Extracting Microservices from a Monolithic Application

Developing a Migration Strategy

Master's thesis in Computer science and engineering

EMIL AXELSSON

ERIK KARLKVIST

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019:06

Extracting Microservices from a Monolithic Application

Developing a Migration Strategy

EMIL AXELSSON

ERIK KARLKVIST

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Extracting Microservices from a Monolithic Application
Developing a Migration Strategy
EMIL AXELSSON
ERIK KARLKVIST

© EMIL AXELSSON, 2019.
© ERIK KARLKVIST, 2019.

Supervisor: Regina Hebig, Computer Science and Engineering
Examiner: Riccardo Scandariato, Computer Science and Engineering

Master's Thesis 2019:06
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Extracting Microservices from a Monolithic Application
Developing a Migration Strategy
EMIL AXELSSON
ERIK KARLKVIST
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Microservices is a relatively new way of developing backend applications. Usually in backend applications, often referred to as monolithic applications, the code is developed and deployed as a single software artifact. In microservices, this artifact is split up into multiple small applications or services, where each service can be developed and deployed independently from one another. Microservices have gained popularity since it enables targeted scaling of specific features and a more fault tolerant system, among other benefits. However, the microservice system has to be designed and implemented in such a way that they enable these benefit, which comes with several challenges.

In this thesis, an action research methodology was used to find and validate information about microservices. The research was therefore split into two phases. The first phase focused on knowledge gathering, and the second phase on validating the knowledge gathered by migrating a microservice from a monolith.

During the first phase the challenges that comes with microservices were explored together with different strategies that can be applied to combat the challenges. Information about microservices was found through reviewing literature and conducting interviews with developers who has worked with microservices. Several challenges were found, were the most dominant challenge was analyzing the domain and defining service boundaries. Another area of focus during the first phase was to find any advantages or disadvantages of being a small company when developing microservices. A migration strategy was created at the end of the phase based on the knowledge gathered. The strategy includes all necessary steps that needs to be conducted in order to successfully migrate microservices from a monolith.

In the second phase, the migration strategy was tried out and a microservice was migrated. Generally the strategy worked well and it resulted in a fully functional microservice. However, some issues were found during implementation, which resulted in some adaptations to the strategy. One of the goals for the migration strategy was that it should be generalized and usable in any context where there is a previous monolith. Whether or not the migration strategy is applicable to other monoliths has not been validated, and further research on it might be necessary.

Keywords: microservices, services, software architecture, service boundaries

Acknowledgements

This paper could not have been done without our supervisor Regina Hebig, who continuously met with us every week and gave us valuable feedback on both report structure and the content in the paper. We would also like to thank our examiner Riccardo Scandarriato for reviewing and giving us insightful feedback. Olof Hartelius, Joakim Fischer and the rest of the employees at Eliq has also been a great source of information, since they helped us with questions regarding Eliq, their domain and their current system. We also want to thank our opponents Sebastion Nilsson and Jonas Åkesson for reading through and giving us additional feedback on the paper. Finally we want to thank all the interviewees who participated in the project, your answers and perspectives on working with microservice has been of great value to us.

Emil Axelsson, Gothenburg, 06 2019

Erik Karlkvist, Gothenburg, 06 2019

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1
1.1 Background 2
1.1.1 Benefits 3
1.1.2 Challenges 3
1.1.3 Eliq 4
1.2 Purpose of the study 4

2 Method 7
2.1 Phase 1: Gathering Knowledge 7
2.1.1 Literature review 7
2.1.2 Interviews 8
2.1.3 Creating the migration strategy 9
2.2 Phase 2: Conducting the migration strategy 10
2.2.1 Analyzing the domain 10
2.2.2 Selecting what microservice to extract 11
2.2.3 Analyzing data 11
2.2.4 Designing the architecture 12
2.2.5 Selecting tools and implementing 12
2.2.6 Evaluating the microservice 12
2.2.7 Enhancing the process 13

3 Phase 1: Gathering Knowledge 15
3.1 Result from literature review 15
3.1.1 Service boundaries 15
3.1.2 Automation in integration and deployment 16
3.1.3 Communication between services 17
3.1.4 Decentralized data 18
3.1.5 Fault tolerance and fault handling 19
3.2 Results from interviews 20
3.2.1 Working processes 20
3.2.1.1 Microservices developed from scratch 20
3.2.1.2 Microservices extracted from a monolith 21
3.2.1.3 Reflections 21
3.2.2 Challenges 22
3.2.2.1 Understanding the domain 22
3.2.2.2 Understanding the current monolith 23
3.2.2.3 Selecting microservice to develop 23
3.2.2.4 Architecture and design 24
3.2.2.5 Technical challenges 24
3.2.2.6 Costs 25

3.2.3 Small companies 26
3.3 Discussion 26
3.3.1 Understanding the domain and defining service boundaries 27
3.3.2 Managing communication and decentralized data 27
3.3.3 Selecting microservice to extract 28
3.3.4 Managing automations 29
3.3.5 Handling and tracking failures 30
3.3.6 Challenges for small companies 31
3.4 The initial migration strategy 32
3.4.1 Analyze the domain 32
3.4.2 Analyze data 33
3.4.3 Select microservice to extract 33
3.4.4 Design architecture 34
3.4.5 Select tools 35
3.4.6 Implement 35
3.4.7 Evaluate implementation 36
3.4.8 Enhance process 36

4 Phase 2: Conducting the Migration Strategy **37**
4.1 Analyzing the domain 37
4.2 Selecting microservice to extract 38
4.3 Analyzing data 39
4.4 Designing the architecture 39
4.5 Selecting tools 41
4.6 Implementing 41
4.7 Evaluating the result 42
4.7.1 Changing the requirements 42
4.7.2 Evaluating the requirements 43
4.7.3 Evaluation by Eliqs' lead developer 43
4.8 Enhancing the process 44
4.8.1 Analyze the domain 47
4.8.2 Select microservice to extract 48
4.8.3 Analyze data 48
4.8.4 Designing architecture 49
4.8.5 Selecting tools 49
4.8.6 Implement 49
4.8.7 Evaluate the result 50
4.8.8 Enhancing the process 50

5 Discussion **51**
5.1 Answering the research questions 51
5.2 Lessons learned from extracting a microservice 53
5.3 Threats to validity 55
5.4 Further Research 56

6 Conclusion **59**

A The Migration Strategy **I**

B Summary of Challenges and Strategies **VII**

List of Figures

1.1 Monolithic Architecture (left) to Microservice Architecture (right). The UI (user interface) represents some sort of application, the middle layer represents the code in modules (the symbols), and the cylinder at the bottom represents the databases. 2

3.1 The initial migration strategy based on interviews and literature created after the first phase. This strategy will change in upcoming sections. The final migration strategy can be read in its entirety in Appendix A. 32

4.1 The domain model created during the event storming session, with a few additions and changes after the workshop. Everything within the dotted line is part of the internal system, and everything outside are third party systems or similar. 38
4.2 Database schema for the microservice user communication, which includes all the data that the microservice needs to send sms, push notifications and emails. 39
4.3 The architecture for the microservice user communication. 40
4.4 The final migration strategy based on interviews and literature, and the experiences gained from phase 2. The final migration strategy can be read in its entirety in Appendix A. 46

A.1 The final migration strategy based on interviews, literature and our own experiences migrating. I

List of Tables

2.1 Data table containing information about the different companies that the interviewees has worked for. 'Company size' is in regards to how many employees work for the company and 'Team size' is how many developers that were working with the microservices when the interview was conducted. 'Developer exp.', meaning developer experience, is how much the interviewee approximated the experience of the developers in the team. 'Team locations' is in regards to whether or not the team was split on different locations or not. 'Nmbr. of MS' translates to number of microservices at the time of the interview. 'Tools' refers to what tools are used by the team to develop the microservice. 'Migration' refers to wether or not they migrated from an existing monolith or not. 9

4.1 A table summarizing all major tasks conducted during the migration, what the outcome of each task was and how it affected the migration strategy. 45
4.2 Table displaying the time, in person hours, it took to complete each task in the initial migration strategy, apart from evaluating and enhancing the process. Configure is a sub task of implement, but was displayed as its own task due to the time it took to configure different things. 47

5.1 A table displaying the major lessons learned from migrating a microservice 54

B.1 Summary of the challenges and strategies found VIII

1
Introduction

In 2011, a workshop near Venice took place where a group of software architects discussed a new form of architectural style that many of them recently had been looking into. The idea behind the architecture was to split the large code bases that companies developed into small, independent code bases that deployed on separate instances. In 2012, the group decided to name the architecture "microservices". [Fowler and Lewis, 2014]

Usually, software is developed using a monolithic architecture [Kalske et al., 2018]. This monolith is normally abstracted into packages and modules based on software design patterns, but are nevertheless executed as a single software artifact. Thus, the modules cannot execute independently, even if they are not actually dependent on each other [Dragoni et al., 2017a]. A visualization of this can be seen in at the left in Figure 1.1, where there is a user interface (UI), the business logic in the center and a database at the bottom. The symbols are separate modules, containing the code in a single monolith. If one of these modules fails, there is a risk that the entire monolith will stop working until the failure is handled properly. Similarly, if one of the modules is used more frequently, it can not be independently scaled. The entire monolith has to be duplicated on a new instance, which will scale the other modules whether they need to be scaled or not [Dragoni et al., 2017a].

Monoliths are common because they are easy to develop, test, deploy and scale when the code base is relatively small [Kalske et al., 2018]. Since they are easy to develop and intuitive at small scales, they are often what companies begin to develop. However, when the code base grows larger, issues occur. Quality attributes such as code maintenance and understandability are harder to keep at a good level. The entire monolith has to be re-deployed whenever a change is implemented, which can take a long time for larger code bases [Kalske et al., 2018] [Dragoni et al., 2017a] [Fritzsch et al., 2019].

Due to the issues with the monolithic architecture, there has recently been a trend in companies to migrate from a monolithic architecture to a microservice architecture [Dragoni et al., 2017a]. A microservice application can be defined as “a distributed application where all its modules are microservices, where a microservice is an independent and cohesive process interacting via messages” [Dragoni et al., 2017a]. In practice this means that a microservice architecture contains several small separated services that are independent from each other. They are programmed to conduct a task and to communicate the result of said task to wherever it is needed in the system.

In Figure 1.1 an example of a microservice architecture is displayed to the right. Similarly to the monolithic architecture to the left, there is a user interface, the business logic, and a database. Each of the modules in the business logic layer are independent of each other, enabling targeted scaling, better fault tolerance and understandability. For example, if one of the modules is used more than the others, that code can be duplicated across multiple instances without duplicating the other modules. It is also possible for a module to crash, without it affecting the other modules. There are many companies who have adapted this sort of architecture, for example Netflix [Mauro, 2015]

and Spotify [Goldsmith, 2015], and cloud providers such as Amazon Web Services[1] and Microsoft Azure[2] provide tools to help with developing microservices.

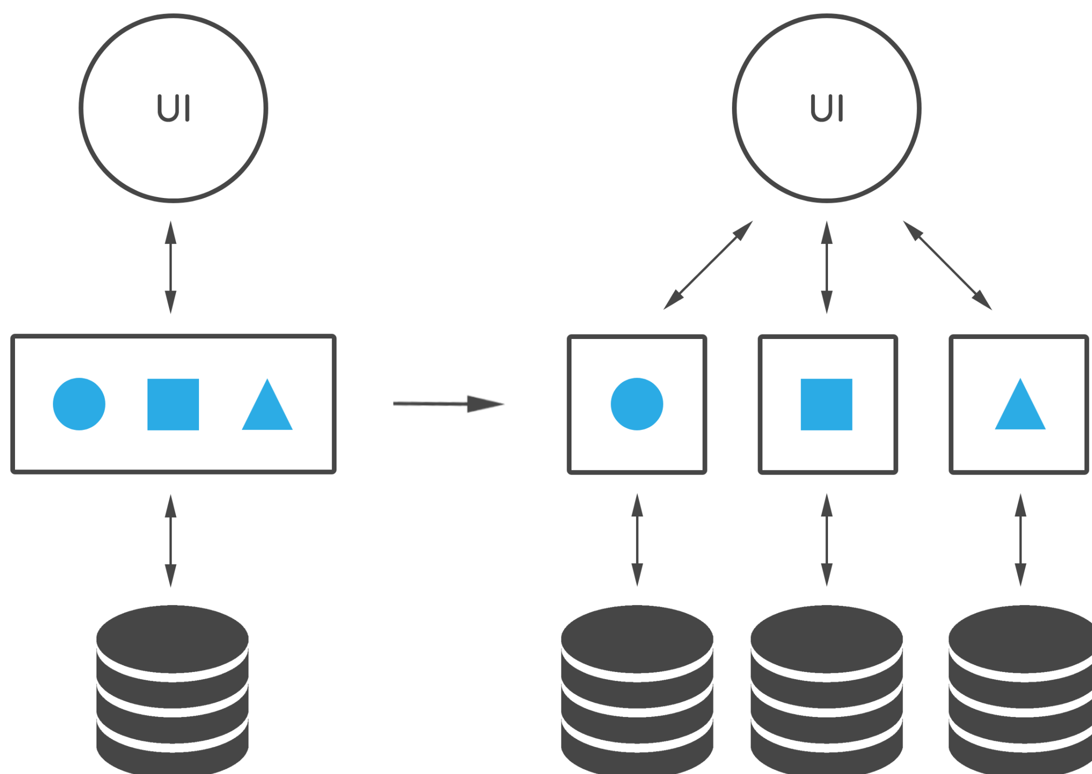

Figure 1.1: Monolithic Architecture (left) to Microservice Architecture (right). The UI (user interface) represents some sort of application, the middle layer represents the code in modules (the symbols), and the cylinder at the bottom represents the databases.

Migrating to microservices from an already existing monolith is not straightforward. There are several challenges to the architecture, such communicating between services, monitoring the services and finding a suitable decompositions of the monolith that fits microservices [Kalske et al., 2018]. This thesis aims to collect and understand the challenges and present strategies and work flows companies use to successfully migrate from a monolithic application to a microservice application.

1.1 Background

This thesis is done in collaboration with a small company in Gothenburg called Eliq[3]. They are interested in finding out how they can migrate from their current monolithic application to a microservice application, how to address eventual challenges with the migration and if there is anything particular that smaller companies should focus on during the migration.

There are papers on how enterprises and large companies uses microservices and what they have done to migrate from a monolithic architecture into microservice architecture. However, there is a lack of information in regards to how smaller companies and startups should work with migrating into microservices, and what the best practices and tools are for them. As will be explained later,

[1] Amazon Web Services: https://aws.amazon.com/
[2] Microsoft Azure: https://azure.microsoft.com/sv-se/
[3] About Eliq: https://eliq.io/about-eliq/

the challenges with microservices is often divided into two categories, technical and organizational challenges. Larger companies probably suffer more from the organizational challenges than small companies. On the other hand, developing microservices in a small company, where there is only a handful of developers and limited resources, other challenges may occur. Is it worthwhile to invest in the time and resources it takes to migrate, or are the challenges outweighing the benefits? In addition, the process of migrating is unclear. There are several challenges to combat, but in what order and what focus companies have during migration is hard to understand.

1.1.1 Benefits

There are many benefits with microservice architecture that any software company would be interested in. Scaling is one of these benefits and a big reason why companies are interested in microservices [Kalske et al., 2018]. Since microservices are independent and focuses on a single task, if that particular task needs to be scaled, only that microservice needs to be duplicated across several instances [Namiot and Sneps-Sneppe, 2014].

In a monolithic application, the entire application is deployed as a single artifact. Small changes to a large system will require a redeployment of the entire application and takes more time as the application grows. Using microservices the deployment can be easier and faster, since the microservices can be deployed independently instead of redeploying the whole application. This can reduce the time to market, make companies release features more frequently and make them react quicker to bug reports [Newman, 2015].

Microservices also allow the usage of different languages and technologies for each service. For example, python might be better when conducting a computation heavy task, while Java might fit another task better. Microservices enables these different types of cross language applications [Fowler and Lewis, 2014].

If the microservices are duplicated with low coupling in a well defined and well thought out architecture, they enable a high rate of resilience and fault-tolerance [Dragoni et al., 2017b]. This means that even though one service might crash, others might not, which in turn means that the user of an app might not even notice that some part of the application is not working. In a monolithic architecture, a server crash might cause the application to stop responding completely.

1.1.2 Challenges

The benefits of working with microservices does not come for free. There are several challenges related to working with microservices, and many of them are caused by the increased number of moving parts in the system. For example, each microservice in a microservice application is independent and manages its own data. Compared to a monolithic application, there is no centralized data store in microservices, and it can therefore be challenging to provide data consistency between them [Fowler and Lewis, 2014]. The microservices can also be deployed and replicated at different hosting locations, which adds complexity to monitoring and logging of the system [Newman, 2015]. Gathering knowledge and understanding about such issues is one of the main focuses in this thesis, and the issues mentioned are elaborated in section 3.1 together with strategies on how to handled the issues.

Microservices also introduces organizational challenges. Melvin Conway concluded in his paper that “any organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations.” [Conway, 1968] This is often referred to as Conway’s law. Considering Conway’s law, the structure of the microservice architecture will reflect the structure of the organization. This is important to consider if there are several teams that will work with the microservices, and re-organizations withing the company might be necessary to maintain a desired architecture. One way to split the teams is to

have teams responsible for certain microservices, and if they need a change in another microservice, they have to request it from the team responsible for that service [Kalske et al., 2018]. Since this thesis is done in collaboration with a small company, with around 15 employees and only three backend developers, the organizational challenges were deemed to have less of an impact and was not a focus during this thesis.

One of the most important parts when migrating from a monolith to microservices, is how to split the monolith and how to define what is a service in the monolith. It's important to have a good idea on how the system will look like in the end, because it might be hard to change afterwards. This issue is often refereed to as *defining service boundaries* [Fowler and Lewis, 2014] [Kalske et al., 2018] [Microsoft, 2018]. There are several refactoring techniques that can be used, but there are still several questions to consider. This thesis will therefor focus on identifying these questions and other issues related to the migrations, and solving them as well as possible.

1.1.3 Eliq

This thesis is done in collaboration with Eliq, which is a small company in Gothenburg. Their goal is to increase customer engagement for utility companies by providing a platform that helps the utility companies' customers to understand and manage their energy consumption. Eliq do this by retrieving and analyzing household energy data from utility companies. The analysis is later used to provide insightful information to the customers, such as predictions, comparisons to similar households and alerts of unusual consumption. Eliq is rapidly growing, both in regards to number of customers and number of employees. The number of integrations to utility companies' systems is also increasing, which adds complexity to the system. To reduce the complexity of the system and to separate integrations from one another, they have thought about changing their backend architecture from a monolithic architecture to a microservice architecture. Eliq is, however, uncertain of how a migration could be done. Their goal is to investigate whether or not they should migrate to microservices, and how that could be done in the best possible way.

1.2 Purpose of the study

The purpose of this study is to identify challenges related to migrating a monolithic application into a microservice application and identify strategies that can be used to face these challenges. In addition, investigate whether there are any differences in how to migrate in small companies, particularly in cases where there is a lack of experience, time and resources. The study aims to answer three research questions, labeled **RQ1-3**.

RQ1: What strategies have industries applied when moving from a monolithic application to a microservice application, and what are the results of these strategies?

A few papers presents recommendations of strategies that can be used to face certain challenges in regards to migrating to microservices. However, there is little to no information about how these strategies are applied in industry and if industry are using any other strategies. This research question aims to clarify that matter.

RQ2: What are the challenges small companies face when moving from a monolithic application to a microservice application?

Microservices is often discussed in the context of larger enterprises. This research question aims to understand if there is any additional challenges related to small companies when migrating to a microservice application.

RQ3: What attributes should be considered when selecting parts of the monolith to extract?

A migration to microservices is usually done in iterations by extracting part by part. This research questions aims to understand what attributes should be considered when selecting a part of the system to extract.

This study was conducted in two phases, one knowledge gathering phase and one phase where the knowledge was tried and evaluated. The details of each phase is described in Chapter 2. The first phase focused on gathering knowledge in regards to microservices by doing a literature review and interviews. The outcome from the literature review and interviews resulted in a migration strategy and can be found in Chapter 3. The migration strategy was then applied and evaluated and the result is found in Chapter 4. Finally, the research questions were answered in Chapter 5, based on the findings from the two phases, together with a conclusion in Chapter 6.

2
Method

This study was conducted using an action research methodology, which means that the research was done in phases. This research method was chosen since it allows us to gather knowledge and later validate and evaluate the knowledge by applying it in practice. In addition, we have no prior experience in regards to microservices which brings several uncertainties. Exploring the subject in phases could provide an opportunity to continuously increase the knowledge and dig deeper into the subject.

