only one service at a time.
Coding language	Hard to change. As codebase is large. Requires big rewriting.	Language and tools can be selected per service. Services are small so changing is easy.
Scaling	Scaling means deploying the whole monolith.	Scaling can be done per service.
DevOps skills	Doesn't require much as the number of technologies is limited.	Multiple different technologies a lot of DevOps skills required.
Understandability	Hard to understand as complexity is high. A lot of moving parts.	Easy to understand as codebase is strictly modular and services use SRP.
Performance	No communicational overhead. Technology stack might not support performance.	Communication adds overhead. Possible performance gains because of technology choices.

Table 1. Comparing monolith and microservices

4 Challenges in adopting microservice architecture

The challenges with adopting microservice architecture can be divided in two parts; the technical challenges and the organizational challenges [1]. Both of these are equally important to get right. The challenges are a bit different whether the application development will be started from scratch compared to converting a big existing codebase from monolith to microservices. In this paper, we focus on the challenges that are related to refactoring existing monolithic application towards microservice architecture. Most of these challenges needs to be addressed also when starting with microservices from the beginning. The biggest differences are that, there is no need for a big organizational change and refactoring is not needed, but the selection of services and their business requirements might be harder, if the teams decide to start with microservice architecture from beginning.

4.1 Technical challenges

There are various technical challenges that needs to be solved before it is possible to utilize microservice architecture. When the starting point is a monolithic application, the organization is then most likely familiar with the domain already and have an idea where the seams of application can be found [1]. The biggest problem in these cases is to separate these services. It can take a lot of time and effort to refactor the services out from the monolithic architecture. This is why the refactoring towards microservices should be done in small parts. Also, when implementing new functionalities, they should not be appended to monolith even though it might be faster. Instead, organizations should expand their microservices offering and add new microservices to replace the old monolithic code [18]. By applying this mechanism organization is slowly moving most of the codebase towards microservices. It is extremely important to be careful when doing this refactoring, because there is the possibility of introducing new bugs in existing features. This is why good test coverage is needed before starting this process.

Testing can be seen as an enabler to the whole refactoring project. If most of the testing is done manually, then it might be good idea to first get the automatic test coverage up and postpone the refactoring towards microservices. Good automatic test coverage helps refactoring, and it also gives the option to get more out of the microservices. Microservices can be released frequently only if it is possible to validate that the software does what it needs to do [1]. There are multiple automatic testing strategies which developers can apply to their application depending on its needs. Continuous integration and continuous delivery both go hand in hand with microservices [19]. Without these two practices, it comes very hard to handle the multiple services, their deployments and validating the actions of the service.

After the automatic test coverage is in place it comes feasible to start thinking about the other challenges. The first thing to do is to define the microservices and their responsibility areas. It is important that the decomposition of services is correct [1]. This is important, because it is expensive to make a lot of changes across the services. Instead it is easy to change functionality inside one service, but when the changes effect multiple services and their interfaces, then the task becomes harder and more time consuming. This is where the earlier experience working with monolith and designing its components is helpful as developers should already have a good understanding about the business concepts of the application. It is probably best to start from the easiest and most obvious services and when the organization has more knowledge about microservice architecture then the services can become more fine-grained. Existing microservices can be split up to smaller services, when there is better understanding about the service composition.

When splitting up the services, attention should be paid to the fact that the services do not become too fine-grained. Microservices can introduce a performance overhead especially if the communication is done over network [1]. For example, if the communication is done using REST over HTTP each inter-service call adds overhead from the network latency and from marshalling and unmarshalling the data. If the services are too fine-grained there will be a lot of traffic between them and as each call adds overhead the outcome can be a system that does not perform well enough.

One of the biggest challenges is the integration between different microservices [2]. It is not recommended to tie the integration between services to some specific technology, because the teams might want to use different programming languages when implementing services. Instead using a technology which does not require a specific programming language is better choice. There are also multiple other challenges when thinking about the integration of microservices. The interface of microservice should be simple to use and it should have good backwards compatibility so when new functionalities are introduced, the clients using the service do not have to be necessarily updated. Like every good interface, it should also hide the implementation details inside.

