Master’s thesis

Master’s Programme in Computer Science

Modernizing usability and development with microservices

Janne Kauhanen

November 06, 2022

FACULTY OF SCIENCE

UNIVERSITY OF HELSINKI

Tiedekunta - Fakultet - Faculty	Laitos - Institution – Department
Faculty of Science	Department of Computer Science

Tekijä - Författare - Author
Janne Kauhanen

Työn nimi - Arbetets titel - Title
Modernizing usability and development with microservices

Oppiaine - Läroämne - Subject
Computer Science

Työn laji/Ohjaaja - Arbetets art/Handledare – Level/Instructor	Aika - Datum - Month and year	Sivumäärä - Sidoantal - Number of pages
M.Sc. Thesis	06.11.2022	63 pp. + 3 appendices

Tiivistelmä - Referat - Abstract

Legacy software systems, which refers to old and likely outdated software applications and practices, are a reality that most software development companies have to contend with. Old practices and technologies are often at fault for slowing down development and deployment of software, as they can have compatibility, security, scalability and economic issues with their continued use, among other issues. Software modernization, reengineering and refactoring can alleviate the issues stemming from legacy systems, whether it be in the form of altering practices, updating technologies or changing platforms.

There are many technologies and methods that can facilitate the modernization of a software system, including a move to using different architectures, specific newer technologies and changing the methods of working and developing the software system. These technologies and methods, and modernization in general, come with their own risks and challenges that must be considered for a successful modernization to take place; These strategic considerations are a key factor in modernization.

This thesis will explore software modernization in general through literature reviews and as a case study for a specific company using data from surveys and the case company's logs, with a look into the technologies, concepts and strategies required for a successful modernization, and what kinds of effects modernization can have on the software system being modernized, both from a user perspective as well as from a developer perspective. The end-result of this exploration reveals that modernization is a complex subject with many challenges, but that also offers benefits to the software system being modernized. These results are best used as a guideline on what issues should be concentrated on during modernization, with a mindful consideration for the limited scope of the case study represented within.

Avainsanat - Nyckelord - Keywords
software modernization, microservices, cloud computing

Säilytyspaikka - Förvaringsställe - Where deposited

Muita tietoja - Övriga uppgifter - Additional information

Table of contents

1. Introduction

Software modernization [Gar06,Mal10], also known as legacy modernization, platform modernization and application modernization, refers to a conversion, refactoring, rewriting or porting of application systems to using newer programming languages, architectures, hardware, libraries or frameworks, for example. This is done to extend the life and value of old software, since often newer platforms and practices allow for easier, faster and more reliable development and maintenance of the system [Gar06]. Presently, software modernization can also involve data migrations to cloud-based platforms as well as the adoption of scalable and modular microservices [BHJ16,Alt19]. In general, software modernization's most common use is to introduce new technologies to the system being modernized.

Microservice architecture allows for creating multiple loosely coupled autonomous services, which communicate with each other to act as a larger software as a whole, while still maintaining a single responsibility principle [Kal17]. This allows for easier maintainability [OS19], development and testing of the software, as each microservice can be addressed by itself and new microservices can be added as needed.

More in-depth benefits of a microservice architecture include developer independence, as each part of the whole can be developed as a completely independent entity [OS19], allowing developers to work better in parallel. Scalability, especially server-side scalability, is improved with microservices [Kal17,OS19], and the smaller scope of each individual service instance can reduce complexity issues. Better quality and security of service is also almost guaranteed, due to the fact that if one microservice fails, it does not cause the whole system to fail, as can be the case in monolithic architectures. Microservices also allow for much higher scalability of software, as they can be scaled to meet the requirements for that component's functionality.

The basic idea behind software modernization via microservices is to improve an already existing service to be much more modular, maintainable and usable. Older legacy software systems are often based on the monolithic architecture [Alt19], as these older software systems have their roots in times before microservices was introduced as a concept, in which everything was developed as a single artifact; Even now, many software systems start out as monolithic systems [Kal17] due to their simplicity in new development. While this is easy and works well for smaller codebases, a monolithic architecture can become very difficult and slow to develop and maintain once the codebase becomes large [Kal17]. Many companies'

codebases can be decades old [Alt19], and for a monolithic architecture with that much legacy code that is still in use, this can become a bottleneck in production, which microservices can alleviate.