The research questions RQ1, RQ2 and RQ3 are all related to identifying challenges when migrating a monolithic application to microservices and to find strategies that can be used to reduce these challenges' impact. The goal for each phase was to find answers to these questions and to gather valuable insight that can be used for anyone interested in developing microservices, particularly in cases where there is a previous system to migrate from.

The action research was done in two phases. Initially three phases were planned, but due to time constraints only two phases were completed. Phase one, named 'Gathering Knowledge', focused mainly on understanding what microservices are and what challenges they introduce. Interviews were held during this phase to understand how it is to work with microservice from an industry perspective. The main goal of phase one was to create a migration strategy based on the information gained from literature and interviews.

The migration strategy created in phase one was tried out in phase two, called 'Conducting the Migration Strategy'. A microservice was extracted from Eliq's monolith and both the migration strategy itself and the result of the microservice was evaluated. The migration strategy was then adapted based on the experiences gained from the phase and on the results of the evaluation.

2.1 Phase 1: Gathering Knowledge

The first phase focused on gathering knowledge about microservices and identifying challenges related to migrating to, and working with, microservices. Challenges that were commonly mentioned during the literature review and interviews were assembled together with strategies to face these challenges. These challenges and strategies were thereafter used as foundations for the migration strategy.

2.1.1 Literature review

The initial step of the first phase aimed to understand the concept of microservices and why companies chooses to apply a microservice architecture in their systems. Articles from various companies and blog-posts were read. These sources were primarily used to gather insights in to why companies choose to work with microservices. Thereafter, a literature review was completed.

Research papers, books and conference papers were found using Google Scholar and Chalmers online library search engines. We used a snow balling approach in finding literature and did not do a systematic literature review. Terms like "microservices" with different endings such as "challenges", "migration" and "monolithic application" were used extensively. The literature review also included exploring the references of the papers in order to gather as much information as possible. The aim of the literature review was to find support and potential answers to our research questions RQ1, RQ2 and RQ3. The literature review also aimed to deepen our understanding of microservice in order to generate a well formed interview.

The challenges and strategies, for migrating to and working with microservices, were categorized in a structured manner. In this way, the result were more concrete and could be used when generating the interview form and mapping the answers from the interviews.

2.1.2 Interviews

The knowledge gathered from the literature review was used in the process of creating the interview form. The interview aimed to, from an industry perspective, highlight the most common challenges and understand how industry faces these challenges. Even though some of the papers present challenges and possible strategies that could be used during a migration, there was a lack of information of how industry approach migration. The interview questions also aimed to clarify uncertainties that were raised during the literature review.

In total five people were interviewed and to keep their anonymity the interviewees will be named A, B, C, D, and E. The interviewees had all been working with microservices in some way. The interviews were semi-structured, meaning that there were a mix of open and closed questions. The closed questions aimed to understand in which context the respondent had been working with microservices. Example of such question was the size of the company, the size of the development teams and what tools and technologies they have used. The data from the closed questions are summarized in the table 2.1.

The open questions aimed to let the respondent develop their thoughts and to answer follow-up questions. The open question focused on a few different areas. First we asked questions about their working processes to further understand the context, but also to be able to compare their working processes with how literature talks about developing microservices. Thereafter we asked questions about the challenges the interviewees faced when working with microservices, and how they dealt with said challenges. Finally we asked questions to understand how small companies should work with microservices and if any challenges might be more or less severe for small companies.

The interviewees were found by contacting several authors from blog posts and experience reports, as well as people from companies that we knew worked with microservices. While not every interviewee had worked with migration from a monolith to microservices, everybody had some experience with microservices, either through working with them professionally or writing papers about them. In table 2.1 data such as number of employees and number of microservices for each company is displayed. The data was gathered by asking closed questions to the interviewees as mentioned before. Interviewee E is not included in the table because they had never worked in a company using microservices. However, E had written articles and researched microservices professionally and came with valuable insights about microservices.

Company ID	Company A1	Company AD	Company B1	Company D1	Company C1	Company C2
Interviewee ID	Interviewee A	Interviewee A & D	Interviewee B	Interviewee D	Interviewee C	Interviewee C
Company Size	1200 employees	11000 employees	200 employees 60-70 developers	50 employees	40-50 employees	12 employees
Team size	3 backend 6 frontend	5-10 developers	Only 1	11 developers	Only 1	7 developers
Developer exp.	1-8 years	2-10 years	Both very experienced and beginners	0-15 years	-	10 years
Team locations	Same	2 different	2 different	Same	Same	Same
Nmbr. of MS	10	100	2	10-15	20	10
Tools	.Net, Internal host	Azure, .Net, Webapps, Servce bus, TFS	Java, Kotlin, Jenkins, Docker. Not on cloud.	Azure Webapps, Webjobs, .Net	Azure Webapps, .Net	Azure service fabric, Azure functions, .Net
Migration	Yes	No	Yes	Yes	No	No
Comments			Investigated wether to migrate or not. Decided not to implement MS		Only maintainace, another team developed	

Table 2.1: Data table containing information about the different companies that the interviewees has worked for. 'Company size' is in regards to how many employees work for the company and 'Team size' is how many developers that were working with the microservices when the interview was conducted. 'Developer exp.', meaning developer experience, is how much the interviewee approximated the experience of the developers in the team. 'Team locations' is in regards to whether or not the team was split on different locations or not. 'Nmbr. of MS' translates to number of microservices at the time of the interview. 'Tools' refers to what tools are used by the team to develop the microservice. 'Migration' refers to wether or not they migrated from an existing monolith or not.

2.1.3 Creating the migration strategy

An analysis of the result took place at the end of the phase, when the interviews had been held and the literature review was complete. The answers from the interviews were mapped to the challenges and strategies that were identified during the literature review. In this way a broad picture of working with microservices was drawn which highlights the hardest challenges and most common strategies.

This information was then used to analyze the entire process of migrating to microservices, and to find strategies to combat the challenges of working with microservices. Based on this analysis the initial version of the migration strategy was created. The migration strategy included several steps that each focused on one or more challenge when migrating. Each step also contains suggestions of how to tackle said challenges, and they are all based on the outcome from the literature review and the interviews. The steps are placed in an order based on the work flows from interviewees and literature, to make the migration as easy as possible.

2.2 Phase 2: Conducting the migration strategy

Phase two focused on conducting the migration strategy created in phase one (Section 3.4) in order to evaluate and improve it. The migration strategy was created by us and was based on the outcome from the literature and interviews in Phase 1. The rationale and description of each step can also be found in Section 3.4.

By trying out the strategy and trying to extract a microservice ourselves, we were not only be able to improve the strategy, but we also gathered further knowledge in to our research questions.

Before any step in the process had began, a survey was sent out to the developers at Eliq. The survey aimed to understand how Eliq's developers expected the microservices to work on a high level, and also to understand why they were interested in working with microservices. The data from the survey could then be used during the evaluation of the implementation to see if the microservice and infrastructure that had been built aligned with their initial vision, and to evaluate why any pivots from this vision had occurred. After that the migration strategy was conducted step by step in accordance to the what the strategy suggests, which can be viewed in Section 3.4. This section briefly describes what was done in each step of the strategy.

2.2.1 Analyzing the domain

The first step in the strategy is to analyze the domain. Almost two weeks of domain analysis took place, where the goal was to create a domain model and to understand what Eliqs goal as a company is. The goal was also to understand what the current system does and how it functions.

This task was done in two steps. First there was a meeting with one of the sales people from Eliq. He pitched Eliq to us as he would to any client, and afterwards there was some discussion about what was said and their goals were identified. This was done to get an high level understanding of Eliq's domain and product.

After that, a more thorough domain analysis was done. As explained in the migration strategy in Section 3.4, it is important to include both domain experts and developers. A four hour workshop was therefore held with three employees at Eliq, one developer, one manager and on person from sales. The aim was to gather as much information as possible regarding Eliq's domain and to visualize the domain model. During the planning of the workshop a method called "Event storming" was found. In an event storming workshop the participants analyze and discuss events in the domain and how they progress in the system. The parts of the domain that are producing events and the parts of the domain that are interested in events are analyzed and specified [Spring, 2018]. An example of domain event in an e-commerce domain could be "Order submitted", and that event could be produced by the Order part of the domain, and received by the Invoicing part of the domain.

Event storming was found to be a suitable solution in our case since it would allow us to both understand what events that exists in the system, and how the different parts in the domain are connected. The next step of the migration strategy was also to analyze the data in the domain and how that progress in the system, and this seemed like good preparation.

Our workshop began with a brainstorming session where as many domain events as possible were generated. After that, the events were placed on a wall, and a time line was created. Each event was then evaluated one by one and previous events that the current event was dependent on were placed before it on the time line. The goals was to have a big time line of all events in the system, that shows how they are dependent on each other and what the causes for each event are, and from this create a domain model.

Once the event storming workshop was completed, a domain model was created and refined in a

few short iterations with feedback from the developers at Eliq.

2.2.2 Selecting what microservice to extract

According to the migration strategy in Section 3.4, the next step after "Analyze domain" should be "Analyze data". However, as will be described in the result in Section 4.2, the task was too time consuming and we therefore switched order between the steps "Analyze data" and "Select microservice". The next step after "Analyze domain" was then "Select microservice to extract".

The selection of microservice was done in a meeting with the lead developer at Eliq. The meeting aim to decide what part of the monolith that would be a suitable start for extraction into a microservice. The selection of microservice was done based on the six questions from the migration strategy. The reasoning behind each question are explained in Section 3.3. The questions were:

1. What is the least tangled parts in the code?
2. What is easiest to extract?
3. What can be extracted to a complete microservice right now?
4. What is going to be changed a lot in a short time?
5. What feels good to extract?

2.2.3 Analyzing data

The initial idea with analyzing data was to analyze where data is located in the domain and how it traverse within the domain. However, since the order of "Analyze data" and "Select microservice to extract" changed, the purpose of "Analyze data" changed as well. Instead of analyzing data for the entire domain, it was decided to analyze what data that was needed by the selected service and what data updates it was interested of. This step therefore turned into a process of specifying requirements.

To analyze the data needed by the selected microservice it was decided to elicit the functional requirements for the microservice. In other words, analyze the use cases the microservice will be responsible for, and analyze what data that is needed to perform said use cases. The requirements were written on the form "Given When Then", which is called "specification by example" and something that Martin Fowler among others recommend [Fowler, 2013]. An example of a requirements was "Given that a message is marked as 'sms', when the message arrives, then send it as an sms".

The requirements were set up in an online tool called Trello[1]. Each new entry in Trello is called a card, and each card can be enriched with additional information like detailed descriptions. Each requirement were explained in detail in the cards, including the data needed to perform the requirement. In addition, we broke down each of the requirement into smaller sub-tasks to understand in detail how the microservice would perform the requirements.

As the requirements had been elicited and the data needed to perform the requirements had been analyzed, a database schema were created. This was done to summarize and to understand the full picture of the data that is needed for the selected service. This database schema were to be used later during the implementation.

[1] Trello: https://trello.com/

2.2.4 Designing the architecture

The next step of the migration strategy was to design the architecture for the microservice. This was done using an online modelling tool. The aim was to design and clarify how the microservice would interact and communicate with the rest of the system from a technical perspective.

The requirements created in the previous step were shown to be helpful and were used to understand when the microservice need to interact with other parts of the system. From this we analyzed whether the communication should be done through asynchronous communication using an event bus, or by synchronous communication using API calls.

2.2.5 Selecting tools and implementing

According to the migration strategy, it was at this stage time to select the tools that were to be used during the implementation. Many tools and frameworks were explored during the process, and some decisions regarding what to use were done before the implementation, such as using Azure Service Fabric for cluster management. However, it was hard to do decisions regarding tools before the implementation started and it was shown that the migration steps "Select tools" and "Implement" were an intertwined process, where some tools where analyzed from the start and used, while some tools were selected when it was time to implement them. The tools were mostly found through Google, and we preferred to use things from Microsoft Azure since that is the cloud provider Eliq uses to host their current system.

A couple of weeks was spent on implementing the microservice. During the implementation there was an effort made to make the new microservice more understandable and structured than the previous code in the monolith. Efforts were also made to set up an infrastructure that could be reused when other microservices are to be developed. For example how the microservices should communicate with the event bus. The goal was to create the microservice as an exemplar that can be looked at in the future as a reference of how to properly build and structure the microservices.

2.2.6 Evaluating the microservice

At the end of the implementation the microservice was evaluated. Firstly, a meeting with the lead developer at Eliq were held where the implementation was discussed. It was concluded during the meeting that the microservice would need to be changed which caused us to go back to implementation again. This will be described in more detail in the result in Section 4.7.

After the changes had been made the implementation was evaluated once again. This was done by first evaluate whether the microservice was able to perform all the specified requirements and then by another meeting with the lead developer at Eliq were held where the implementation was discussed once more. No additional evaluations, such as performance testing or including people who had not been part of the implementation, was done. Ideally the implementation would have been done by a person who had not been a part of the migration, but who was knowledgeable in microservices. Eliq is a small company with one small development team, and everyone there had been involved during the implementation. Therefore we could not find people that could evaluate the microservice that had not been involved in the process of creating it.

In addition to evaluating the implementation, the evaluation meeting with the lead developer also aimed to understand if Eliq were still keen on working with microservices in the future. It also aimed to understand if his vision of working with microservices had been changed during the project. As mentioned before, a survey was sent out to the developers of Eliq at the start of the phase. Some of the questions during the evaluation meeting were based on this survey to see if the answers differed.

2.2.7 Enhancing the process

Finally the strategy in its entirety was evaluated. Everything that had been done when conducting the migration strategy was reflected upon, and some changes were made in order to make the strategy reflect the actual working process better. The time it took to do preparatory work, implementing the code and configuring different tools and frameworks was measured throughout the phase. This data was analyzed and later used to figure out why we did not have time to do the migration strategy again and migrate another microservice, and if anything could have been done more efficiently.

3

Phase 1: Gathering Knowledge

The goal of phase one was to find support and potential answers to the research questions RQ1, RQ2 and RQ3. In other words focus on identifying challenges industry have faced when working with or migrating to microservices. We also wanted to identify the challenges that a small company might face during the migration and to understand what attributes that should be considered when selecting what of part the system to extract. A literature review was done along with five interviews. The outcome from the literature review and the interviews together with an evaluation of the result is presented in this section. Lastly, the result was evaluated and a migration strategy was created that was later used in the second phase.

3.1 Result from literature review

The literature review aimed to identify challenges when working with microservices together with strategies that can be used when facing these challenges. All challenges and strategies found were then categorized and summary of this categorization can be found in Appendix B. The papers did not always use the same notion for challenges and strategies which resulted in some interpretations, in order to create a unified language in the report.

3.1.1 Service boundaries

One of the first steps when a microservice is going to be developed is to define the purpose with it and its responsibilities [Kalske et al., 2018]. Each microservice in an application will be responsible for a set of use cases and hold functionality to meet these use cases. In order to avoid that the services responsibilities overlap with each other, it is important to thoroughly investigate the system and discover areas of responsibility. This process can be complex and will in the end affect the result of the application's architecture. The process of deciding the responsibilities of a service is often referred to as *defining service boundaries* in literature and is important when migrating microservices from a monolithic application [Fowler and Lewis, 2014] [Kalske et al., 2018] [Microsoft, 2018].

Challenges/Risks
Microservices can provide several benefits to a system and improve the development process. One example is that each service is isolated and have independent lifecycles which allows developers to maintain and test a certain service without affecting other parts of the system. Apart from having independent lifecycles, microservices also enables developers to scale each service independently and use various programming languages within a service [Dragoni et al., 2017a]. To which degree a system can take advantage of these benefits is heavily dependent on well defined service boundaries. For example if a service is highly coupled with other services the lifecycle might not be independent. The service boundaries also affects other aspects of the architecture such as communication and handling of data, which is explained later.

Poorly designed service boundaries could lead to a need for refactoring in the future. In a microservice architecture, refactoring between service boundaries is complex and harder than refactoring between in-process libraries. This is because all services that are relying on the refactored services needs to be updated according to potential updates in the communication interfaces. In order to make older versions of a service still work after an update, the update should be backwards compatible [Fowler and Lewis, 2014].

The process of defining service boundaries is challenging. This might be due to uncertainties regarding the size and granularity of a service boundary. If a microservice is too big it reduces maintainability [Dragoni et al., 2017b], and if the services are too small or too fine-grained, performance overhead can be introduced due to chatty communication between services [Kalske et al., 2018]. The software company MGDIS SA discusses this in their experience report regarding migration to microservices and states that it was difficult [Gouigoux and Tamzalit, 2017], which strengthens the claim that defining granularity and size is hard.

Strategies
One strategy that is commonly recommended when facing the challenge of defining service boundaries is using practices from Domain Driven Design (DDD). More specifically forming the service boundaries around bounded contexts, which is a concept from DDD [Fowler and Lewis, 2014] [Newman, 2015] [Microsoft, 2018]. A bounded context is a certain area in a business model. The functionality and data needed to execute the business logic within a bounded context exists within that bounded context. In an e-commerce business domain, an example of bounded context could be "Shipping" and "Orders". All business logic in regards to creating an order is within the "Order" bounded context, and the same is true for shipping. [Vernon, 2013]. The idea is that these bounded contexts forms a natural boundary between the parts of the system which fits well with microservices [Fowler and Lewis, 2014].

DDD aims to define a real-world model of the business domain that the software acts in, and therefore software should then be formed around the domain model. The bounded contexts are then formed from the domain model. Another benefit of a good domain model is a unified language in the organization, which will help developers and domain experts to communicate. However, rich domain knowledge is needed in order to create a good domain model. One important step is therefore "knowledge crunching" where domain experts and developers collaborate with the purpose of understanding and visualizing the domain [Evans, 2004].

There does not seem to be any definite answers to what the size and granularity of a service should be, but some recommendations have been found. In one of Microsoft's articles regarding microservices, it is stated that the focus should not lie on making the service as small and fine-grained as possible, but rather focus on defining a clear purpose. Finding a suitable size of a service is normally done in iterations, since knowledge is gained about the domain over time [Microsoft, 2018]. The software company MGDIS SA stated in their report that they applied a trial and error approach before they found a suitable size [Gouigoux and Tamzalit, 2017], which supports Microsoft's statement about finding a suitable size in iterations.

3.1.2 Automation in integration and deployment

One key benefit when using microservices comes from them being independent and having independent development cycles [Dragoni et al., 2017a]. This enables companies to independently deploy updated services without redeploying the entire application. As a result the time to market is reduced, which is one of the reasons companies choose to apply microservices [Newman, 2015]. However, this sets some constraints on the underlying infrastructure. To take advantage of the benefits mentioned above and to deploy services efficiently, automation strategies needs to be put in place [Balalaie et al., 2016]. Automating the deployment process is often called continuous delivery which automatically builds, tests and packages the software artifact for release, after a change have been committed [Humble and Farley, 2010].

Challenges/Risks
One of the underlining principles of continuous delivery is continuous integration. The idea with continuous integration is to continuously integrate committed changes to the application which is then built and tested. In this way, the developer can receive feedback right away on whether the change works or not. The goal with this is to always have a working copy of the software [Humble and Farley, 2010]. There exists several tools that can be used to set up continuous integration, which can be of help during implementation. It is important that the continuous integration is set up in a way that lets the microservices be independent. For example, if there is one line of change in one of the services, all the other services should not always need to be tested and rebuilt [Newman, 2015]. To take advantage of the continuous integration and the services independence, careful considerations must be done.

Continuous integration is great for quick feedback loops when changes are implemented. The next step is to automatically prepare the application for deployment. This automation is often set up as a pipeline built up through multiple stages. The stages can be additional tests such as load testing, which can provide even more feedback to the developer. If the software passes all the stages in the pipeline, the software artifact is ready to be deployed [Humble and Farley, 2010]. Having this process automated is beneficial, and in some cases essential [Fowler, 2015]. Handling deployment for all services manually would be time consuming and reduce the time to market. This is something that MGDIS SA discusses in their experience report where they did an analysis of how the operations costs can be lowered by applying automations. If an automated process for testing and building the services is implemented, the deployment costs stay the same as the number of services increases [Gouigoux and Tamzalit, 2017].

In addition to having this process automated, it is important to have good test coverage. Especially during a migration to microservices from a monolithic application. Newly implemented microservices can then be validated against those tests to reduce the amount of possible bugs and ensure they are working correctly [Kalske et al., 2018].