Using microservices in production environment provides new challenges that needs to be handled. There can be hundreds of different services running in production and many services can have multiple instances running to comply with the scale that is needed from the application. This big amount of microservices organizations run in production means that there has to be tools to automatically to deploy, scale and manage these services. Manual deployment process is not an option when deployments to the production environment are made multiple times per day. Docker and similar technologies enable easier development and deployment of microservices [30]. Docker makes microservices easily portable and isolated [31]. There are no conflicts of dependencies or the need to configure each environment. With Docker developers can easily imitate the production environment in their local development environment. If the decision is to use Docker in production environment, then there are multiple tools to handle the scaling, deployment and management of these containers. These tools such as Kubernetes make it easier when solving these challenges [32]. Kubernetes provides multiple features like horizontal scaling, service discovery, load balancing and so on.

In addition to the infrastructure challenges, there are also challenges like logging and monitoring which need more attention with microservices than when working with a monolithic application [30]. In case of failures, there needs to be good logging in microservices. This logging has to be easily searchable and all services should aggregate logs to one place so problem finding becomes easier. When there is only one monolithic software running in production, it is a lot easier to monitor that. The monolithic application might be scaled to multiple nodes, but still there is less nodes or containers to monitor than when running microservices in production. This means that when there is more to monitor, there should also be good automated tools which notify the persons who need to act when a microservice fails. Because there are more moving parts it becomes more likely that a service will go down or have other problems such as high latency. Users might not notice that one microservice is down and it might seem that everything is working properly. When running monolith in production the users will immediately notice that the whole service is not working.

When the organization has more than a few microservices, then it should also take in to consideration the possibility that a service might not respond [20]. The design of microservices has to be fault-tolerant. With distributed system that has a lot of services it is inevitable that at some point a service might be under heavy load and cannot respond in timely manner or the service might down. This where circuit breaker pattern becomes useful. Circuit breaker pattern handles failures fast and can provide fallback which returns default data instead of waiting for the response from a dependency [20]. Circuit breaker monitors for failures and when there are enough failures the subsequent calls to the dependency won't be made and instead an error is returned [22]. This means that instead of adding more load to the dependency by making new calls to it, an error is returned immediately to the user which gives the dependency time to recover from the load. Also, a fallback method can be provided if it is possible. For example, when a product service fails to fetch personalized product recommendations, it could fallback to recommendations that are tied to that product as a default or instead just return nothing as recommendations and the UI could then handle this case. This kind of approach means that the user might not even notice that the microservice serving the recommendations is down. There are multiple ready solutions which can be used in microservices. The most famous one is probably Hystrix [21]. Hystrix is a library that provides latency and fault tolerance to distributed systems. Using Hystrix is simple and makes it easy for developers to make their calls to dependencies latency and fault tolerant.

Data management is an important part of every application. There are many important questions such as whether to use relational database or NoSQL, what database provider suits best for the use cases of the application and which kind of schema the database should have. Microservices provide the freedom to use multiple database engines. This pattern, which is called database per service, comes with its own challenges [17]. Multiple different databases means that managing them is harder and the organization might not have that much of knowledge about the database. Previously if the monolith application used traditional rela-

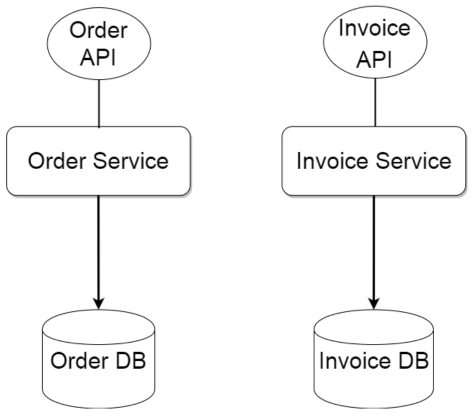

Fig. 3. A database per service.

tional database then using ACID (Atomicity, Consistency, Isolation, Durability) transactions was easy. Now when there are multiple services which each have their own database the transactions are harder to handle and more time needs to be spent dealing with the transactions. Instead of transactions microservices can agree on eventual consistency of data [23]. This means that the changes done by other services might not be persisted immediately, but they will be eventually persisted, when the service has processed the message. If previously the user had to wait for the whole transaction to complete now the user might not be able to immediately explore the data that the dependency service will create. Figure 3 illustrates the one database per service approach. In this case, the service owns the data and if for example the order service needs to know something about the invoices it has to go through invoice API. This leads to loose coupling of the services. If the team managing invoice service has to modify invoice database schema they can do it without changing any other service than invoice service as long as the API stays the same.