Strategies
There are no specific strategies used for setting up the required automations. The automation and the release pipelines will differ from software to software [Humble and Farley, 2010], which means that there is no single correct answer. Setting up the infrastructure to allow for continuous delivery is important, but also complex. It is therefore encouraged that teams are set up as DevOps teams where developers and operations work closely together [Kalske et al., 2018] [Balalaie et al., 2016] [Fowler, 2015]. Through collaborations between developer and operation teams the automations can be used more effectively [Fowler, 2015].

3.1.3 Communication between services

When developing a monolithic software application the software is often divided into components. These components serves some type of purpose and holds some kind of functionality and can communicate with other components through in-process calls, by invoking language-level methods [Microsoft, 2018b]. In a microservice application these components may not be running on the same process and in-process calls are no longer available. Another communication protocol must therefore be used to communicate between the components, such as HTTP [Microsoft, 2018b].

Challenges/Risks
Changing communication mechanisms is one of the harder challenges when moving to microservices [Microsoft, 2018b]. This could be supported by Alshuqayran N., Ali N. and Evans R. mapping study from 2016 that shows that the most mentioned keyword related to challenges with microservices is communication [Alshuqayran et al., 2016]. This is partly because the communication between components in a monolith can afford to be "chatty" since it is in-process calls. However, chatty communication between microservices causes performance overhead and needs to be limited [Newman, 2015]. How and when microservices should communicate must be considered carefully.

The challenge with setting up the communication mechanism is tightly coupled with the challenge of defining service boundaries of a microservice. Each service should be independent, and in order to be independent each service must have high cohesion internally and low coupling to other services. In other words, functionality related to the service purpose should be kept in the service and it should have loose dependencies on other services [Newman, 2015] [Dragoni et al., 2017b]. If a service is dependent on many other services to complete a tasks it is likely to cause an overhead and lower the performance. This kind of issues could also be a result of the services being too fine-grained [Kalske et al., 2018].

Strategies
The communication is heavily dependent on how well the service boundaries are specified. The communication should therefore be considered during the process of defining service boundaries. It is also important to consider type of communication that is needed in specific situations. One important decision is whether to use synchronous or asynchronous communication [Microsoft, 2018b]. Some operations can benefit from, or require, an answer from its request. In these cases the synchronous communication is more suitable. Unfortunately synchronous communication blocks until it receives a response, which could cause delays. If the caller does not necessarily need an answer from the recipient, it could be more suitable with asynchronous communication [Newman, 2015].

3.1.4 Decentralized data

As mentioned before, microservices should have high cohesion internally and low coupling to other services. To provide this, each microservice should manage their own data [Fowler and Lewis, 2014]. In this way, each microservice have direct access to the data required to do certain computations without asking other services to provide data. Therefore, a concept referred to as database-per-service is often promoted. Database-per-service means that all services that require a database should have their own database [Richardson, Chris, 2018] [Newman, 2015] [Fowler and Lewis, 2014] [Hasselbring, 2016]. Using the concept of database-per-service opens up opportunities to use different types of databases depending on the purpose of the service, and lets database tables update without affecting other services [Richardson, Chris, 2018].

Challenges/Risks
Letting the services manage their own data comes with a few drawbacks and adds complexity to the software. In some cases, data will be replicated across several databases and mechanisms are needed to provide consistency among the databases [Fowler and Lewis, 2014]. Microsoft presents an example in their article about this subject where there are two services, one basket service (responsible for items in users basket) and a catalog service (responsible for items in the e-commerce) in an e-commerce application. If a price updates for an item in the catalog service, the price information in the basket service also needs to be updated [Microsoft, 2018a]. Mechanisms for propagating updates to other services are therefore needed.

Using the concept of database-per-service in a microservice architecture can be more complex than in a monolithic application. In a monolithic application there is often one database used to store data and manage data and consistency is reached by using transactions [Kalske et al., 2018] [Fowler and Lewis, 2014]. Guaranteeing consistency in a microservice architecture requires mechanisms to execute distributed transactions. Distributed transactions are complex to implement and is discouraged when using microservices. Instead, using eventual consistency is recommended [Fowler and Lewis, 2014] [Newman, 2015]. This means that the replicated data will eventually be consistent, but not guaranteed to be consistent at all times.

During a migration from a monolithic application to microservices, the database tables used in the monolithic application will likely need to be changed. In a relational database, some relations between tables will have to be removed and separated into different databases. This process can be complex, since changes to the database queries are needed to make sure that the tables still updates as desired [Newman, 2015].

Strategies
As explained above, database-per-service is encouraged in a microservice architecture. Therefore, a strategy to define what entities that should exist in each database is desirable. What entities that exists in each database is, however, dependent on the service boundaries and should be considered when defining service boundaries. As discussed in Section 3.1.1, practices of domain driven design (DDD) is encouraged when defining service boundaries in the system. DDD also contains tactics and patterns that can be used to define what data is needed in each service boundary. The idea is to define the data entities that exists in each bounded context and analyze relations to other entities and what values these entities hold. As the relations and values for a entity has been analyzed, they are put into something called *aggregates*. One entity is said to be the root of the aggregate and holds a global id which will be the entry point to the aggregate [Vernon, 2013]. In an e-commerce, one aggregate could for example be an order and that order holds several order items. The order will be the aggregate root and the order items will be child entities to the order [Microsoft, 2018c].

Aggregates can reference other aggregate instances [Vernon, 2013]. As the e-commerce example described earlier, the items in the basket refers to the items in the catalog by item id. When a domain event happens, such as the prices changes for an item, the referenced aggregates needs to be informed. In DDD, this is accomplished by defining "domain events". Domain events are created when something in the domain changes and can be propagated to others using communications patterns [Vernon, 2013]. In order to know who is interested in which events they should be considered during the process of defining service boundaries.

3.1.5 Fault tolerance and fault handling

One of the benefits of using microservices is that it allows an application to be resilient to failures. Since each service is containerized and independent to others, a failure of a service should not directly affect other services; there is no single point of failure[Dragoni et al., 2017b]. Detecting, reacting and making a microservice application resilient to failures can be complex.

Challenges/Risks
Moving towards microservices from a monolithic application introduces new sources of possible failures, more specifically; the fallacies of distributed systems now applies. Therefore, mechanisms to handle these types of failures needs to be put in place. If there is a lack of failure handling, there is a risk for failures to cascade and cause failures of other parts of the system [Newman, 2015]. To avoid that services, that rely on a response from a failing service in the system, fail as well, it is important to be aware and handle such cases [Dragoni et al., 2017b].

It is inevitable to encounter failures in a microservice application. In these cases it is good if other services can handle the failures, but developers will also need to detect and react to these failures. In a monolithic application, for example a REST API, there is one or more instance running the API. There is only one application to monitor and if there is an issue, such as slow response times, it is relatively easy to know where to start. When migrating to microservices there is no longer a single point of failure and the problem can be caused by various services. Therefore monitoring is needed in order for a developer to understand what causes the error and where to start looking [Newman, 2015]. Monitoring introduce complexities to the system since there are many services that needs to be monitored and multiple hosts that the services are running on. The developers need to consider what to monitor, for example response times on HTTP calls and CPU load on all hosts, and setting up a system that lets them be aware of when problems occur [Newman, 2015].

Monitoring is good for detecting failures, but in order to troubleshoot a failure in the system, logging system needs to be implemented. It is also important to have good tools for visualizing the monitors and logs, to help the developers understand errors and statuses from the microservices [Fowler and Lewis, 2014].

Strategies
In order to provide good fault tolerance, circuit breaker pattern is often recommended [Newman, 2015] [Kalske et al., 2018]. After a few continuously failed request attempts, the circuit breaker "accepts" that the service has failed and stops executing requests. The circuit breaker thereafter checks if the failed service has recovered. If the faulty service is healthy again, new requests will be executed [Newman, 2015]. By implementing this, the failure does not cascade to other services.

It is important that the developers can detect and be aware of failures happening. Visualizing the monitors using a dashboard makes them more accessible to the developers [Fowler and Lewis, 2014], and there exists several tools today that can be used to do this.

Monitoring produces logs and these logs can be used to troubleshoot a failure, and therefore it is important that the logs is accessible and searchable. Logs should therefore be sent to a centralized store where you can query logs and e them [Newman, 2015]. Microsoft also suggests sending a unique correlation ID in each operation that is passed in as a variable when logging. In this way the developer is able to trace an entire operation [Microsoft, 2018d].

3.2 Results from interviews

This section presents the results from the interviewees with focus on three main areas; the working processes the interviewees had when developing microservices, the challenges they faced and how they dealt with them, and how they would work with microservices in small companies. A table with data about the companies and the interviewees can be found in Section 2.1.2.

3.2.1 Working processes

To understand the interviewees' context with microservices, questions were asked about their working processes. There were two main areas in which the interviewees had worked, either with developing microservices from scratch or migrating from a monolith. The highlights of what they said during the interviews are presented here. However, interviewee E had primarily worked with microservices in an exploratory way to gain a competitive edge as a consultant, therefore E did not have experience in any work process.

3.2.1.1 Microservices developed from scratch

Three of the interviewees had worked with developing microservices from scratch, however two of the interviewees (A and D in Table 2.1) had worked on the same company with the same project. The third interviewee (C in Table 2.1) has worked for two different companies with microservices, both of which developed microservices from scratch.

In the company interviewee A and D worked for (AD in Table 2.1), a feature specification came from outside of the team through some internal ordering manager. The team analyzed in a short meetings what the feature was and created mental models for how it fitted into the architecture. Everything was done case to case, sometimes a feature needed several microservices, sometimes one, and sometimes it was just a simple change in a microservices such as adding an endpoint. Sometimes the project leaders drew boxes on a whiteboard on how the communication between services would take place, but there was no detailed planning. This worked very well at this company and the flow was good, and everybody knew what they were doing according to interviewee A and D.

Interviewee C had worked with microservices in two different companies. In company C1 they spent little time doing preparatory work such as understanding the domain, business flows and designing an architecture. They worked in iterations and started the implementation without any deeper analysis of the system. C said that this system could be improved upon a lot, but it does work. However, how the system is affected when new code is added is unclear, and it might be necessary to change the implementation in several places to add new functionality.

In the other company C worked for, they had a very different work process. Here they analyzed the business carefully before implementing the system. They identified all business flows, tried to understand what the system should do, designed the architecture based on this and then started to code. C said that they preferred this work flow and that it yielded a better result compared to the other company.

3.2.1.2 Microservices extracted from a monolith

Three of the interviewees (A, B and D in 2.1) had worked with migrating a monolithic application to microservices. As mentioned before, A and D had also worked with building microservices from scratch.

On the other company A worked for (A1 in Table 2.1), the teamed owned the entire development cycle, both on the current monolithic application and the new microservices application. They made their own decisions and focused on developing services that could be used by the customers immediately. They did not want to implement services that could not be used right away due to delays from external providers. This process worked well enough, but it was better at the other company A had worked for, where they built the system from scratch.

Interviewee D had also worked for a smaller company (D1 in Table 2.1) where they had a monolith from buying another company. However, the company decided that they did not want to keep this monolith, and instead implemented microservices from scratch. This was mainly because the old monolith was badly coded. When they built the new system, they designed the microservice architecture by forming it around the business logic.

The company B worked for (B1 in Table 2.1) wanted to look into microservices since they were growing and moving parts of the production to another country, which caused issues with communication. Therefore, interviewee B investigated whether or not the company should move towards microservices, by conducting a proof of concept on how the migration would occur. Although interviewee B did not do an actual migration, he managed to migrate two microservices for his proof of concept. In the end the company decided not to migrate to microservices, mostly due to the cost of training all employees, explained further in Section 3.2.2. Instead they deemed it safer to continue with the monolith for now, and do not have any plans to move to microservices at the moment.

3.2.1.3 Reflections

It was hard to understand from the interviews how a good migration could occur. The only company that truly migrated from microservices was company A1, and while their process was successful, interviewee A said that he preferred to develop microservices from scratch. In addition to this, company D1 had an old system when the began developing microservices, but decided to create a completely new system to replace it and therefore built the microservices from scratch. Perhaps building the new system from scratch can be a good option even if you have an old system. It was also interesting that interviewee B managed to extract two microservices in his proof of concept, but the cost of migrating was not worth it according to the company.

We realized during the interviewees that most companies did not do any specific domain analysis.

For example in company AD they simply focused on the new feature request they received and mapped it into where it fitted in to the microservices, without detailed planning. It is interesting that such as simple approach worked in a company with around 100 microservices, where the teams worked at two different locations (data from Table 2.1). The fact that they also did not do any heavy domain analysis in company C1 and still managed to create a fully functional system, could suggest that domain analysis is not as important as literature describes it (discussed in Section 3.2.2.1). However, interviewee C also said that he preferred the work process in company C2 and that it yielded a better decomposition of the system, and that it is unclear how the system at company C1 is affected by new features. We therefore inteperate the result as that you may not necessarily need to do a thorough domain analysis to build microservices, but if you want to build the best possible system it is important.

3.2.2 Challenges

During the interviewees the biggest challenges each interviewee had during development of microservices were discussed, along with what they did to combat the challenges. This section details what was discussed.

3.2.2.1 Understanding the domain

All interviewees talked in some way about the importance of understanding the domain and what exists in the domain. Several of them said that it is the hardest challenge when developing microservices. However, the way they approached the challenge varied a lot.

Interviewee D and E mentioned domain driven design as a key way to design the microservice architecture. They said that it is important to look at what entities exists in the system, and to map these entities to microservices. Using this principle, it is easy to form the microservices around what exists in the domain. E said that it is important that if you try to understand the domain, then do not only talk to the developers and the architects. You should include the sales people as well, and understand their perspective on the product. Interviewee B also touched upon this subject, and said that it can help in understanding the current code if there is a business analyst available who knows about the domain and how the monolith is structured.

Interviewee A said that they would look at the current code as little as possible and try to understand what functions and requirements are necessary when developing a new system. Based on this you can design an architecture or at least an architecture draft on how the system should look like in the end. Otherwise it is very easy to get stuck in how the system looks right now, and not how it should look like.

Interviewee C said that there are guidelines to follow when decomposing a system, but you should not stare blindly on requirements that may exists and instead ask yourself, what can change in the system? And if such a change occurs, how will the change impact our system? Try to minimize the effect of change. C also talked about different decomposition methods such as volatility based decomposition, flow first design among others. Methods like these can help in designing a system that supports the business. C continued to talk about how to understand the current domain and system when you are new to it. C said that they would try to get some sort of flow chart from existing documentation. If there is no documentation, then logging and follow steps in the current system can help a lot, and talk to business people about the domain. It is important to understand what the purpose of the system is and what problems it tries to solve. If there is no real documentation and no one responsible, then the easiest way to understand the flows is with tracing.

Reflections
We thought that it was very interesting that most of interviewees had not analyzed the domains

at the companies they worked for, but they still thought that it was important and beneficial to do when migrating. They all agreed that finding a suitable decomposition before migrating is important, but they had some varied ways on how to do it.

Interviewee C was the one who promoted the idea of finding a suitable decomposition the most, and had research several different methods of doing it. Interviewee C was, as we discovered when talking about the working processes, not too happy about the system in company C1 where they had not done a proper domain analysis. Even though that system works, C was skeptical in how this system handled changes when new features are implemented.

For these reasons, we felt confident in that analyzing the domain is an important step when migrating to microservices.

3.2.2.2 Understanding the current monolith

Interviewee A said that the biggest issue when migrating in Company A1 was understanding the current monolith, claiming that it was a huge project with tens of thousands lines of codes. This made it incredibly difficult to understand what you could change without affecting something else. Their solution for understanding existing code is to do simple things such as looking at classes and methods, and trust in the naming of methods and classes. According to A, it is important that you have a goal when doing this, for example looking into a specific data flow. If there is no documentation, then that is the easiest approach.

Interviewee B also talked about the issue of understanding the current monolith, and said that it was hard to deal with a tangled code base. It was also hard to find all places where a method or class is referenced and how it interacts with the rest of the monolith. A tangled code base makes it hard to extract without touching a hundred places. B said that in order to solve this issue it could be beneficial to consult with a system expert who can explain how the different parts of the system works. However, sometimes it might be a good option to rewrite the code for the new microservices, but that might not be the best solution from a business perspective.

Reflections
Understanding the current monolith seems to be a difficult thing, especially if the monolith is big. As interviewee B said, if you want to extract something from the monolith but the code is complex, it will slow down the process since you have to refactor code everywhere. This issues could probably be solved by having people with experience and knowledge in the old system extracting the microservices, since they know how the features are implemented and might be able to extract them more easily.

3.2.2.3 Selecting microservice to develop

The interviewees were asked questions about how they selected what to develop into a new microservice, both in extracting but also developing from scratch, since we thought that was unclear from the literature review and we wanted to know more about it.

As mentioned in Section 3.2.1.1, company AD, where both A and D had worked, received feature specifications from outside of the team through a feature request, and from this specification they decided if it was one new service, several new services or just a simple change in an old one. They developed the microservices from scratch, but prioritized building the feature request they received. D said that the process of translating a specification to microservices was done internally in the team and was based on some principles, but mostly experience.

Interviewee B said that when he migrated the first microservice, he selected one which was easy to extract, and that he would do that again. Interviewee C said that they tried to design flows in

the system and then build the flows one by one.

A said that in their current team, they simply formed their own style over time when defining a microservices, through trial and error and discussions in meetings, and that there was no definitive answer on how they defined a microservice.

Reflections
It was not very clear how any of the interviewees selected what microservice to develop, or how they selected what to extract if they had a previous monolith. Most of the time it seemed to be based on experience working with microservices, and they did not have any particular attributes to consider. Company AD, who built microservices from scratch, received a feature request and built microservices based on that. They then had to evaluate whether or not it was a new microservice, or simply adding more functionality to an existing one, which they did based on previous experience. The only real attribute to consider came from interviewee B who suggested that you should start by extracting something easy and untangled if you have a previous monolith.

3.2.2.4 Architecture and design

Interviewee A mentioned that designing the architecture from the start can be a good idea, and building a skeleton of how the architecture should look like in the end. How the architecture should look like depends on case to case, and there is no single correct answer. Interviewee C mentioned that if the architecture is poorly implemented, it can be hard to implement new functionality into the system, and things that should not be affected by a change might be.

Interviewee B talked about the difficultly of figuring out how the microservice architecture should look like, and that it might depend on the current architecture. If the current architecture is a mess, designing the new one will be hard. B said that the solution to these issues is simply to try to get an overview of the system, figure out the purpose of the software and then go into smaller pieces.

Reflections
Interpreting the interviews we conclude that designing an architecture for the system is beneficial and can help in understanding how the microservices should communicate and share data with each other. It can also help in understanding where to add new functionality to the system and how changes will affect the system as a whole.

3.2.2.5 Technical challenges

Technical challenge such as how to handle monitoring, logging or testing were not discussed as much by most interviewees. However, interviewee E mentioned a couple of issues they had while exploring microservices that might be hard to grasp for people new to the subject. E said that changing the mindset was hard in itself, with things like stateless servers to enable horizontal scaling, how to use redis as a distributed cache among other technical things. It was also hard to create different configurations according to E. What if you want a testing environment, a staging environment and a production environment? If there are a lot of services, and you want to create environments for all these cases, how do you do it? A branch (git branch) for each, environmental variables, different repositories? These were all things E had issues understanding while researching microservices and trying them out.

Interviewee D talked about challenges in regards to communication between microservices and that it is typically asynchronous. D stated that if synchronous communication is used you loose a bit of the purpose of microservices. However, asynchronous communication comes with several drawbacks and a need for eventual consistency, especially if the data is decentralized. User experience can suffer if a user does something and expects a synchronous response, but the request goes through

several microservices and takes a long time to finish. D did not have a definitive way to solve this, perhaps by using some layer that control the state. It all depends on a case to case basis according to D.

Interviewee B mentioned that supporting functions such as continuous integration and continuous delivery might be hard to grasp, especially if there is a lack of knowledge in such tools and concepts. In a monolith, the testing and deployment can be done manually, but you can not do that with microservices. Finally, interviewee D mentioned that even though it is possible to have many different technologies and languages in a microservice architecture, it is probably not a good idea. It takes time to learn about all the technologies and if the microservices are built in different languages, some developers might not be able to contribute to those microservices.

Reflections
There seem to a be a few technical challenges to consider when developing microservices, but none of the interviewees said that any technical challenge was the hardest challenge for them. We interpret this as that you should be aware of the technical challenge and investigate how to address them, but it is more important to understand the bigger picture and how to structure the microservices as a whole. We also found, as mentioned by interviewee D, that you should try to limit the number of technologies you choose, otherwise it may be complex to develop the microservices and some developers might not be able to contribute as much.

3.2.2.6 Costs

Cost was mentioned several times during the interviews as something to consider. Interviewee A and C said that microservices should not be seen as a cheaper way to develop, host and scale. A said that it is harder to predict the cost of hosting microservices, since there are many services to take into account and not a single large one. It is important to consider what you actually need, and remember to scale down or even shut of services when the load is low, otherwise the bill might be high.

Interviewee D said that one of the reasons that they use a lot of eventual consistency is due to cost. Many of the integrated systems they have inputs data into the microservices in batches. It would be very expensive in infrastructure costs to be able to handle all that data synchronously, thus an asynchronous solution with eventual consistency is required.

Interviewee B also talked about cost as an issue and said that training the employees was the main reason why company B1 decided not to migrate to microservices. There was a lack of knowledge in the teams on how they should work with the microservices. For example, some teams had no previous experience in docker and some teams lacked DevOps experience, which would be required. It would cost too much to teach all teams about the technical parts, and thus they decided not to migrate to microservices.

Reflections
We found it interesting that literature hardly mentions costs as an issue when developing microservices, and yet some of the interviewees reflected on it as a big issue. The fact that company B1 decided not to migrate to microservices because it would cost to much to train all developers in working with microservices, could be an argument against developing microservices, especially in larger companies where there are both technical and organizational challenges to consider. It is probably important to analyze the costs against the benefits of microservices before extracting, to avoid spending time and money on implementing something that might not benefit the company in a significant way.

3.2.3 Small companies

Another area that was discussed during the interviews was how the interviewees would conduct a migration in small companies and if any of the challenges in the previous section were harder or easier for smaller companies.

The unpredictability of the cost when using microservices might be an issue for small companies according to A. Each services has a price, and if you want automatic scaling the cost will vary. Also, developers can have a tendency not to feel responsible for the cost and create new services and databases whenever they need it when developing microservices, which might result in a high price for the infrastructure. According to A, it is much easier to predict and understand the price of a monolithic system, since you know that this is the system that is running on this machine.

Interviewee B talked about issues with supporting functions such as continuous integration could be a big issue for smaller companies, since there might not be enough developers to properly handle such functions. Although that might be an issue, B also said that it could be a benefit to start with microservices when you are a small companies, since it will cost less to train all employees about microservices and the tools needed. All employees does not need to understand things such as docker if that is used, only a couple in each team that can help the others, and in a small company that might be one or two people.

Migrating in a small company might be easier than in a large company, according to C. It is easier to split the responsibility to specific people if it is a small company, since small teams often know each other well. However, C also mentioned that the cost of hosting and scaling microservices might not be something small companies are ready to commit to. Finally, E thought that it might be hard for small companies to develop the microservices while at the same time maintaining the current system. You also need to introduce DevOps into the company, and teach all employees about it, which is costly and takes time.

Reflections
We would argue, based on the finding here, that the challenges for small companies are not that big and that you can develop microservices in small companies. Of course there are a lot of challenges, such as cost for training developers in new tools and technologies such as continuous integration, but these challenges applies to larger companies as well. One issue that might more challenging for small companies is the small number of developers, since that might make developing microservices, while simultaneously maintaining the old system, difficult. However, if the company is prepare for the cost of developing microservices and are aware of the challenges with migrating, we believe that it is possible to migrate as long as the company fully commits to doing it.

3.3 Discussion

The literature review and the interviews yielded valuable insights in regards to challenges when developing microservices. One of the main differences between the literature review and interviews was the amount of technical challenges that were brought up. The literature review resulted in more technical challenges than the interviews. This could be due to time limitations during the interviews which did not allow for going into details, or that the technical issues are not as challenging as one may interpret from literature.

This section will discuss similarities between the interviews and literature. It will also present a summary of the biggest challenges that was found in regards to a migration to microservices, together with strategies of how these challenges can be addressed. These challenges and strategies will later be used as foundations during the design of the initial migration strategy. Below, a list of all the challenges and strategies can be seen. The rationale behind each is explained in the upcoming sections.

3.3.1 Understanding the domain and defining service boundaries

From the literature review we found that a microservice applications ability to take advantage of the benefits with microservices is heavily dependent on the the service boundaries. As discussed in Section 3.1.1, finding suitable decompositions of the application is of high importance, but also one of the most challenging steps during the migration. The process requires careful considerations and a deep understanding of the domain. In order to obtain deep domain knowledge, the domain should be analyzed, which was also highlighted by our interviewees. The interviewees stated that it is important to have a great understanding of the domain, but also that the process of gaining that understanding is hard.

Arguably the challenge of managing decentralized data (Section 3.1.4) is closely related to defining service boundaries. Each service boundary should manage their own data, and it is important to consider the data in the system during the process of defining service boundaries. By understanding where certain data exists and where data is replicated, relations between boundaries can be found.

Strategies
As discussed in Section 3.1.1, using practices from domain driven design and define the bounded contexts of the system can be helpful, in order to obtain an understanding of the domain and find natural boundaries of a system. Domain driven design was also mentioned as a good way of understanding the domain by some of the interviewees (Section 3.2.2.1). The process of defining these bounded contexts requires inclusion of people with great domain knowledge, preferably a collaboration between developers and domain experts. The domain model should also continuously be developed as the system changes, to keep them relevant and useful for the developers when they implement the microservices.

Domain driven design also contains concepts that can be used when analyzing where data exists in the system and how that data traverse in the system, as discussed in Section 3.1.4. Using this in combination with the bounded contexts could be a good option during the domain analysis.

The interviewees also stated that domain analysis is important in order to get a high level understanding of the system. However, domain analysis often seemed to be neglected and ignored in their work process. Only one interviewee said that they had done a proper domain analysis before starting to work with microservices. This does not align with literature where it seems to be the most important part. Nevertheless, some of the interviewees said that they would have preferred to have done a better domain analysis, and the interviewee who had done it said that the result was better because of it. Therefore we still interpret it as a proper domain analysis is a vital part of extracting microservices, at least when you are new to the concept. However, as the knowledge about the domain and the system increases, domain analysis might be less important.

Challenge CH1: Understanding the domain and defining service boundaries
Well defined service boundaries is important in order to take advantage of the benefits with microservices. Defining service boundaries is complex and requires great understanding of the domain.

Strategies
Domain knowledge is obtained by analyzing the domain in collaboration with domain experts and developers. Concepts from domain driven design can be of help during the process.

3.3.2 Managing communication and decentralized data

A migration to microservices will introduce challenges in regards to data management. As shown in Section 3.1.4, each service should be responsible of its own data and services will need to collaborate in order to provide consistency among the services. Because of the complexities using

distributed transactions to guarantee consistency, eventual consistency is encouraged. Handling eventual consistency can also be a challenge. Interviewee D gives an example of how eventual consistency can affect the user experience in cases where users expects updates to be done instantly and synchronously, when it is in fact updated asynchronously.

The challenge of managing decentralized data is closely related to the challenge of communication between services. Updates of replicated data should be propagated to all interested parties using some form of communication protocol. Decisions must be made in regards to which services that will communicate with each other and how that communication should be done. Making these decisions can be a challenge since poor decisions can lead to performance overheads.

As discussed regarding challenge CH1, decentralized data management is related to defining service boundaries. During an analysis of the domain, the data in the system should also be analyzed in order to understand where it is located and which boundaries that is interested in data updates. Challenge 1 is related to the understanding of where decentralized data will be needed to be managed, whereas the challenge discussed here is more related to the technical difficulties.

Strategies
The strategy of facing challenge CH1 will be of help in order to face this challenge. It is also important that this challenge is considered during the process of facing challenge CH1. On a technical level, no specific strategies was found but rather recommendations of what to consider when deciding on how it should be handled. One of these considerations is where synchronous or asynchronous communication should be used and how to handle inconsistencies. Interviewee D discussed asynchronous communication during the interview and stated that asynchronous communication is encouraged, since it can handle bigger load of requests and does not have to handle them all at once. In order to handle the same amount of load using synchronous communication, more services will have to be run which increases the costs.

Challenge CH2: Managing communication and decentralized data
Handling decentralized data is a challenge. When data is updated, it needs to be propagated to interested services. The propagation is done using some communication protocol, and poor decisions of communication will affect the performance of the application.

Strategies
This challenge is affected by the outcome from handling Challenge CH1. By understanding the domain, you should also gain knowledge in how data traverse in the system, which should help when considering how and when to propagate updates. Where to use synchronous and asynchronous communication and how to handle data consistency should also be considered.

3.3.3 Selecting microservice to extract

When developing microservices, the microservices will probably be extracted in iterations. The implementation process will most likely be complicated, since the code needs to be extracted from the monolith in an efficient way. In order to understand how to facilitate this process, we identified attributes that could be considered when selecting what part of the system to extract.

Strategies
Both literature and the interviews brought up strategies that can help in this answering these questions. Interviewee B said that when they migrated their first microservice, they focused on picking an easy, untangled part in the system. This was because they had little experience in the field of microservices and wanted to focus on learning and setting up the system. Starting with a complex task might have made the process even harder. This is also confirmed by in Zhamak Dehghani in an article about breaking up monoliths [Dehghani, Zhamak, 2018]. She says that you should warm up with something easy and decoupled to extract from the monolith and primarily focus on building the infrastructure and setting up the environment.

The interviewees who had migrated from a monolithic system to microservices all mentioned that the hardest part of migrating was to understand and change the current monolith. This somewhat relates back to understanding the domain, but they also talked about the challenges of understanding the code in the monolith and how to deal with a tangled code base. A tangled code base makes it hard to extract without touching a hundred places and it can be extremely difficult to understand what you can change without affecting something else. A solution might be to simply ignore the code in the monolith and rewrite the code when developing the microservice, something B mentioned. This will remove the "bad code" from the system, but from a business and cost perspective it might be sub-optimal.

We would also argue that continuously extracting microservices that are loosely tangled with the rest of the system can be a good idea, since removing these loosely tangled parts can help untangle other parts. This process can be exemplified with the game mikado (also called pick-up sticks)[1]. By continuously picking up sticks that are easily removed, other sticks become available. However, sometimes a player has to sacrifice their turn by picking up a stick that is tangled, which might shake up the sticks enough for future players to pick up the rest. Similarly, sometimes the developers might be forced to pick a more tangled part in the monolith to make future extractions more feasible, even though it might cause headaches.

Interviewee A said that when they select what microservice to develop next, they look at what services can be used and implemented into the current system right away in order to avoid spending time and effort on developing something that is never used. You should also look at what parts of the system are undergoing continuous change, since these parts often hinder developers from delivering value [Dehghani, Zhamak, 2018].

Often times during the interviews, the choice of what to migrate or what microservice is needed at the moment seems to be a gut feeling. During the interviews we realized that there are no real reasons as to why one part of the system is extracted before another, instead it is decided in a short meeting where everybody agrees on what seems to be a good idea. There might not actually be any good reasons to pick one microservice over another, it just feels better to do it in a certain order. This primarily applies when the developers are experienced in working with microservices and knows how the system functions.

Challenge CH3: Selecting microservice to extract
Extracting microservices from the previous system will likely be complicated. The ease of extraction will be dependent on what microservice is chosen to be extracted.

Strategies
During the initial part of the extraction, easy and decoupled parts of the system should be considered. It is also important to consider services that can be used right away, and what will change in short period of time. As the experience increases, the development team will have a better understanding of what makes a good service in the system.

3.3.4 Managing automations

In Section 3.1.2 we found that it is important to implement automations, more specifically implement continuous integration and continuous delivery. Applying automations increases the development teams ability handle deployments and decrease the time to market. Implementing these automations in a way that lets the microservices have an independent lifecycle can be complex. In addition, these automations are usually set up using tools and if no prior experience exists, then these tools needs to be learned. Interviewee B, the only interviewee who mentioned automations, mentioned that this is important and challenging to set up, especially in a small team if the competence does not exist.

[1] Mikado (game) : https://en.wikipedia.org/wiki/Mikado$_{(}game)$

Strategies
As the automations is set up it is important to consider and making sure that each microservice can be deployed and developed independently of each other. Other than that, no specific strategies have been found. Based on our literature review and interviewee B, it seems to be a question of experience and it is important to have people in the development teams that is comfortable in operations. In other words, development team should have a DevOps culture where developers and operations work closely together. If no prior experience exists, the automations will likely be set up using an trial and error approach and as experience increases, the pipelines will become better.

Challenge CH4: Managing automations
Setting up the infrastructure for automation is complex and requires experience. Automations are important to the development process and is recommended.

Strategies
Structuring the teams as DevOps teams can help with automations. Also, ensure that the microservices have independent lifecycles in the automations.

3.3.5 Handling and tracking failures

One advantage of using microservices is that it can provide high resilience to failure, but to provide that, mechanisms for handling failures are important, as discussed during the literature review (Section 3.1.5). There is a risk that failures in one service could cascade and cause other services to fail if there is no proper handling of failures. This can be complex since the services need to be able to detect and understand that a service has failed.

In addition to a system being able to handle failures, it is also important to detect and react to them. This is achieved by implementing solutions for monitoring and logging. Setting up this infrastructure does however require various tools to be configured or developed. Learning new tools, finding best practices and finding a suitable solution for a specific system will most probably be time consuming and require considerations. Deciding what and how to monitor and writing logs will be a challenge since the services runs in separate processes on several hosts.

Strategies
In order to handle failures, a pattern called "Circuit breaker patter" is often promoted, as discussed in Section 3.1.5. This is a pattern that allows a service to understand that a service has failed and will stop executing requests to that service. To react and detect these failures it is encouraged to use a dashboard that show statuses of the services in the application. Logging provides additional information about the system and can be used as a tool during troubleshooting failures. The logs should also be sent to a centralized logging storage that can be queried, to further help developers debug a failure.

What a system should monitor and what the logs should produce is not clear. This is because the need of logging and monitoring is dependent on the system. Monitoring and logging solution will likely evolve over time as experience is gained.

Challenge CH5: Handling and tracking failures
Handling, detecting and reacting to failures is important when using microservices, since there are many independent parts to consider. Monitoring and logging can be of great help, but setting up the infrastructure for this is complex.

Strategies
Try implement mechanisms such as "Circuit breaker pattern" to handle failures in other services. ing the logging and monitoring through a dashboard that the developers can use is recommended. Consider what properties that are interesting to monitor and log, and add the logs to a centralized storage to enable querying.

3.3.6 Challenges for small companies

The interviews yielded a few interesting insights in regards to what challenges small companies may face when migrating to microservices. In general, there does not seem to be any large disadvantage to being a small company, but the number of employees could affect the development process.

Interviewee B mentioned that providing the supporting functions, such as CI/CD, is complex and can be hard to implement, especially if there are no developers with experience in these functions. It could be argued that the probability for larger companies having employees with such experience is higher than for smaller companies. Small companies could therefore suffer from not having this experience which can lower the efficiency of the development process.

Another challenge for small companies could be to maintain the monolithic system during the migration towards microservices. This is something that interviewee E expressed during the interviews. Due to the low number of developers, it can be challenging to develop both in parallel. It could potentially be easier to develop them parallel in a bigger company with more developers and resources.

Another thing that might be a disadvantage for small companies could be other costs that comes with microservices. Interviewee A said that developing microservices is not a cheaper way to develop host and scale, and that it is harder to predict the cost of microservices. Interviewee D also mentions the cost of developing microservices, and said that microservices often comes with more infrastructure requirements, and hosting the infrastructure might be expensive. If you want to cut down costs when developing microservices, you have to plan out the infrastructure and maybe implement some sub-optimal solutions.

Interestingly, we found that it can be an advantage of being a small company. Interviewee C mentioned that it can be easier to split responsibilities to specific people in a small team where everybody know each other well. In addition, company B1 decided not to implement microservices because of the cost to train all employees to learn about microservices. This was in a company with about 50 employees. It could be argued that for this reason it is actually a benefit being a small company, since it is easier to train a small number of employees to learn about a new technology.

3.4 The initial migration strategy

Once the knowledge gathering had been completed, a strategy on how extract a microservice from the monolith was created. The strategy includes how to understand the monolith, how to design the architecture, and how to take the first steps when extracting a microservice. The strategy created is an iterative process where the steps above are done in iterations. The rationale behind the steps in the strategy are based on the findings and challenges from Section 3.3. This strategy was later used to extract the first microservice in phase two. A visualization of the strategy can be seen in figure 3.1. This strategy changed after we tried it out, explained further in Chapter 4. The final migration strategy can be read in its entirety in Appendix A.

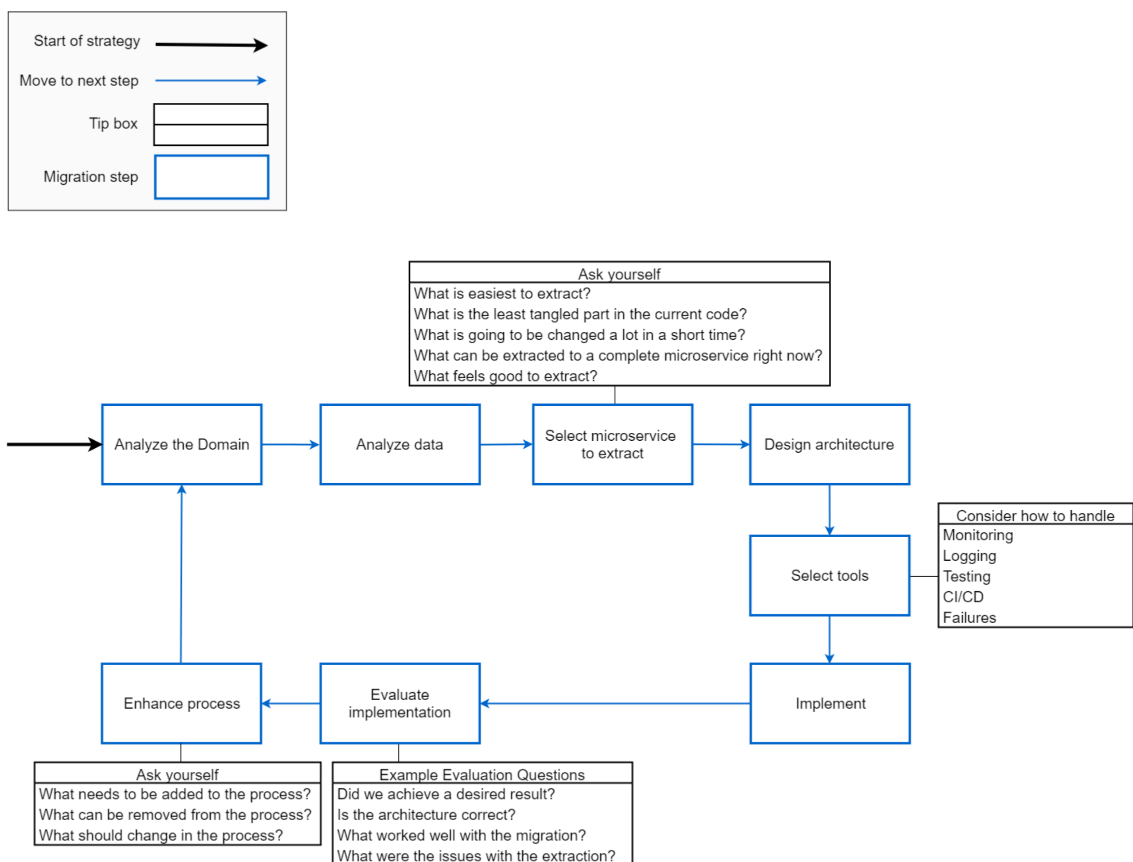

Figure 3.1: The initial migration strategy based on interviews and literature created after the first phase. This strategy will change in upcoming sections. The final migration strategy can be read in its entirety in Appendix A.

3.4.1 Analyze the domain

Description
During this first step of the migration strategy the business domain should be analyzed. Understanding the domain, and preferably visualize it through a diagram, can be of great help during the process of finding a suitable decomposition of the system. The domain analysis should be done in collaboration with developers and domain experts and the discussions should be on a fairly high level and not go into implementation details. Understanding the concept and principles from domain driven design can provide guidance to the team in the process of understanding the domain and defining service boundaries.

The domain analysis can be seen as an up-front task, and will require most effort during the first iteration of the migration strategy. However, the domain model created should be re-visited at each iteration to make sure that it is up to date.

Rationale
This step serves to combat Challenge CH1 from Section 3.3, which highlights the complexity and importance of understanding the domain and defining service boundaries. Even though this process can be complex, it will likely be time is well spent. Finding a suitable decomposition of the services will be beneficial for the migration as a whole. Domain driven design is commonly mentioned and could be used as a framework to help in the process.

It is important that the domain model created during the domain analysis is updated. The domain model will serve as a useful tool during later stages when selecting parts of the system to extract, and must be up to date in order for it to be of help.