It is also possible to use one single database for all the services [24]. This approach is illustrated in Figure 4. One database for all services is however problematic as now the database schema is tightly coupled [23]. One database also mean that services have access to data that should only be available through calls to other services. This can result in loss of modularity as it is very easy to rather query the data from different table directly instead of making a service call to proper service which should return this data. In Figure 4 we can see that the order service has access also to the schema of invoices. This makes it now easy for order service developers to get data from invoices without going through the invoice API. Which leads to tight coupling during development time, if team developing invoice service wants to change the schema they now have to coordinate this effort with multiple other teams [24]. Shared database diminishes many good sides of microservices and use of one shared database is not recommended [1]. Instead, when refactoring towards microservice architecture

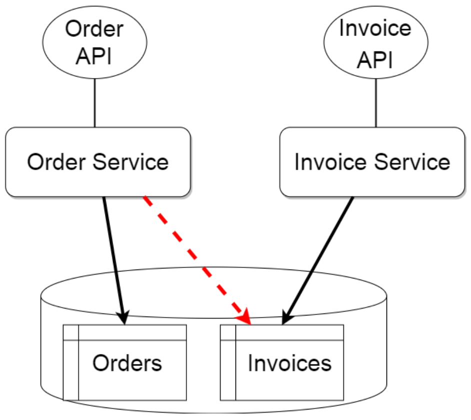

Fig. 4. One shared database for all services.

also the monolith database should be split up to multiple databases which can be accessed only by the service that handles that business context.

4.2 Organizational challenges

Besides the technical challenges that microservices provide, there are organizational challenges that needs to be addressed when moving from monolithic architecture to microservice architecture. Even if the organization solves all the technical challenges, the structure and skills of organization should also support the new architecture [1].

One of the organizational challenges is the structure of the organization. In order to develop good application, the organization must align their structure with the structure of the application architecture [25]. If previously with monolithic application the organization had big teams which had clear roles like quality assurance, development and database administration then this kind of organization structure does not work with microservices. Conway's law states that the organization which designs the system will produce a system which structure is a copy of the organizations structure [15]. If the structure of the organization is monolithic then microservices approach does not work. The organization must split these big teams to smaller teams which can work autonomously. This way the structure of the architecture is in line with structure of the organization and they do not conflict with each other.

Figure 5 contains a monolithic organization which consists of teams which have very specialized focus areas. The teams are very good in their specialized areas but when delivering a business functionality, they need to collaborate with each other and there is a hand-off process before a release can be made [26]. This kind of structure results in slower development cycles. Figure 6 shows an organization that is structured around microservices and products. When teams are organized like this they have more autonomy considering their releases. Now

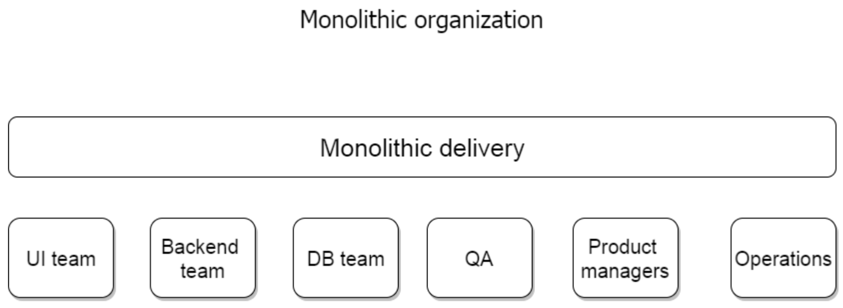

Fig. 5. Monolithic organization which delivers monolithic application.

there is no hand-off process to a third party and teams do not have to wait for other teams to complete their changes. If there are new business requirements for invoicing then the team that is responsible for invoice service can deploy their changes to production when they are ready. This kind of structure reduces the need for fine-grained communication as teams can work separately. Teams have full control of their service and deployment timetable which introduces ownership of the product to the teams. They will also develop product specific skills on top of their technical skills which means that for example the accounting service team will have better business capabilities considering that area.