Outcome
The team who conducted this step should at the end have a great understanding of the domain, which includes knowing what parts that exists in the domain and what parts in the domain that are related to each other. In the end there should be a visualized and well defined domain model that will help the developers and architectures in the upcoming steps. The domain model will also work as a dictionary and form a unambiguous language, which ensures that everybody is talking about the same thing when discussing the different parts of the system.

3.4.2 Analyze data

Description
This step focuses on understanding the data that exists in the domain, where that data is being used and when the data is updated. This can be done by analyzing the events that certain parts of the system is responsible for and what data that is needed to perform these events.

There exists practices that can be used to define the events and data needed. For example, the concepts of aggregates and domain events from domain driven design could be used. By applying these concepts, it is possible to understand what data is needed and what events causes data to update [Vernon, 2013].

Rationale
As stated in challenge CH2, the challenge of managing decentralized data and communication is closely related to to challenge CH1. Understanding what data that exists in the system is important and is related to the service boundaries. Analyzing the data will provide an opportunity to understand what and where data exists in the domain and how the data travels in the system. This step could be seen as an expansion of the previous step "Analyzing the domain" and a way of understanding what data that exists in the system, and how the data is used in the system.

Outcome
At the end of this step, it should be clear where data belongs in the domain, how this data is updated and where this data can be retrieved. This should also be documented for later use, and some changes to the domain model might be necessary. The outcome will be of value for developers, since developers will likely get a deeper understanding of the domain and what data exists in the system, which will be of help during the implementation phase.

3.4.3 Select microservice to extract

Description
As the domain and data has been analyzed it is time to select what part of the domain to extract

in to a microservice. The domain analysis will be of help during selection process, but it is not always straight forward what to pick. These five questions can act as guidance during the selection.

1. What is the least tangled parts in the code?
2. What is easiest to extract?
3. What can be extracted to a complete microservice right now?
4. What is going to be changed a lot in a short time?
5. What feels good to extract?

Rationale
The questions are all based the solving Challenge CH3 from Section 3.3. The strategy in this challenge recommends starting with something easy and decoupled. Question one and two are based on this recommendation. These questions will mainly be used in the initial iterations of the extractions, since the first iterations will require developers to learn several new tools and work flows. As experience and knowledge regarding microservices is gained, these two questions should be less important to consider.

The strategy also recommends focusing on important parts of the system to avoid spending time and money on building services that cannot or will not be used, which is why question three is included. Question four is important to consider since parts that are frequently changed are likely parts that are refactored often, because they are affected by several things in the system. Continuously fixing these part is often time consuming for the developers, and by extracting these parts they might not be refactored in future changes. This question is probably more important when the developers are more experienced in microservices and ready to tackle a more difficult part to extract.

Finally, question five is based on the last recommendation from the strategy in Challenge CH3. The question could be considered fairly vague, but we would argue that as developers gain knowledge about the domain and the system, the intuition of what makes a good service in the system could be an important factor to consider.

Outcome
After this step is completed, what to extract into a microservice should be clear and concise. If this is the first microservice, it should probably not be too complex, since the main learning objective of the first extractions will be how to work and think about microservices. A complex microservice might take much longer to implement without experience. As the developers gain knowledge and experience, more complex migrations can be probably be done faster and better.

3.4.4 Design architecture

Description
When a microservice has been selected, it is time to start think about the implementation of the microservice. However, before the implementation starts it can be of help to understand the basic architecture of the microservice, since it helps the developers understand how to implement the service and what is needed for it to function. The documentation from the domain and data flow analysis will be useful when analyzing what parts of the system the service needs to interact with and how data traverse in the system.

Rationale
Many of the challenges mentioned in Section 3.3 does not provide any specific strategies that to use, but rather that it is important to consider them and face them. For example, challenge CH2 discusses the importance and complexity of selecting communication protocol, and challenge CH5

discusses fault tolerance. There are several uncertainties related to exactly how these challenges should solved. Designing an architecture will force the architect and development team to consider the challenges and try to combat them in the best way possible.

Outcome
The outcome of the step should be a clear and usable architecture that the developers can rely on when implementing the new microservice. This step should also provide an opportunity for developers and architects to consider and understand how the service interacts with other services, and what different options exists to combat the challenges of working with microservices.

3.4.5 Select tools

Description
There are several tools that can be of help when combating the technical challenges when developing microservices. Therefore, it could be helpful to take a step back to explore and consider what tools that may fit the application that is being developed. For example, how will the microservice handle failures? How will it be monitored? Will it be implemented into continuous integration and delivery pipelines, and how? How will events be logged? How will the features be tested, and should the test be written before, after or during implementation? These questions are probably mostly useful in the beginning of the migration to microservices, and after a couple of iterations of the process the tools have likely been selected and implemented.

Rationale
Challenge CH2 states that it is important to consider communication, challenge CH4 that it is important to implement automations and challenge CH5 that it is important to handle failures and to implement monitoring and logging. How these are implemented and in what order is not clear, and it the order might not matter, but it should be done. This step is an opportunity to explore tools and get familiar with them.

Outcome
Most of the uncertainties around the implementation should be clear after this step, or there should at least be some ideas on how to solve the uncertainties during implementation. There will likely still be issues with implementation regardless of how much preparatory work is done. However, it can be cost effective to think about the uncertainties before implementation starts, in order to not implement something that will not be used, or forget to implement something vital.

3.4.6 Implement

Description
Once everything is considered and all the preparatory work is completed, it is time to implement. If this is the first microservice to be implemented, it will likely take more time due to environment configurations such as setting up a cluster, setting up pipelines and setting up event buses etcetera. However, this step is very similar to developing any new feature to a system, and while it might take some time, it should not be a big problem if the preparatory work is properly done.

Rationale
At this stage all preparatory work should be completed and the developing team should be ready to implement everything.

Outcome
The outcome of this phase should be a functional microservice that can be validated against any requirements and the architecture.

3.4.7 Evaluate implementation

Description
In this step the microservice is evaluated and compared to what was envisioned at the start, and why any pivots from the vision has occurred. There could also be some general discussions on if the achieved result is what was expected from the start.

It is also important to reflect on difficulties during the implementation and how to avoid them in the future, but also acknowledge what worked well and ensure that the things that worked well are present in upcoming iterations.

Rationale
This step was highly influenced by the sprint retrospective from the agile framework Scrum, where the goal is for the team to identify improvements to implement in upcoming iterations [Scrum.org, 2018]. It is important to improve and learn from eventual mistakes or new insights that occurred during implementation, and having the discussion as a part of the process forces the team to reflect best practices.

Outcome
Everyone in the team should agree on that the microservice is well implemented, and hopefully new knowledge has been gained that will improve future extractions.

3.4.8 Enhance process

Description
In this final step the process itself is analyzed and adapted to better suit the team and how they prefer to work. There might be a step missing that needs to be added, or there might be a step in the process that is redundant. Discuss the work flow in a meeting and adapt it to better fit the team.

Rationale
This strategy has been created by us and has not been verified by external sources. Whether or not it is applicable to all companies is questionable, and therefore the process itself should be reflected upon, and adapted to better fit the working process of those who use it. By reflecting on the process after each iteration and improving it, it will be more efficient and yield a better result as more iterations are completed.

Outcome
The outcome of this step is either an updated migration strategy, or a consensus that the strategy works well and should be used as it is.

4

Phase 2: Conducting the Migration Strategy

In this phase the migration strategy, created in phase one, was conducted and validated on whether or not it is accurate and if any changes are needed to improve it. In this chapter, the result from each part of the process is showcased step by step. The initial migration strategy can be seen in Figure 3.1.

4.1 Analyzing the domain

A couple of things were done to analyze Eliqs domain. First, to gain a better understanding of Eliq and their company goals a meeting with the sale person at Eliq took place. Once that was completed, an event storming workshop was held with two developers a Eliq together with another person from sales. Finally, the domain model generated was discussed and built upon in iterations with continuous feedback from the people at Eliq.

The sales person presented Eliq during the meeting as he would to any client, and showed a presentation on what they do and what their product is. The meeting was very insightful and gave us some a better understanding of Eliq. Two primary stakeholders for the new system were identified during this meeting; the utility companies and the utility companies' customers (called users by Eliq). After having these identified, the desires for the two stakeholders were discussed. For example, utility companies wants to have satisfied, loyal customers and have more insights into customer engagement. On the other hand, the users wants to understand their energy bills and save money. The common thing both the clients and the users wants are insights, even though they want different types of insights. From this it was concluded that Eliqs product, in its most basic form, takes data, generates insights from it, and delivers it to users and clients. Therefore, any domain model created would have to visualize these areas and how they are related.

The next step in analyzing Eliqs domain was holding an event storming workshop. Approximately 50 events were generated during the initial brainstorming part of event storming. After that the actual event storming began. It proved to be very difficult to keep the event storming at a high level and not discuss implementation details. After a while, we gave up on event storming and instead the workshop formed into discussing the actual domain model. An attempt was made to simply draw the domain model on a whiteboard followed by discussions of what each sub-domain is for and how it interacts with the rest of the system. This turned out to be much more insightful than discussions about specific events. In the end, the overall workshop was semi successful, a domain model had been created, but not through event storming.

The domain model from the event storming workshop was then refined with some additional feedback from the developers, and can be seen in Figure 4.1. There are three main areas, Location, Client and Analytics. Initially, user was one of the main areas instead of location, but after further

discussions it was concluded that the location has a user, and it is the location, such as a house, that has all user related data connected to it. The Client is connected to what the Client can do and what they control in the system. Lastly the analytics is a main focus for Eliq, since they collect data, interpreter it and creates different reports and comparisons based on it. The analytic in the domain is connected to all the different things Eliq does with data, such as importing from external system and creating business insights (BI).

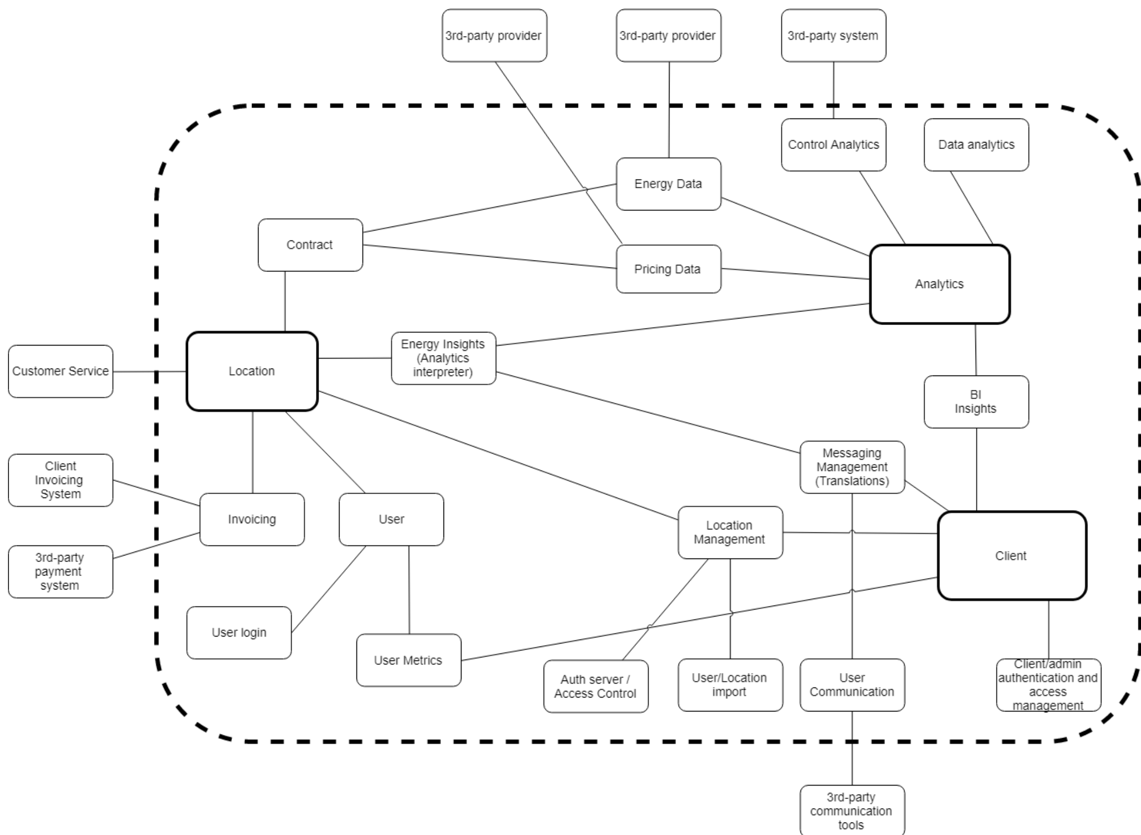

Figure 4.1: The domain model created during the event storming session, with a few additions and changes after the workshop. Everything within the dotted line is part of the internal system, and everything outside are third party systems or similar.

4.2 Selecting microservice to extract

According to the migration strategy in phase one, as seen in Figure 3.1, the next step in the process was to analyze data in the domain. However, after some considerations it was realized that it would take a lot of time to do this for the entire system. Therefore, the second and third step were switched, and it was decided that what to extract had to be selected from the domain model and then the data should only be analyzed for that particular sub-domain.

What microservice to extract was decided during a meeting with the lead developer at Eliq. During the meeting the questions from Section 2.2.2 were asked and there were some general discussions on what the best microservice to start with was. It was concluded that the user communication sub-domain from the domain model in figure 4.1 would be a good start, since it was decoupled from everything else, can be released immediately once it was finished and was loosely attached to the rest of the code in the monolith.

The general features for the microservice that was selected for extraction, the user communication service, was also discussed during the meeting. It was said that it should handle messages to the end user, such as email, push notifications and sms. It should store the data needed for it to send each message type, and be responsible for push notification data such as certificates and user push tokens, since that data should not be necessary anywhere else in the system.

4.3 Analyzing data

As mentioned in Section 2.2.3, the purpose of this step changed when the steps were flipped. Instead of analyze the data for the entire domain only the data for the selected service was analyzed. As explained in Section 2.2.3, this was done by an investigation of what use cases the selected service was responsible was done and requirements were created for each of them. In addition to defining an explanation of the requirements, the data needed to perform the use case were analyzed. For example, one use case was Given that a message is marked as 'sms', when the message arrives, then send it as an sms". The data needed to perform this action was "Message body", "User phone number", "Country code" and "Sender name". Initially, 12 requirements were created, but after consulting with the lead developer one of the requirements was deemed to be out of scope.

Once the requirements were complete, a database schema based on the data required to fulfill the requirements was created, and can be seen in figure 4.2. In this schema, all the data needed to send sms, push notifications and emails for any user in the system is included.

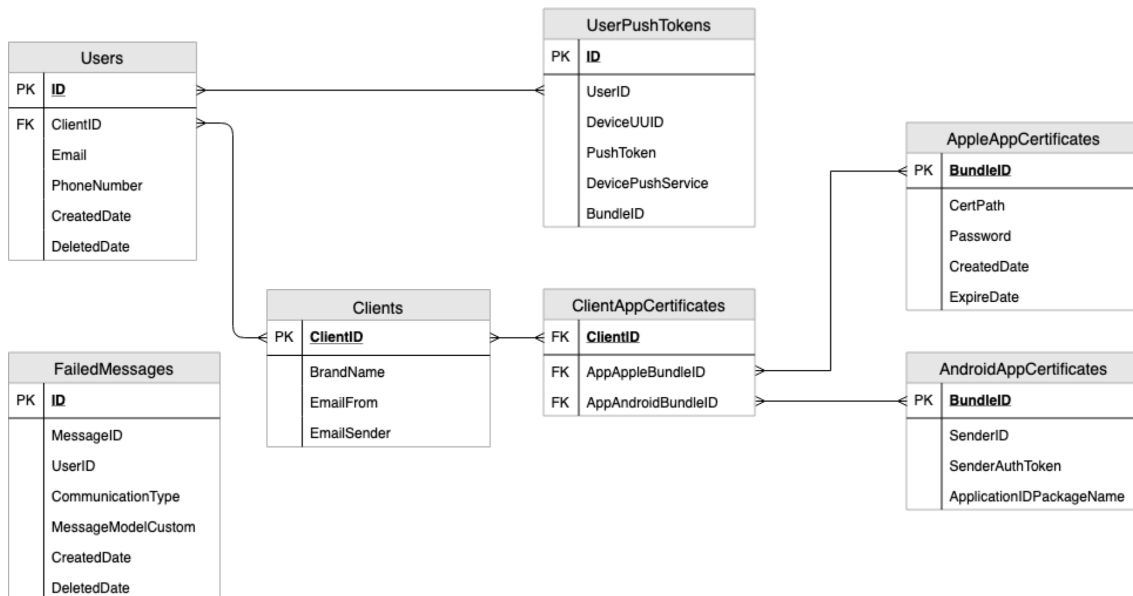

Figure 4.2: Database schema for the microservice user communication, which includes all the data that the microservice needs to send sms, push notifications and emails.

4.4 Designing the architecture

Designing the architecture was fairly straight forward since the requirements were well defined from analyzing the data. The architecture can be seen in figure 4.3.

There are two users who can interact with the system, the end user, simply called user at Eliq, and the client (the utility companies). These user can interact both through a mobile application and

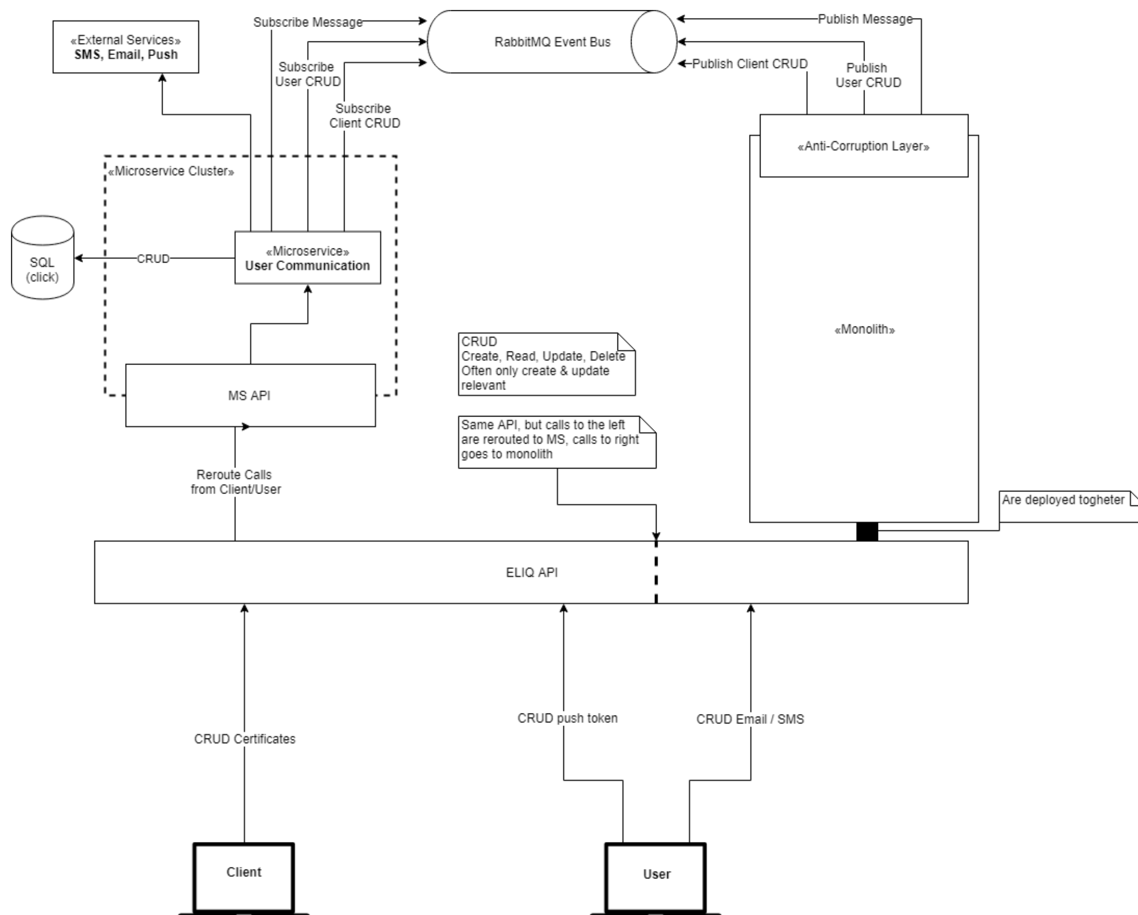

Figure 4.3: The architecture for the microservice user communication.

through a website, but they are only represented with a laptop in the architecture. Whenever a user changes their email or phone number (CRUD Email/SMS in the figure), that call goes to the Eliq API. The Eliq API and the monolith are hosted as the same entity, but for understandability they are separated in the design. The monolith is also interested in these changes, thus the call goes through the monolith to the right and the changes are stored where they are needed. Once the monolith has handled the call, it forwards it to the anti-corruption layer that packages it into an event and sends it to the event bus. The anti-corruption layer ensures that there is no mismatch in how the microservice receives the information compared to how the monolith sends it. The dotted square to the left represents the microservice cluster where all future microservices will exists. Currently, only the user communication microservice exists in the cluster. Once this service is ready to handle a new event, it polls the user CRUD event, stores the relevant changes in its database and waits for new events to handle. If a message is created in the system, it also goes through the anti-corruption layer into the event bus and is handled properly by the microservice. Depending on if it is an sms, a push notification or an email, it uses the external provider it needs to send the corresponding message.