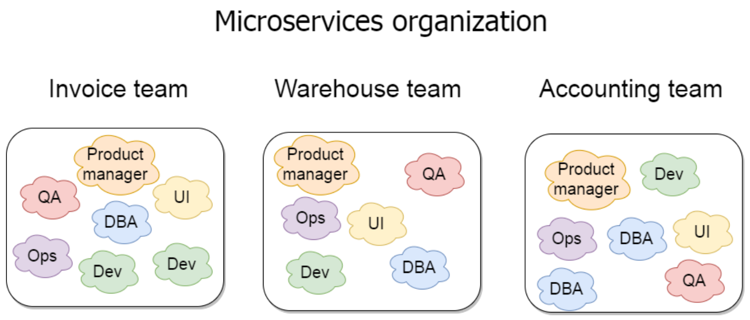

Fig. 6. Organization structure that supports microservices.

When organization adopts microservice architecture style, the teams should have more freedom and responsibility but less process [26]. This means that the teams can deploy their service to the production when they need to instead of waiting for approval from someone else. Teams own their codebase and are responsible for the functionality of the service. They no longer can blame someone else about their failures and if there are problems in the production the team that is responsible for that service has to fix it. Ownership of code brings developers pride in appearance, improvement and commits developers to long-term involvement instead of always thinking that the problems in the codebase are someone else's problems and the next person will clean up the mess [27]. The ownership also gives teams freedom to develop the service as they want. It might still be sensible to have some constraints but there are far less constraints with microservices than with monolith. This kind of change is very dramatic. It might be scary to suddenly be fully responsible of the code, if before the architectural refactoring people had the chance to hide from responsibility. It takes time that teams adopt to this new responsibility and see the good sides of it. A good middle ground would be to have an operations team in the beginning which still has the main responsibility of production so teams have time to adopt that they own the codebase [1]. When teams are comfortable with owning the codebase then they can gradually also take over the production responsibility of their service.

When teams take full responsibility of their service, they might require new skills in order to deploy and fix problems in production. This means adopting DevOps mentality. DevOps can be described as "a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality" [28]. Since fast change cycle is one of the main points of microservices, the deployments need to be fast and smooth. This kind of deployment process is called continuous delivery. It aims to shorten the release cycle of application by making developers and operations work together [29]. In a monolithic organization developers just committed code and then the deployments were responsibility of operations team. Now every team needs to be able to handle deployments. There might not be enough people with corresponding skills so team members need to learn new skills and enhance their deployment process. When operations and developers are working together in same team they have similar goals but the education and adaptation takes time and patience.

5 Conclusion

Based on the different challenges that organizations face when they move from monolith architecture to microservice architecture we can conclude that this transition is not easy and it will require a lot of time and effort from various parts of the organization. Making a system distributed introduces new challenges that needs to be addressed. Even though microservices can be considered novel software architecture, they are still somewhat mature in the sense that the tooling around microservices is pretty good and most of the challenges can be solved by applying open sourced tools made by companies which have already made the transition from monolith to microservices. These tools do not however solve the problem of refactoring and removing the tight coupling of codebase. Most of the focus will of course be on the technical side of the challenges, but organizations should not forget the Conway's law. The structure of the organization has to be similar as their architecture. So, the technical and organizational challenges have to be both solved in order to be successful with microservices.

Refactoring to microservices is a big process which can take a long time and this process requires buy-in from every part of the organization. This transition is still doable, as previous examples have showed us. Organization that is considering this transition should however evaluate the cost and the reward of the transition and think about their own problem base. Microservice architecture is not a silver bullet that works for every organization and in some cases the challenges outweigh the rewards. However, for some organizations it is the only way to continue the rapid development and deliver software fast to their customers.

Acknowledgements

The research was supported by the Academy of Finland (project 295913).

References

1. Sam Newman, *Building Microservices, Designing Fine-Grained Systems*, 1st ed. United States of America: O'Reilly Media Inc., 2015.
2. James Lewis, Martin Fowler, (2014, March) Microservices a definition of this new term [Online] https://martinfowler.com/articles/microservices.html
3. Andrea Gallidabino, Cesare Pautasso, Ville Ilvonen, Tommi Mikkonen, Kari Systä, Jari-Pekka Voutilainen, and Antero Taivalsaari. On the Architecture of Liquid Software: Technology Alternatives and Design Space. In Proceedings of the 2016 13th Working IEEE/IFIP Conference on Software Architecture, 122–127, 2016, IEEE.
4. Chris Richardson, (2014, March) Microservices — Pattern: Microservice Architecture [Online] http://microservices.io/patterns/microservices.html
5. Johannes Thones, *Microservices* IEEE Software, 32, no. 1, pp. 116-116, 2015.
6. M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas and S. Gil, *Evaluating the monolithic and the microservice architecture pattern to deploy Web applications in the cloud* In Proc. of CCC 2015, pp. 583-590
7. Chris Munns, (2015, October) I Love APIs 2015: Microservices at Amazon [Online] https://www.slideshare.net/apigee/i-love-apis-2015-microservices-at-amazon-54487258
8. Tony Mauro, (2015, February) Nginx — Adopting Microservices at Neflix: Lessons for Architectural Design [Online] https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/
9. Seven Ihde, (2015, March) InfoQ: From a Monotlith to Microservices + REST: the Evolution of LinkedIn's Service Architecture [Online] https://www.infoq.com/presentations/linkedin-microservices-urn
10. Phil Calcado, (2014, June) SoundCloud: Building Products at SoundCloud - Part 1: Dealing with the Monolith [Online] https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith
11. Chris Richardson, (2017, March) Microservices — Pattern: Monolithic Architecture [Online] http://microservices.io/patterns/monolithic.html
12. Stefan Tilkov, (2015, June) Don't start with a monolith when your goal is a microservice architecture [Online] https://www.martinfowler.com/articles/dont-start-monolith.html

13. Michael Feathers, *Working Effectively with Legacy Code*, Prentice-Hall
14. A. Balalaie, A. Heydarnoori, P. Jamshidi, *Microservices Architecture Enables Devops*, IEEE Software, vol 33, no. 3, 2016, pp. 42-52.
15. Conway, M. E. (1968). How do committees invent. Datamation, 14(4), 28-31.
16. Martin Fowler, (May, 2015) Microservice Premium [Online] https://martinfowler.com/bliki/MicroservicePremium.html
17. Chris Richardson, (March, 2016) Microservices — Pattern: Database per service [Online] http://microservices.io/patterns/data/database-per-service.html
18. Josh Clemm, (July, 2015) A Brief History of Scaling LinkedIn [Online] https://engineering.linkedin.com/architecture/brief-history-scaling-linkedin
19. Balalaie A., Heydarnoori A., Jamshidi P. (2016) Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. In: Celesti A., Leitner P. (eds) Advances in Service-Oriented and Cloud Computing. ESOCC Workshops 2015. Communications in Computer and Information Science, vol 567. Springer, Cham
20. Montesi, F., Weber, J. (2016). Circuit Breakers, Discovery, and API Gateways in Microservices. arXiv preprint arXiv:1609.05830.
21. Netflix Inc., (2013) [Online] https://github.com/Netflix/hystrix
22. Martin Fowler, (March, 2014) [Online] https://martinfowler.com/bliki/CircuitBreaker.html
23. Hasselbring, W. (2016, March). Microservices for scalability: keynote talk abstract. In Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (pp. 133-134). ACM.
24. Chris Richardson, (November, 2015) [Online] http://microservices.io/patterns/data/shared-database.html
25. Sam Newman, (June, 2014) [Online] https://www.thoughtworks.com/insights/blog/demystifying-conways-law
26. Tony Mauro, (March 2015) [Online] https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/
27. Nordberg, M. E. (2003). Managing code ownership. IEEE software, 20(2), 26-33.
28. L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2015
29. Wettinger J., Andrikopoulos V., Leymann F. (2015) Enabling DevOps Collaboration and Continuous Delivery Using Diverse Application Environments. In: Debruyne C. et al. (eds) On the Move to Meaningful Internet Systems: OTM 2015 Conferences. Lecture Notes in Computer Science, vol 9415. Springer, Cham
30. Stubbs, J., Moreira, W., Dooley, R. (2015, June). Distributed systems of microservices using docker and serfnode. In Science Gateways (IWSG), 2015 7th International Workshop on (pp. 34-39). IEEE.
31. Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014(239), 2.
32. Kubernetes (March, 2017) [Online] https://kubernetes.io/
33. Mesos (May, 2017) [Online] http://mesos.apache.org/
34. Mark Richards, Microservices Antipatterns and Pitfalls, 1st ed. United States of America: O'Reilly Media Inc., 2016.