The other flow that can occur is if the user changes their phone and the push token has to be updated. The monolith is not interested in this event since it is only relevant for the user communication microservice. Therefore, the call is simply forwarded to the microservice API and subsequently forwarded to the microservice, who updates its database with the new token for that user. The SQL database to the left is outside the cluster, but is connected only to the user communication microservice. No other service can access this database.

4.5 Selecting tools

Selecting the tools to use, understanding when external tools are needed and understanding how the tools work took longer than expected. It was known that the user communication microservice would need access to an event bus, a database and third party providers to send messages of different types, which was clear from the architecture.

It was also decided that Azure service fabric would be used as microservice cluster, and that RabbitMQ[1] would be used as event bus. These were picked mostly because Eliqs current backend was already implemented using .Net, Microsoft Azure[2] and RabbitMQ. However, Azure service fabric was something new and had to be researched and some tutorials on how to use service fabric were followed. This greatly helped in understanding how it worked, but since it is a complex system there were always more questions. There were tools in Azure service fabric that helped with logging and monitoring, and it was decided that those would be used.

Since we learned from phase one about the importance of continuous integration, we knew that we needed a DevOps environment. The choice for this was Azure DevOps which supports Azure service fabric and is easy to set up. It was also decided that the unit tests should be written during the implementation after new features were added, but it was unclear what testing framework to use at this point.

4.6 Implementing

A couple of weeks was spent on implementing the microservice. Following the requirements and the architecture, it was relatively easy to put the code together. A lot of the code was already written in the monolith, and while it was not copied completely, it was often reused. No code in the monolith was removed, but some parts had to be added, particularly the anti-corruption layer and the parts calling on this layer to generate events to put in the queue. It was decided that it was safer to keep the code in the monolith and remove it at a later date, since removing it might break the code in the monolith as it was not sufficiently tested.

26 tests were written for the code in the new microservice using xUnit[3], a testing environment for .Net. These test mostly covered sending different types of messages, both correctly but also with incorrect emails and phone number, to validate that the microservice can handle bad data.

A lot of time was spent on configurations of different things. For example, setting up the cluster at Azure involved a lot of decisions, most of which had to be taken by the lead developer since they often entailed a cost. For example, choosing what virtual machine to use, which includes picking how much ram and how big the disk size should be, was hard without experience.

The DevOps pipelines for continuous integration was also set up. Setting up the basic pipeline at Azure DevOps was easy, but making the test work took a bit of effort. In the beginning, NUnit tests were used. However, these proved to be very difficult to set up in the pipeline, and the test framework was changed to xUnit, which worked fine before the database was implemented. Initially the database used was only hosted locally, but this had the consequence that the test only worked locally as well, since they used testing data stored in the database. To easier work with a local database, the database was switched from a database first approach to a code first approach, where the database was generated from the models and not vice versa. After some more configurations with the pipeline, the tests started to work in the pipeline again, and the continuous integration was successfully set up.

[1] RabbitMQ: https://www.rabbitmq.com/
[2] Microsoft Azure: https://azure.microsoft.com/sv-se/overview/what-is-azure/
[3] xUnit: https://xunit.net/

Another time consuming implementation was configuring the event bus. Eliq has previously used RabbitMQ to handle events, therefore it was decided that the new microservice should also use RabbitMQ. The goal with the implementation was that the configuration of the event bus in future microservices should be as painless as possible. Therefore, a local library was created that handled the set up of the queue, with code for publishing and consuming from the queue. This was more complex than expected and took a couple of days to get right. However, the outcome was satisfactory and each new microservice needs only a minimum amount of code to consume and publish to the event bus.

4.7 Evaluating the result

The next step in the process was to evaluate the result of the implementation. However, the evaluation became more of an iterative process where continuous meetings with the developers at Eliq were held. This was to ensure that what was developed was what Eliq envisioned. Once the implementation phase began to wrap up, one of these meetings led to new insights into what the microservice should do and how it should function. This section explains what these insights were and why they occurred, followed by a final evaluation of the implemented microservice.

Throughout the implementation some tests were written to ensure that the microservice functioned correctly. 26 tests were written and they all passed when the microservice had been fully implemented. Unfortunately we could not do any performance testing or stress testing on the microservice, since that would have led to us sending out thousands of emails, sms or push notifications. We did, however, find that the microservice could handle high load of data on the event bus. We sent 15 000 messages to the event bus containing all the data needed for the microservice to operate, that the microservice polled and populated into it's database. This took a couple of minutes to complete but was successful.

4.7.1 Changing the requirements

During a short status meeting with the lead developer at Eliq at the end of the implementation, the microservice at its current state was showcased and discussed. This was not the final evaluation, but more a check up meeting to look at the progress so far. The microservice was at this stage largely done according to the requirements created earlier. During the meeting, we realized that the microservice was missing a few key features. While these new features were easy to implement, further issues with the microservice came into light.

The problem with the microservice was that it handled and stored an unnecessary amount of data. When we started to think about how we were going to deploy it and hook it up to the current monolithic application, we realized that the data this service needed was also needed in the monolith for other purposes. We then questioned whether or not it was this service responsibility to store this data or if it should be fetched or sent together whenever a call to it was made. For example, the service stored a lot of data about the user, such as email and phone number, data that several other parts of the system required as well. However, it was deemed out of scope, during the elicitation of requirements, that this service should check for user and client permissions on weather it was allowed to send messages or not to a specific user. While discussing this, it was also realized that the service that had to check the permission would likely also need to store user data. Therefore two services would have to store data about the users and clients to send messages, which seemed unnecessary. We still wanted this service to only handle sending messages, but decided that it should only store data that no other service in the system needs to know about, which is push notification tokens and push notification certificates. All other data should be sent to the microservice together with the message. This also meant that the database schema in Figure 4.2 was no longer valid and had to be updated to accommodate the changes. The data that was

unnecessary to store was then removed and the code was refactored to instead receive this data together with the rest of the message.

While this did not require any further domain analysis or changes in the architecture, it did impact the requirements and how the microservice was implemented in code. This meant that after this meeting we went back to the implementation step and changed parts of the code to fit this new vision of the microservice. It was pretty easy to refactor the microservice since the code base was relatively small and well structured, which indicates that the microservice was well implemented.

4.7.2 Evaluating the requirements

After the microservice was updated, which took about 24 working hours to do after the first evaluation, the implemented microservice was compared to the initial requirements. Since the requirements changed during the meeting with the lead developer explained in the previous section, it was obvious that a lot of the initial requirements would not be fully implemented. Five of the 12 requirements were implemented as they were expected to be. Six requirements were removed completely, the reason was that they mostly evolved around updating user and client data. One requirement should still be implemented but is not necessary for the microservice to function. This requirement will therefore be on hold for now. Additionally, three new requirements has been implemented which were not planned from the start.

The architecture created was implemented almost exactly as the image in figure 4.3 suggests. However, the communication messages to the event bus such as "subscribe user crud" were not implemented since they were out of scope, but a few other messages was added. The architecture was mostly helpful during its own creation, since it gave us insights into how it should look like and what would be needed for the microservice to function, but it was not really used as a tool during implementation.

4.7.3 Evaluation by Eliqs' lead developer

An interview with Eliqs' lead developer was held after the requirements had been evaluated. The interview aimed to understand if the result of the implementation were satisfactory and what the lead developer first envisioned, and if he still felt comfortable with migrating. The result of this evaluation may be somewhat biased since the lead developer participated in the development of the microservice. However, the implementation was mainly done by us, and since Eliq is a small company there was not anybody else knowledgeable enough in microservices to answer the questions properly.

Selection and implementation of the microservice
Many interesting things where brought up during the interview. Firstly the result of the implementation was discussed and whether the selected microservice was a good choice. Since the microservice scope continuously decreased, the end result was not what the lead developer first envisioned. He did, however, say that it is not necessarily a bad thing but rather shows that it is hard to define exactly what the service should do. In regards to the selection of the microservice, a few interesting points were brought up. The low coupling in the code was good since the extraction of code were relatively easy. However, communicating with users is an important function which complicated the process of changing from the old system to the new service in production. He also means that is is hard to test and load test that service since it handles end-user communication.

Uncertainties regarding microservices
The second part of the interview aimed to understand if there are any uncertainties with working with microservices. In the survey sent out at the beginning of the phase, the lead developer expressed concerns about monitoring the application. At this stage, he felt more comfortable knowing that there are tools, such as dashboards and logging triggers, that can be used to monitor

the system. He was also uncertain about performance when using microservices, but feels more confident now. He is still uncertain about the performance in a microservice cluster, but is certain that Eliq will find a solution.

Continuing the migration
The lead developer still thinks that striving towards microservices would be beneficial for Eliq. He is confident that the working process can be improved when using microservices since the tasks can be easily distributed. He also stated that he does not think that Eliq needs to do a full-fledge microservice architecture. Instead of dividing everything into separate microservices he believes that a first step could be to separate code in to libraries. These libraries could later be moved to a microservice if needed.

Selecting the next service
Finally, we were curious about what Eliq would like to extract next. The lead developer was not certain that the next microservice will focus on extracting a service from the monolithic system, instead they might develop a new feature. Eliq are also frequently working with integrations towards utility companies and their systems, such as fetching energy data, pricing data etcetera. Eliq also have scheduled services running in today's system to handle these feature. The lead developer said that the next migration might focus on migrating one of those scheduled services.

4.8 Enhancing the process

Overall the migration strategy was successful. The developers at Eliq are pleased with the result and will continue to work with microservices and adapt their system. However, everything did not go according to plan and several issues occurred during the migration. This section goes through each step of the process and discusses what worked well and what did not work, and if any changes to the migration strategy had to be done in each step. A summary of the major tasks done during the migration and what their outcome was can be read in Table 4.1. The new migration strategy can be seen in figure 4.4 and can be read in its entirety in appendix A.

Description	**Outcome**	**Strategy changes**
Analyzing the domain	The meeting with sales person was very successful and is highly recommended. Event storming led to good discussions about the domain but the event storming itself failed.	The step was kept in the process without changes.
Analyzing data	Analyzing data for the entire system would have taken too much time, and therefore we began by selecting the next microservice to extract. Also, more time could have been spent considering data and what the service should do, since we had to refactor parts later on.	Switched place with "Selecting microservice to extract".
Selecting microservice to extract	The questions worked well and led to a good selection of what to microservice to extract first. However, the selection should be done before analyzing data.	Switched place with the step "Analyzing data".
Designing architecture	Designing the architecture helped us understand how the microservice should function. However, during implementation the architecture was hardly used. We still believe it is a good idea to design the architecture due to the discussions during its creation.	The step was kept in the process without changes.
Selecting tools	Considering and trying out tools gave us a deeper understanding in how microservices can be implemented. A lot of tools were replaced during the implementation step. Implementing and selecting tools is much more of an iterative process.	Add line back to "select tools" from "implement"
Implementing	Implementing went well but was very time consuming, and configuring tools and frameworks took much more time than anticipated. During evaluation we found that some changes were required, and we realized that evaluating and implementing is an iterative process.	Add iterative lines to "select tools" and "evaluation"
Evaluating the result	We realized that the implementation had to be improved during evaluation. We went back to implementing, improved what we found during evaluation, evaluated again and realized that the microservice was better implemented.	Add line back to "implement" from "evaluating"
Enhancing the process	As is evident in this section, several improvements has been done to the migration strategy which validated that this step is important.	The step was kept in the process without changes.

Table 4.1: A table summarizing all major tasks conducted during the migration, what the outcome of each task was and how it affected the migration strategy.

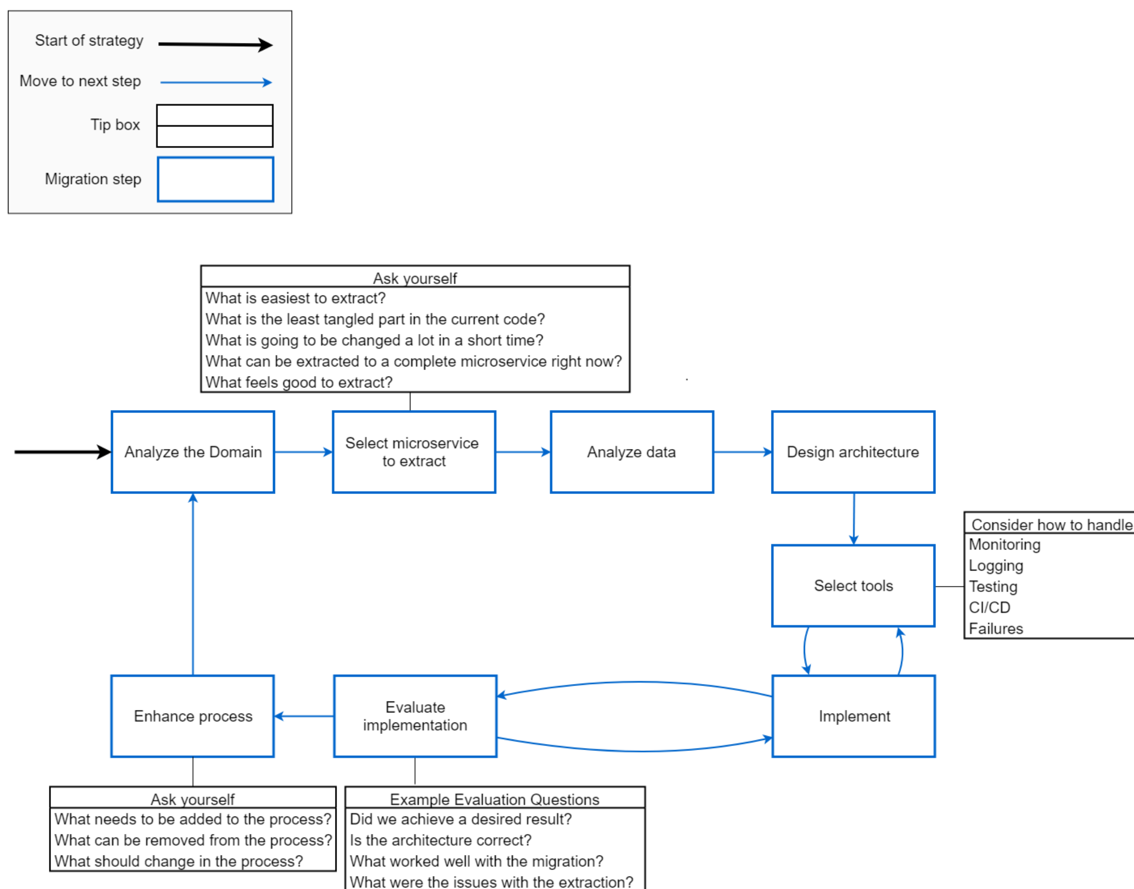

Figure 4.4: The final migration strategy based on interviews and literature, and the experiences gained from phase 2. The final migration strategy can be read in its entirety in Appendix A.

The time spent on each task in the strategy, up until evaluation, was measured for each week and can be viewed in Table 4.2. Every heading in the table is a task in the migration strategy, except for configure. The configurations of different tools and frameworks took place during the implementation, but since configuring was the second most time consuming activity, we wanted to highlight it in the table. As can be seen in the table, implementing, configuring and analyzing the domain were the most time consuming activities. This was not that surprising to us, even if we did not anticipate that configuring would take that long. We also spent quite a bit of time on selecting tools, but that was mostly because we also included testing the tools and trying out tutorials for them in this step, which also helped us in better understanding how to work with microservices.

Note that time has also been spent on other things during the project such as writing report, attending meetings with supervisors etcetera, but this table focuses only on the tasks in the migration strategy.

Week	Total	Analyze domain	Analyze data	Select microservice	Design Architecture	Select tools	Implement	Configure
7	56	56	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
9	22	0	0	0	0	22	0	0
10	80	2	28	4	12	0	28	6
11	45	0	0	0	0	19	10	16
12	28	0	0	0	0	8	8	12
13	40	0	0	0	0	14	21	5
14	0	0	0	0	0	0	0	0
15	19	0	0	0	0	0	12	7
16	52	0	0	0	0	0	26	26
17	2	0	0	0	0	0	2	0
Total	344	58	28	4	12	63	107	72

Table 4.2: Table displaying the time, in person hours, it took to complete each task in the initial migration strategy, apart from evaluating and enhancing the process. Configure is a sub task of implement, but was displayed as its own task due to the time it took to configure different things.

The rest of this section goes into detail about each step of the migration strategy and discusses what went well and what did not, and how it affected the migration strategy.

4.8.1 Analyze the domain

The first step of the migration strategy was to analyze the domain and understand what parts that exists in the domain. The process was done in two steps; understanding from a business perspective what Eliq is selling, and analyzing the domain using event storming.

Meeting with sales person
As mentioned in section 4.1 the domain analysis started with a meeting with a sales person from Eliq. The outcome from this meeting was very helpful in regards to understanding the product at a higher level. This allowed us to understand the big picture and highlighted the importance of including people from various areas in the organization during the domain analysis. If there is little knowledge about the domain in the development team, such as in cases where there are new consultants or new employees, a meeting like this with a sales person is recommended by us, since it helps in understanding the product at a higher level without going into implementation details.

Event storming workshop
The event storming was not successful. The main reason for that was probably due to lack of experience of the facilitator, which in this case were ourselves. It was hard to control the level of details that were discussed which quickly resulted in discussions about implementation details. More experience is probably required by the facilitators in order to deal with such situations. As described in Section 4.1, the domain model was instead created through discussions about the domain and drawings on a whiteboard, using the events that was brainstormed at the beginning of the workshop. The level of details was easier to control here and it resulted in a domain model. In this sense, the workshop could be considered successful, even though the actual event storming did not go as planned.

After the workshop, we realized that it was hard to evaluate whether the domain model was good or bad. There does not seem to be any way of making sure that the model is correct, and the validity of the model is subjective. However, some validations was done by analyzing dependencies and responsibilities for each part in the domain model. Other than that the model was compared to other examples such as the example in Vaughn Vernon book [Vernon, 2013] and the Microsoft example of a domain model [Microsoft, 2018]. We would argue that Eliq's domain is more complex than their examples, but it still provided extra confidence that the domain model was good enough.

4.8.2 Select microservice to extract

As can be seen in Figure 4.4, "Select microservice to extract" has switched place with "Analyze data" compared to the old strategy (visible in Figure 3.1). This switch was mostly due to time constraints, where we quickly realized that analyzing the data for the entire system would take too much time. Eliq is a small company where the developers time is valuable, which is probably true for most companies. By switching these two steps, the time required to analyze the data flow is reduced and more feasible, especially when you are new to the system. This does, however, contradict one of the findings that were made regarding the challenges (CH1, CH2) from the previous phase (Section 3.3), which states that it is important to analyze the data in combination with defining service boundaries. By switching these two steps, the data is not properly analyzed in combination with the full domain analysis, which introduces the risk of defining poor service boundaries. However, during the domain analysis it was inevitable to not discuss and understand on a high level where certain data belongs. This high level understanding of the data could be enough for the big picture of the system, and analyzing the data in detail can instead be done on specific parts.

The selection of what to extract was done together with the lead developer at Eliq. As mentioned in Section 4.2, the microservices selected was the user communication service based on the questions from the migration strategy. During the implementation it was proven to be a good selection, since it allowed us to get familiar with the tools and set up the underlying infrastructure. We were new to microservices when we began this project, which meant that we had to learn about new tools and frameworks. We would therefore argue that it was a good choice to focus on an easy part of the system during the initial phase of the extraction.

Having a service that handles three different types of messages could be considered a “macroservice” instead of a microservice. As discussed in phase one, there is no definite measure of what defines a microservice. According to Microsoft it is more important that the microservice serves a purpose rather than being small [Microsoft, 2018]. This service does serve a purpose; it sends notifications. However, from another perspective it serves three purposes; it sends emails, sms and push notifications. Some might argue that this should be three separate services. At this stage the decision was made to combine these three message types into one service, based on the idea of “go macro, then micro” [Dehghani, Zhamak, 2018].

All in all, the questions from the migration strategy were successful in finding a suitable microservice to start with. While some additional questions could be added to ensure that the next selection is the best possible at any given time, the questions as they were before the migration were deemed to still be relevant and good enough, which is why they are the same in the final strategy.

4.8.3 Analyze data

Analyzing data and designing the architecture was fairly straight forward, largely because the in depth domain analysis. By documenting the use cases and data needed, it was easy to communicate with the lead developer at Eliq make sure that we had a unified understanding of what the service responsibility was. However, the use cases were hardly used during implementation. We felt that we had a clear picture and idea of what the service should do and how it should be implemented, and the use cases where used more as guidance to ensure that everything was implemented. We still think that creating use cases and analyzing data is important, since it made us understand how to implement, even if it was not used much later in the process.

When the microservice had been implemented and was being evaluated, it was discovered that some of the use cases where out of scope, and some were missing. This lead to us refactoring the microservice in order to accommodate these changes. Perhaps more time should have been spent analyzing the exact use cases for the microservice, but the changes were implemented in only a weeks time and did not have major impacts on the structure of the code. It might be that even if

we spent several additional days discussing use cases, these new insights would still not have come to us until after implementing.

4.8.4 Designing architecture

The architecture served a similar purpose as the use cases, in the sense that they helped in understanding how the microservice should function and how it should interact with the other parts of the system. The architecture was mainly used as a tool during discussions, and was not used much during implementation. However, the implemented architecture is very similar to what was drawn. Perhaps the architecture for this particular microservice was very easy to understand, which meant that we had it memorized and did not need to look at it. In any case, we still think that designing the architecture helped us to understand how the microservice should interact with the rest of the system. We therefore recommend anyone who migrates into microservices to create the architecture before implementing the next microservice.

4.8.5 Selecting tools

Selecting what tools to use was much more time consuming that initially thought. In order to implement the microservices, that infrastructure had to be set up. This included setting up cluster management in Azure service fabric, CI/CD pipelines in Azure DevOps, configuring RabbitMQ for communication etcetera. This required us to learn several new tools and work flows which was more time consuming than we first thought. Finding appropriate tools, best practices and how to configure them took a lot of time, as can be seen in Table 4.2.

During the implementation it proved to not be feasible to decide all tools and frameworks that were to be used before implementing. Looking at Table 4.2, it is clear that most of the time selecting and exploring tools was spent in parallel with implementation. Implementation and selecting tools can not be completely separate, and it makes sense when implementing to do it in parallel. This is why there is an arrow going back and forward between "implement" and "select tools" in the new migrations strategy (Figure 4.4). We would still argue that the things to consider, such as how to test and how to monitor the services etcetera, should be thought about before implementing, but selecting the exact tools can be done during implementation.

4.8.6 Implement

When we implemented, we wanted to set up an infrastructure that could be reused by other services. Since this was the first microservice in the extraction, we also wanted the service to serve as a proof of concept that could be used as inspiration during later phases of extractions. We therefore wanted to set up the tools and frameworks in the best way we could, in order to generalize them and be able to reuse them in other services with little configuration. This meant that a lot of time was spent on configuring the tools, which can be seen in Table 4.2. This proved later on to be time well spent, since the developers at Eliq were pleased with the structure and thought that adding new microservices should be easy. This configuration of the infrastructure is also most time consuming when implementing the first microservice, and time spent on configurations for future migrations will hopefully be much less.

While the implementation was very time consuming, it did result in a functioning microservice, and the lead developer said during the evaluation (Section 4.7) that Eliq will continue to work with microservice.

4.8.7 Evaluate the result

During the evaluation of the microservice, it was concluded that the microservices that was initially created did not have a clear and concise purpose and that some requirements were missing. These requirements were then added and some additional changes on how the microservices functions were implemented. This made us realize that it will probably be common to go back and fourth between implementing and evaluating, and thus an arrow moving back to implementation from the evaluation was included in the migration strategy diagram. This back and forward approach fits well into the agile work flow where what is being implemented should constantly be evaluated by the stakeholders [Campbell, 2018].

Generally the evaluation went well, and we think it is an important step in extracting a microservice. Evaluating provided us with valuable insights, helped us in validating that what we had implemented was correct and helped the developers at Eliq understand what we had accomplished and if it was useful for them.

4.8.8 Enhancing the process

Enhancing the process was also deemed to be a valuable step and should be kept in the migration strategy, since it helped us improving the migration strategy. However, the enhancing the process in this thesis is very in depth and analyzes every task done during the migration in detail. This was done since the migration strategy had not been tested at all before and we wanted to ensure that it is a valid strategy. When enhancing the process after each migration outside of this thesis, it can likely be done in a meeting where only the most relevant tasks are discussed.

5 Discussion

This project aimed at answering three research questions (***RQ1-3:***). In this Chapter the research questions are expanded upon and answered based on the knowledge gathered throughout the thesis. Additionally, the lessons learned from migrating a microservices are presented, as well as the threats to validity to this research together. Finally, any future research that could be interesting in regards to microservices and our migration strategy are discussed.

5.1 Answering the research questions

RQ1: *What strategies have industries applied when moving from a monolithic application to a microservice application, and what are the results of these strategies?*
In the beginning of the thesis we found that there were several challenges when migrating to microservices. From a first exploration of literature we found suggestions and strategies of how to face the challenges, but it was not clear how industry handles the challenges and if they use the suggestions from literature, or if they have developed other solutions. This research question aimed to clarify that and understand what strategies industry uses.

To understand what strategies that are used in industry, we first needed to understand what challenges that should to be addressed. To answer this, we conducted a literature review together with interviews and we found several challenges and strategies on how to handle the challenges. The interviewees were people who worked, or had previously worked, with microservices. A few experience reports were read, but the primary source of information about strategies industry used were based on the interviews. The result from this can be read in Appendix B, were we summarized the challenges together with the strategies to face them.

Our general perception from the interviews is that there are no specific strategies for how to solve the challenges. The strategy used to face certain challenges is based on intuition and what seems to be a good solution for the development team at the moment. Despite this, the result for the interviewees who migrated to microservices seemed to be good and they were happy with the result. From this we conclude that the exact strategy used is not necessarily important, but rather that the development team is aware of the challenges that needs to be addressed.

This is reflected in our migration strategy that was created during the project. The migration strategy does not promote any specific strategies to be used, but instead suggests when certain challenges should be addressed during the migration process and recommendations of how to address them.

RQ2: *What are the challenges small companies face when moving from a monolithic application to a microservice application?*
Before we started with this thesis, we found that in literature, microservices are often discussed in the context of large enterprises. During the literature review of this thesis, we also did not find much information about how small companies should work with microservices and if there was any difference. This was therefore one of the main questions we asked the interviewees during the interviews.

As mentioned in Section 3.3, the biggest disadvantage of being a small company is the few number of developers. During a migration from a monolithic application to microservices the team must maintain both the newly developed microservices and the monolithic applications. This process can be hard if, especially if there are only a couple of developers, which could reduce the efficiency of the development process.

In addition, fewer developers also increases the probability that certain experience and skills does not exist in the company. One of our interviewees said that it is important that the team has knowledge in working with supporting functions, such as CI/CD pipelines. In a small company there might not be enough developers with such experience, which could negatively affect the development process.

The challenge of having a few developers is also something that was found from our experience with working with microservices. The developers at Eliq developed and maintained the monolithic application when we implemented the microservice, and did not have much time to work with us. We think it would have been beneficial to have more experience developers involved in the implementation. Since we had no previous experience with microservices it was sometimes hard to know what to do. Time was, therefore, spent on understanding different third party system such as RabbitMQ and Azure Service Fabric. The process of understanding and configuring these system could have been sped up if someone with experience in these system was available to help.

Another thing that might be a challenge for small companies is the cost that comes with hosting and developing microservices. A couple of the interviewees talked about this cost and said that microservices should not be seen as a cheaper way to develop, host and scale. It is also harder to predict the cost of microservices and the price might fluctuate from time to time, since one of the key feature of microservices is independent and automatic scaling. Money is something that many small companies struggles with, and the cost issue might be a big problem. It is possible to cut the costs by implementing some sub-optimal solutions, but that might take away from the purpose of developing microservices in the first place.

RQ3: *What attributes should be considered when selecting parts of the monolith to extract?*
Several attributes to consider were found during the knowledge gathering in Chapter 3. These attributes formed into the questions in the migration strategy, which are:

1. What is the least tangled parts in the code?
2. What is easiest to extract?
3. What can be extracted to a complete microservice right now?
4. What is going to be changed a lot in a short time?
5. What feels good to extract?

Both question one and question two are based on extracting something easy, especially when you first begin to extract from a monolith to microservices. As discussed in Section 3.3, starting with something easy is important since you should focus primarily on building infrastructure and setting up the environment when you start migrating. These tasks can be challenging in themselves and it might slow down the process if you start with a tangled and complex microservice, even if that service might be the primary reason to build microservices to begin with. Some of the interviewees

also mentioned that dealing with a tangled code base can be difficult. By continuously migrate loosely coupled microservices, other services might be untangled and easier to extract in the future.

Another insight that was gained during the knowledge gathering was the fact that you should not build and implement something that is never used, and therefore you should focus on microservices that can be implemented and used by the current system immediately once they are complete, which is the motivation behind question three. Also, it is important to migrate parts from the monolith that are undergoing continuous change (Question four), since these part hinder developers from delivering value. This may contradict question one since the parts that are always affected by change and are thus in need of change, are likely very tangled in the code. However, successfully migrating these parts of the code will likely make future extractions easier and faster.

One thing we realized during the interviews was that most of the interviewees did not have a structured way of selecting what to migrate next. Very often it is simply a matter of gut feeling, which is why question five is included. When developers gain experience in microservices and how they are developed, they will probably gain an intuition of what fits into a microservices and what will benefit the system. Even if it can be hard to motivate a decision based on some attributes, it can still be a good choice.

The entire migration strategy was conducted in phase two, which meant that we used the questions to figure out what microservice to extract. We asked the lead developer at Eliq, and he had a couple of ideas on what to migrate but decided on extracting the user communication microservice. The motivation behind extracting that particular services was that it was relatively easy to extract and untangled, could be extracted into a complete microservice and deployed, and generally felt good to extract. During the implementation it was proven to be quite easy to extract, and most of the time was spent on building the infrastructure and configuring different third party integrations. However, it was harder to deploy into the existing system than expected. As the lead developers said during the evaluation in Section 4.7, user communication is a very important part of the system and it has to work. It is also hard to stress test since you cannot really send thousands of messages to one phone or email address, which made it hard to ensure that everything worked as it should. If we spent a little bit more time considering question three, we might have realized this before implementing. They will, however, use the microservice for user communication in the future.

It would have been interesting to try out the questions one more time, since the infrastructure is built now. This might have affected the choice of what to extract next. Nevertheless, when the lead developer at Eliq was asked what he would extract next, he said that he would like to focus on small things that are still easy to extract and loosely coupled. Maybe you need a couple of iterations to feel confident in extracting before migrating something more complex.

5.2 Lessons learned from extracting a microservice

During this thesis a microservice was extracted from Eliq's monolith application. The extraction was done using the migration strategy that was created during the first phase (Section 3.4). Initially, three phases were planned for the thesis. One phase to gather knowledge and create the migration strategy, and two phases to try out the strategy and adapt it based on knowledge gained from extracting. During our initial planning four weeks were assigned to phase two. At the end of phase two it was realized that more time was required to complete the implementation and make it production ready. This meant that the third phase had to be skipped in favor of properly conducting the second phase. The reasons for why the time plan was breached together with other interesting insights are discussed in this section. A summary of the lessons learned can be read in Table 5.1

Lessons learned
It is important to visualize documents and diagrams.
It is hard to validate the domain model.
Authority to make decisions is desirable.
Evaluating the implementation can be very insightful.
Setting up the infrastructure takes time and requires experience.
Extract something that can be used right away to avoid deployment delays.
Migrating takes time.

Table 5.1: A table displaying the major lessons learned from migrating a microservice

It is important to visualize documents and diagrams
Many of the steps in our migration strategy, such as “Analyze the domain” and “Design architecture”, aims to visualize and document certain information about the system and the microservices. These visualizations proved to be useful and were a great tools when discussing and communicating with Eliq’s development team. The concept of microservices was new to many employees at Eliq and we think that the visualization provided an opportunity for everyone to get an understanding of how ii could work. The documentation could also serve as a tool when explaining the system to new employees.

It is hard to validate the domain model
The first step of the migration strategy is to analyze the domain and create a domain model, with the aim of finding natural service boundaries in the system. The domain model in our case was created and validated together with a few employees of Eliq, and all agreed that it seemed to represents Eliq’s domain. Even though the domain model gave us great insights in regards to the domain, we found it hard to validate the correctness of it and whether it would be useful in the future. The microservice that was extracted was based on the domain model, but it would be interesting to see how much it helps during later stages of the migration.

Authority to make decisions is desirable
During the implementation of the microservice, several decisions had to be made in regards to the infrastructure of the application. For example, how the microservice cluster and databases should be hosted, and what framework that should be used for centralized logging. Many of these decisions will affect the cost of the application and have to be done by a person with authority to make those decisions, which in our case was Eliq’s lead developer.

Approximately 2 hours every week were initially planned for discussions on the progress of the implementation and to get guidance from the developers at Eliq. The meetings every week mainly focused on making decisions regarding tools and getting approval for setting up resources at the cloud provider. For example deciding size of the virtual machines in the cluster, setting up a database were all things that needed approval by the lead developer, since they are all related to costs. Ideally, the lead developer should have been working together with us one to two days a week. That would have increased the efficiency of the development process since decisions could have been made instantly.

Evaluating the implementation can be very insightful
At the end of the extraction, the implemented microservice was evaluated as described in Section 4.7. It was discovered that the microservice was not as precise and narrow as it could have been, and thus the requirements were rewritten and the microservice was refactored. While this refactoring was not a big issue, it did result in a week of extra work. The evaluation and the refactoring that was done improved the boundaries of the microservice, and we think that it is a good idea to always take time to evaluate the implementation before moving forward.

Setting up the infrastructure takes time and requires experience
As stated above, the time plan for the thesis was breached and it was due to several reasons. However, the root cause for the delay was mostly due to the time it took to set up the required infrastructure for the microservices. For example, cluster management in Azure service fabric, CI/CD pipelines in Azure DevOps, configuring RabbitMQ for communication all had to be implemented and configured. This required us to learn several new tools and work flows which was more time consuming than we first thought. As can be seen in the time table in Section 4.8, configuration and selecting tools took a lot of time. Finding appropriate tools, best practices and how to configure them takes time and it requires knowledge and experience.

One of the reasons for why the setup was time consuming was due to lack of experience. We both have developer experience from industry and university, but have never worked with setting up infrastructure for software systems or worked with microservices before. This resulted in a large amount of time spent on understanding and configuring these tools. However, some of the tools mentioned above, such as RabbitMQ and Azure DevOps, is not microservice specific and can be used by other applications as well. Experienced developers may have experience in working with similar tools and a better understanding of how to use them. We would therefore argue that the efficiency of working with microservices is partly dependent on the skill set and experience of the team.

Extract something that can be used right away to avoid deployment delays
One of the steps in the migration strategy is to select a microservice to extract and the attributes to consider during the selection can be found in Section 3.3. The choice we made (user communication) seemed to be a good start for migrating, since it was fairly easy to develop. However, the functionality in the service serves an important purpose to the application and it is important that it works at all time. This complicated the deployment process and is something that Eliq's lead developer discussed during the evaluation meeting. We would therefore argue that considering if the microservice can be used right away is very important when selecting the first microservices to extract, and something we should have considered more when selecting what to extract.

Migrating takes time
There are several challenges in regards to migrating to microservices and the developers are required to learn about new things. From our experience, this process is time consuming especially, in the beginning when it is all new. We think that it is important to let it take time and to get familiar with everything that comes with microservices.

5.3 Threats to validity

There are several threats to the validity of this paper, ranging from the knowledge gathered to implementation details. This section highlights the biggest threats.

Relatively low number of interviews
One of the bigger threats to validity is the relatively low number of people we interviewed during the knowledge gathering phase. Only five people were interviewed. Three of the interviewees had worked with migrating from a monolith to microservices. Two of the interviewees had worked with developing microservices from scratch, where one of them had also migrated. Lastly, one of the interviewed had only researched microservices professionally but never built and deployed them at a company. While this gave us different perspectives on working with microservices, we still think it would have been beneficial to interview more people. The main reason as to why not more people were interviewed was that is was hard to find people willing to be interviewed. We asked several companies that we knew worked with microservices if there was anyone we could interview, but they either did not respond or did not provide a candidate.

No systematic literature review
Another threat to validity was the fact that we did not do a systematic literature review. Instead we had a snow balling approach where we first look up relevant literature through search engines and libraries, and then used the sources of that literature to further find interesting and relevant information. We believe that the approach we had was good enough, and that a systematic literature review might have yielded some additional sources, but would not have impacted the overall result in a significant way.

Bias when conducting the migration strategy
As is evident, the migration strategy was both created and conducted by us. We knew why each step was part of the process, since we designed the process. This might have introduced some bias, and the migration strategy might have been questioned more if it was conducted by someone who had not previously researched microservices as thorough as we had. We also can not determine whether the strategy is usable by someone else or if it is hard to follow step by step, since we have not had anyone else try out the strategy.

In addition, we did not have time to test the migration strategy twice, and we can therefore not determine if it is applicable to further iterations. One of the core ideas of the strategy is that it is iterative and all steps should be done for each microservice that is extracted, but we did not have time to validate this.

Generalization of migration strategy
The migration strategy has only been applied at Eliq during the extraction of a microservice from their backend. We can not determine if the migration strategy is applicable to other companies, domains or monoliths. In order to generalize the migration strategy, it must be validated and tested in other contexts.

Multiple services are required to validate if the challenges are solved
Several challenges were found during the first phase of the project and is summarized in Appendix B. To some extent, all of these challenges have been faced during the extraction of the microservice. Nevertheless, some of these challenges may not be completely solved since only one service have been created. Even though we have implemented for example logging, communication to propagate data updates, monitoring and CI/CD pipelines, we do not know how this will be affected when there are more services running. It is not possible for us to test whether the microservice cluster can handle partial failure, since there is only one service. Several services are required in order to validate that some of the challenges have successfully been combated.

Microsoft stack
The microservices created in this thesis is mostly implemented using Microsoft's stack, with Microsoft Service Fabric, .Net and Microsoft Application insights as tools to name a few. Since these tools where used, a lot of the information about these tools and how to properly build the infrastructure have been sourced from Microsoft's documentation about microservices. This might have influenced the result, and it is unclear what would have happened if we instead used a completely different technology stack, and for example Amazon Web Services as a cloud provider.

5.4 Further Research

For future research, we think it would be interesting to conduct a bigger study in regards to challenges and strategies companies applies. This study contained of five interviews and in order to generalize the result, more data will need to be collected.

During this thesis only one microservice was extracted. Several updates were made to the migration strategy at the end of the extraction, which were not tested. For future work we think it would be interesting to conduct the migration strategy in several iterations, in order to understand if it is useful in later stages of the migration and to make further improvements to the strategy. Something

we think would be interesting would be to investigate whether or not the domain analysis is useful in further iterations and in that case how the domain model is affected.

In addition, we think it would be interesting to do more research in regards to quality assurance and automations when working with microservices. As found during the literature (Section 3.2.2), it is important that the services have independent lifecycles. We think it would be interesting to investigate how independent lifecycles can be ensured, while still having high quality in the application.

6 Conclusion

Migrating from a monolithic application to microservices is not straightforward and there are several challenges to combat during the migration. This thesis has explored the different challenges and found strategies for them through analyzing literature and conducting interviews. A summary of the most prominent challenges can be found in Appendix B. The biggest challenges found is understanding the domain and defining service boundaries. This means that it is hard to find a suitable decomposition of the system that fits into microservices, and it is hard to know what is a suitable microservice is and how it should interact with the rest of the system. Nevertheless, it is very important to find a good decomposition in order to fully take advantage of the benefits of microservices.

This thesis has also had a focus on small companies and whether it is a disadvantage to being a small companies when developing microservices. The biggest disadvantage seems to be the the cost that comes with microservices and the small amount of developers in the company. Hosting and automatic scaling incurs a cost that is hard to predict and it is often more expensive to host microservices than to host a monolith. This may be an issue for small companies with a tight budget. The small amount of developers can introduce challenges because of lack of experience and time to research new tools. When developing microservices, other technologies and frameworks will be required compared to developing monoliths, and these takes time to fully understand. Apart from these disadvantages, there does seem to be much else. In fact, sometimes it is even an advantage to be a small company. With few employees, the cost of teaching everyone how to work with microservices and how new technologies work might be cheaper. It can also be easier to split responsibilities for the services in a small team where everybody knows each other.

Selecting what to begin extracting can be a bit of a challenge. Before the selection the developers should have a deep understanding of the current system and the domain they are working in. Preferably they have a domain model or similar that visualizes good candidates for microservices in the domain. Once this has been established, a few attributes should be considered when selecting what to extract. The most prominent attributes to consider are "what is easiest to extract", "what is the least tangled part", "what can be implemented into a complete microservice", "what part of the monolith is prone to change" and "what feels good to extract". Some of these are more important for the first migration, such as selecting something easy and untangled. As the developer gain knowledge in how to build microservices, the latter attributes are more important. After several microservices has been migrated, the choice of what to extract is often a "gut feeling" from the developers, and only minor discussion might be needed before making the next choice.

As is evident in the report, a migration strategy has been created and conducted to validate the information gathered in this project. In general the strategy worked well and it led us to successfully migrate a microservice. A few insights were gathered about migrating when we tried it out, and those insights manifested in some changes to the migration strategy. For example, we realized that going back and forth between evaluation and implementation is a good idea, since you might want to change the microservice depending on what was discovered during the evaluation. Unfortunately these changes has not been validated by conducting the migration strategy a second time and developing another microservice, which is something that could be done in future research.

The final migration strategy together with descriptions, rationales and outcomes for each step in the strategy can be read in its entirety in Appendix A.

Migrating from a monolithic application to microservices is very time consuming, and even migrating a single service might take more time than expected. When you initially migrate to microservices, time will be spent on finding and configuring different tools and setting up the infrastructure. However, these things are often only done once, and future migrations may be faster. Not having authority to make certain decisions, particular in cases that the decision incurs a cost, can also waste time. Additionally, if the service is not well defined or well thought out before implementing, refactoring the microservice might be necessary before it can be deployed. These things and similar issues may cause delays when developing microservices.

Overall the developers at Eliq are pleased with the result. They said that they will continue working with microservices and migrate more of their old system into the new system that was developed during this project. In retrospect they would have preferred if we had migrated something that was not as important in the system, since they are not fully confident in that the new microservice will be able to handle the load required. However, they are still confident that the new microservice will be used in production. Having participated in the migration and seen the result of it, the lead developer at Eliq feels more comfortable in building and maintaining microservices going forward.

Bibliography

[Alshuqayran et al., 2016] Alshuqayran, N., Ali, N., and Evans, R. (2016). A Systematic Mapping Study in Microservice Architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE.

[Balalaie et al., 2016] Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2016). Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. pages 201–215. Springer, Cham.

[Campbell, 2018] Campbell, H. (2018). Agile in a Nutshell: An Executive Overview. *https://hackernoon.com/agile-in-a-nutshell-an-executive-overview-db19f3400c1a*. Accessed: 2019-04-25.

[Conway, 1968] Conway, M. E. (1968). How do committees invent. *Datamation*, 14(4):28–31.

[Dehghani, Zhamak, 2018] Dehghani, Zhamak (2018). How to break a Monolith into Microservices. *https://martinfowler.com/articles/break-monolith-into-microservices.html*. Accessed: 2019-02-15.

[Dragoni et al., 2017a] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017a). Microservices: yesterday, today, and tomorrow. Technical report.

[Dragoni et al., 2017b] Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R., and Safina, L. (2017b). Microservices: How To Make Your Application Scale.

[Evans, 2004] Evans, E. (2004). *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley.

[Fowler, 2013] Fowler, M. (2013). GivenWhenThen. *https://martinfowler.com/bliki/GivenWhenThen.html*. Accessed: 2019-03-10.

[Fowler, 2015] Fowler, M. (2015). Microservice Trade-Offs. *https://martinfowler.com/articles/microservice-trade-offs.html*. Accessed: 2019-03-12.

[Fowler and Lewis, 2014] Fowler, M. and Lewis, J. (2014). Microservices - a definition of this new architectural term. *https://martinfowler.com/articles/microservices.html*. Accessed: 2019-01-30.

[Fritzsch et al., 2019] Fritzsch, J., Bogner, J., Zimmermann, A., and Wagner, S. (2019). From Monolith to Microservices: A Classification of Refactoring Approaches. Technical report.

[Goldsmith, 2015] Goldsmith, K. (2015). Presentation: "Microservices @ Spotify". *http://gotocon.com/berlin-2015/presentation/Microservices%20@%20Spotify*. Accessed: 2019-02-15.

[Gouigoux and Tamzalit, 2017] Gouigoux, J.-P. and Tamzalit, D. (2017). From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 62–65. IEEE.

[Hasselbring, 2016] Hasselbring, W. (2016). Microservices for scalability: keynote talk abstract. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 133–134. ACM.

[Humble and Farley, 2010] Humble, J. and Farley, D. (2010). *Continuous delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley.

[Kalske et al., 2018] Kalske, M., Mäkitalo, N., and Mikkonen, T. (2018). Challenges When Moving from Monolith to Microservice Architecture. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 32–47. Springer, Cham.

[Mauro, 2015] Mauro, T. (2015). Adopting Microservices at Netflix: Lessons for Architectural Design. *https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/*. Accessed: 2019-04-29.

[Microsoft, 2018a] Microsoft (2018a). Challenges and solutions for distributed data management. *https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/distributed-data-management.* Accessed: 2019-04-30.

[Microsoft, 2018b] Microsoft (2018b). Communication in a microservice architecture. *https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/communication-in-microservice-architecture*. Accessed: 2019-02-15.

[Microsoft, 2018c] Microsoft (2018c). Design a microservice domain model. *https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/microservice-domain-model.* Accessed: 2019-05-13.

[Microsoft, 2018d] Microsoft (2018d). Designing microservices: Logging and monitoring. *https://docs.microsoft.com/en-us/azure/architecture/microservices/logging-monitoring.* Accessed: 2019-01-02.

[Microsoft, 2018] Microsoft (2018). Identify domain-model boundaries for each microservice. *https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/architect-microservice-container-applications/identify-microservice-domain-model-boundaries*. Accessed: 2019-02-15.

[Microsoft, 2018] Microsoft (2018). Using domain analysis to model microservices. *https://docs.microsoft.com/sv-se/azure/architecture/microservices/model/domain-analysis.* Accessed: 2019-02-15.

[Namiot and Sneps-Sneppe, 2014] Namiot, D. and Sneps-Sneppe, M. (2014). On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27.

[Newman, 2015] Newman, S. (2015). *Building microservices: designing fine-grained systems.* O'Reilly Media, Inc.

[Richardson, Chris, 2018] Richardson, Chris (2018). Database per service. *https://microservices.io/patterns/data/database-per-service.html.* Accessed: 2019-02-15.

[Scrum.org, 2018] Scrum.org (2018). What is a Sprint Retrospective? *https://www.scrum.org/resources/what-is-a-sprint-retrospective*. Accessed: 2019-03-28.

[Spring, 2018] Spring, D. (2018). A facilitators recipe for Event Storming. *https://medium.com/@springdo/a-facilitators-recipe-for-event-storming-941dcb38db0d*. Accessed: 2019-02-15.

[Vernon, 2013] Vernon, V. (2013). *Implementing domain-driven design*. Addison-Wesley.

A

The Migration Strategy

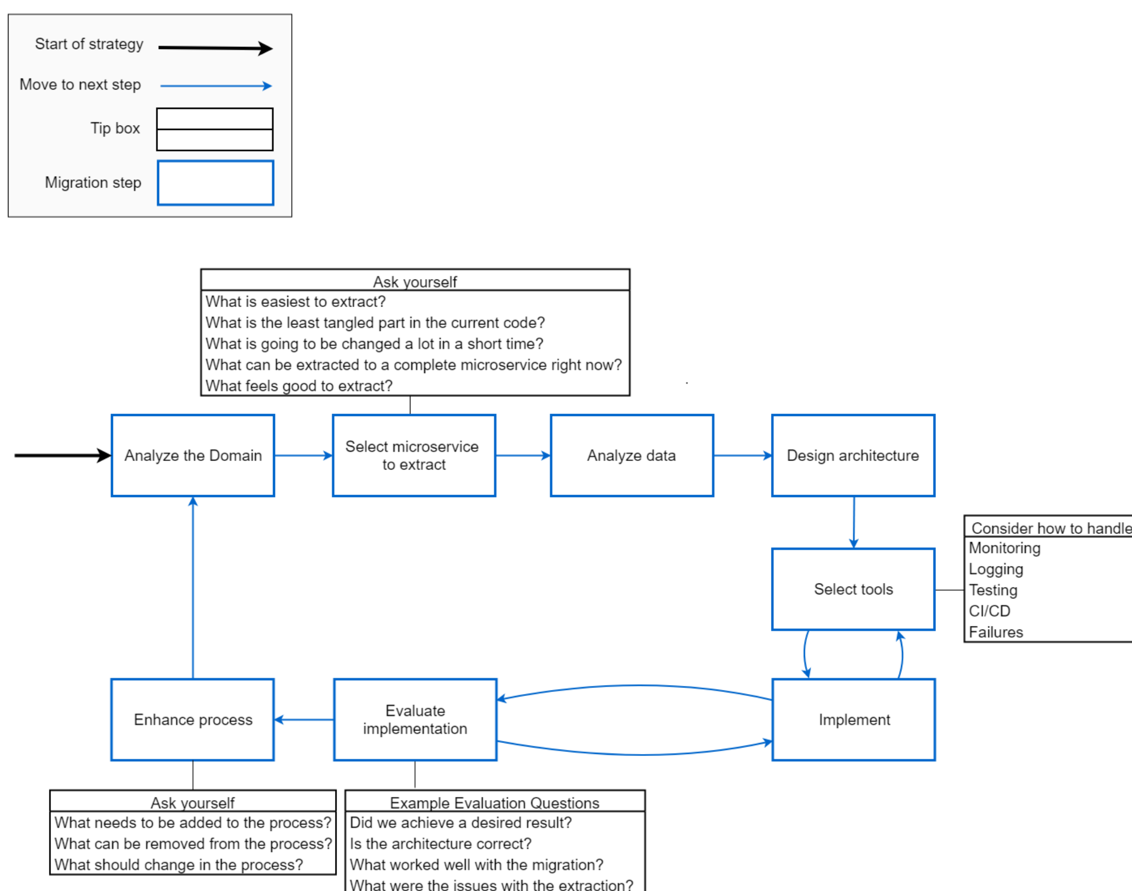

Figure A.1: The final migration strategy based on interviews, literature and our own experiences migrating.

Analyze the domain

Description
During this first step of the migration strategy the business domain should be analyzed. Understanding the domain, and preferably visualize it through a diagram, can be of great help during the process of finding a suitable decomposition of the system. The domain analysis should be done in collaboration with developers and domain experts and the discussions should be on a fairly high level and not go into implementation details. Understanding the concept and principles from domain driven design can provide guidance to the team in the process of understanding the domain and defining service boundaries.

The domain analysis can be seen as an up-front task, and will require most effort during the first iteration of the migration strategy. However, the domain model created should be re-visited at each iteration to make sure that it is up to date

Rationale
This step serves to combat Challenge CH1 from Section 3.3, which highlights the complexity and importance of understanding the domain and defining service boundaries. Even though this process can be complex, it will likely be time is well spent. Finding a suitable decomposition of the services will be beneficial for the migration as a whole. Domain driven design is commonly mentioned and could be used as a framework to help in the process.

It is important that the domain model created during the domain analysis is updated. The domain model will serve as a useful tool during later stages when selecting parts of the system to extract,and must be up to date in order for it to be of help

Outcome
The team who conducted this step should at the end have a great understanding of the domain, which includes knowing what parts that exists in the domain and what parts in the domain that are related to each other. In the end there should be a visualized and well defined domain model that will help the developers and architectures in the upcoming steps. The domain model will also work as a dictionary and form a unambiguous language, which ensures that everybody is talking about the same thing when discussing the different parts of the system.

Select microservice to extract

Description
As the domain has been analyzed it is time to select what part of the domain to extract in to a microservice. The domain analysis will be of help during selection process, but it is not always straight forward what to pick. These five questions can act as guidance during the selection.

1. What is the least tangled parts in the code?
2. What is easiest to extract?
3. What can be extracted to a complete microservice right now?
4. What is going to be changed a lot in a short time?
5. What feels good to extract?

Rationale
The questions are all based the solving Challenge CH3 from Section 3.3. The strategy in this challenge recommends starting with something easy and decoupled. Question one and two are based on this recommendation. These questions will mainly be used in the initial iterations of the extractions, since the first iterations will require developers to learn several new tools and work flows. As experience and knowledge regarding microservices is gained, these two questions should

be less important to consider.

The strategy also recommends focusing on important parts of the system to avoid spending time and money on building services that cannot or will not be used, which is why question three is included. Question four is important to consider since parts that are frequently changed are likely parts that are refactored often, because they are affected by several things in the system. Continuously fixing these part is often time consuming for the developers, and by extracting these parts they might not be refactored in future changes. This question is probably more important when the developers are more experienced in microservices and ready to tackle a more difficult part to extract.

Finally, question five is based on the last recommendation from the strategy in Challenge CH3. The question could be considered fairly vague, but we would argue that as developers gain knowledge about the domain and the system, the intuition of what makes a good service in the system could be an important factor to consider.

Outcome
After this step is completed, what to extract into a microservice should be clear and concise. If this is the first microservice, it should probably not be too complex, since the main learning objective of the first extractions will be how to work and think about microservices. A complex microservice might take much longer to implement without experience. As the developers gain knowledge and experience, more complex migrations can be probably be done faster and better.

Analyze data

Description
This step focuses on understanding the data that exists in the selected microservice, and what data updates that the service is interested of. This can be done by analyzing the use cases that the selected part will be responsible for and what data that is needed to perform these use cases. The use cases could also be seen as a tool for later evaluation of the microservice, to make sure that it can perform the use cases that were anticipated.

Rationale
As stated in challenge CH2, the challenge of managing decentralized data and communication is closely related to to challenge CH1. It is therefore important to understand what data that is needed in the service, and what data updates that needs to be propagated to this service. By analyzing the use cases for the service, all the data needed can be found.

Outcome
At the end of this step, it should be clear what data that belongs in the service and how this data is updated. It should also be clear what use cases the service will be responsible for. The outcome will be of value for developers, since developers will likely get a deeper understanding of the responsibilities of the service and how data is used and updated, which will be of help during the implementation phase.

Visualize architecture

Description
When a microservice has been selected, it is time to start think about the implementation of the microservice. However, before the implementation starts it can be of help to understand the basic architecture of the microservice, since it helps the developers understand how to implement the service and what is needed for it to function. The documentation from the domain and data flow analysis will be useful when analyzing what parts of the system the service needs to interact with and how data traverse in the system.

Rationale

Many of the challenges mentioned in Section 3.3 does not provide any specific strategies that to use, but rather that it is important to consider them and face them. For example, challenge CH2 discusses the importance and complexity of selecting communication protocol, and challenge CH5 discusses fault tolerance. There are several uncertainties related to exactly how these challenges should solved. Drawing and visualizing an architecture will force the architect and development team to consider the challenges and try to combat them in the best way possible.

Outcome
The outcome of the step should be a clear and usable architecture that the developers can rely on when implementing the new microservice. This step should also provide an opportunity for developers and architects to consider and understand how the service interacts with other services, and what different options exists to combat the challenges of working with microservices.

Select tools

Description
There are several tools that can be of help when combating the technical challenges when developing microservices. Therefore, it could be helpful to take a step back to explore and consider what tools that may fit the application that is being developed. For example, how will the microservice handle failures? How will it be monitored? Will it be implemented into continuous integration and delivery pipelines, and how? How will events be logged? How will the features be tested, and should the test be written before, after or during implementation? These questions are probably mostly useful in the beginning of the migration to microservices, and after a couple of iterations of the process the tools have likely been selected and implemented.

Rationale
Challenge CH2 states that it is important to consider communication, challenge CH4 that it is important to implement automations and challenge CH5 that it is important to handle failures and to implement monitoring and logging. How these are implemented and in what order is not clear, and it the order might not matter, but it should be done. This step is an opportunity to explore tools and get familiar with them.

Outcome
Most of the uncertainties around the implementation should be clear after this step, or there should at least be some ideas on how to solve the uncertainties during implementation. There will likely still be issues with implementation regardless of how much preparatory work is done. However, it can be cost effective to think about the uncertainties before implementation starts, in order to not implement something that will not be used, or forget to implement something vital.

Implement

Description
Once everything is considered and all the preparatory work is completed, it is time to implement. If this is the first microservice to be implemented, it will likely take more time due to environment configurations such as setting up a cluster, setting up pipelines and setting up event buses etcetera. However, this step is very similar to developing any new feature to a system, and while it might take some time, it should not be a big problem if the preparatory work is properly done.

Rationale
At this stage all preparatory work should be completed and the developing team should be ready to implement everything.

Outcome
The outcome of this phase should be a functional microservice that can be validated against any requirements and the architecture.

Evaluate implementation

Description
In this step the microservice is evaluated and compared to what was envisioned at the start, and why any pivots from the vision has occurred. There could also be some general discussions on if the achieved result is what was expected from the start.

It is also important to reflect on difficulties during the implementation and how to avoid them in the future, but also acknowledge what worked well and ensure that the things that worked well are present in upcoming iterations.

Rationale
This step was highly influenced by the sprint retrospective from the agile framework Scrum, where the goal is for the team to identify improvements to implement in upcoming iterations [Scrum.org, 2018]. It is important to improve and learn from eventual mistakes or new insights that occurred during implementation, and having the discussion as a part of the process forces the team to reflect best practices.

Outcome
Everyone in the team should agree on that the microservice is well implemented, and hopefully new knowledge has been gained that will improve future extractions.

Enhance process

Description
In this final step the process itself is analyzed and adapted to better suit the team and how they prefer to work. There might be a step missing that needs to be added, or there might be a step in the process that is redundant. Discuss the work flow in a meeting and adapt it to better fit the team.

Rationale
This strategy has been created by us and has not been verified by external sources. Whether or not it is applicable to all companies is questionable, and therefore the process itself should be reflected upon, and adapted to better fit the working process of those who use it. By reflecting on the process after each iteration and improving it, it will be more efficient and yield a better result as more iterations are completed.

Outcome
The outcome of this step is either an updated migration strategy, or a consensus that the strategy works well and should be used as it is.

B

Summary of Challenges and Strategies

Challenges List
Challenge CH1: Understanding the domain and defining service boundaries Well defined service boundaries is important in order to take advantage of the benefits with microservices. Defining service boundaries is complex and requires great understanding of the domain. ***Strategies*** Domain knowledge is obtained by analyzing the domain in collaboration with domain experts and developers. Concepts from domain driven design can be of help during the process.
Challenge CH2: Managing communication and decentralized data Handling decentralized data is a challenge. When data is updated, it needs to be propagated to interested services. The propagation is done using some communication protocol, and poor decisions of communication will affect the performance of the application. ***Strategies*** This challenge is affected by the outcome from handling Challenge CH1. By understanding the domain, you should also gain knowledge in how data traverse in the system, which should help when considering how and when to propagate updates. Where to use synchronous and asynchronous communication and how to handle data consistency should also be considered.
Challenge CH3: Selecting microservice to extract Extracting microservices from the previous system will likely be complicated. The ease of extraction will be dependent on what microservice is chosen to be extracted. ***Strategies*** During the initial part of the extraction, easy and decoupled parts of the system should be considered. It is also important to consider services that can be used right away, and what will change in short period of time. As the experience increases, the development can make more decisions on intuition.

Challenge CH4: Managing automations Setting up the infrastructure for automation is complex and requires experience. Automations are important to the development process and is recommended. ***Strategies*** Structuring the teams as DevOps teams can help with automations. Also, ensure that the microservices have independent lifecycles in the automations.
Challenge CH5: Handling and tracking failures Handling, detecting and reacting to failures is important when using microservices, since there are many independent parts to consider. Monitoring and logging can be of great help, but setting up the infrastructure for this is complex. ***Strategies*** Try implement mechanisms such as "Circuit breaker pattern" to handle failures in other services. Visualizing the logging and monitoring through a dashboard that the developers can use is recommended. Consider what properties that are interesting to monitor and log, and add the logs to a centralized storage to enable querying.

Table B.1: Summary of the challenges and strategies found