Microservices do have their own issues [Fo15,Kal17], as a distributed microservice infrastructure is more difficult and complex to create than a monolithic system, and maintaining the connections, communications and transactions between different microservices within a microservice architecture is more challenging, which in turn creates additional challenges for testing, deploying and production use. These issues can pose great challenges for companies with little experience in their use, as training an organization's personnel to have the new skills required will take a great amount of time and effort [Fo15] that could otherwise be used for more development and testing. In terms of modernization from a legacy product, this also creates management and planning challenges, such as how to properly allocate resources to the aforementioned training [Ber99,Fo15], choosing the solutions and products to use for microservices, making a concrete schedule and communicating the necessary changes within the organization. In the case of a software that needs to be kept in production use throughout the whole switch to microservices, such as the case company of this thesis, this also creates additional issues as services cannot simply be taken down to be made into a microservice, but must be maintained and developed in parallel with the old methods.

This thesis explores the software modernization that happened in a small-sized Finnish software company between 2018 and 2020 as a case study.

1.1. Case company introduction

The case company, referred to as Case A of this thesis, provides software-as-a-service (SaaS) for several Finnish and international companies using enterprise resource planning (ERP), financial planning and contract management solutions; These contracts include rental contracts as well as other financial contracts, such as lease and billing.

The users of the software product come from many different background and computer-skill levels, ranging from no experience to extensive experience. The most common use-case is using the software to log and confirm different work-orders and service requests in the software.

Case A underwent an evolutionary software modernization process that started from the late 2018 and ended halfway through 2020, with the aim of taking advantage of new techniques and technologies as well as letting go of old practices. While the majority of the changes were finished halfway through 2020, the

iterative development and adoption of new practices is now an ongoing effort within Case A, so as to keep Case A's practices modern without having to specifically overhaul most of their existing practices.

Throughout the software modernization process, the software has been in constant production use, and there have been parallel development tasks and software maintenance tasks that have been undergoing as well.

While Case A had undergone many updates and modernizations to its techniques and technologies previously, these efforts had been on a smaller scale, with more of an iterative development approach than the one planned in 2018.

As its main programming languages and technologies, Case A uses C# as the back end language, with Javascript/JQuery for most of its front end work. At the start of the modernization, there were still several services that were using Delphi for back end functionalities, but those were being slowly phased out of use. For the database, Case A's product uses Microsoft SQL Server, with local and remote servers.

1.2. Research questions

This thesis addresses the general research problem of software modernization [see e.g., Gar06,Mal10, Alt19], such as changing the platform of a software from local servers to a cloud service or changing a monolithic software architecture into a microservices architecture, and how it can affect the software product and what challenges it poses, with the following specific research questions:

RQ1. How may software modernization affect a software product?

RQ2. How may software modernization affect the usability of a software product for the end user?

RQ3. How may software modernization affect the development of a software product from a developer's point of view?

RQ4. What challenges can modernizing a legacy system that is still in production use pose?

RQ5. How can the challenges of modernizing a legacy system that is still in production use be mitigated?

1.3. Thesis structure

This section gives an outline of the structure and flow of the thesis, and explains what the different chapters of the thesis contain.

Chapter 2 covers the background and motivation for the thesis itself. In that chapter we go through what is meant by a legacy system, what monolithic architectures mean and what is meant by modern architecture in this thesis. We also cover the various tools and frameworks used for the modernization effort in Case A at length, along with the restructuring of Case A's development teams that was a part of the modernization.

Chapter 3 covers the research methods used in the thesis, going through the research environment within and without Case A. The survey and the in-depth questionnaire used in our case study and the methodology used for employing them are covered, as well as the literature reviews used for the parts of the thesis that are not company specific. In the research environment part we discuss the situation in Case A before and after the modernization changes, and the survey and in-depth questions are covered after that. We also cover the study plan for the thesis in this chapter. The literature reviews go through what modernization strategies, methods and techniques are available in the field overall.

Chapter 4 covers many of the risks and challenges of modernization, as well as some of the solutions proposed for them by existing literature.

Chapter 5 covers the answers gained from the survey and in-depth questionnaire, as well as analysis on that data. The data is analyzed by comparing the answers gathered from the survey and in-depth questionnaire to each other, and then comparing these results to those generally gained from modernization efforts in existing literature. These results are used to measure the efficacy of the modernization efforts between the situation in Case A before the modernization and the situation after the modernization. We also cover the results of Jira ticket analysis to see how the modernization impacted user ticket activity.

Chapter 6 covers discussion on the general topic of modernization, and what further research could be done on the subject. This chapter covers company-specific discussion on the modernization effort.

Chapter 7 covers the overall conclusions we can draw from the research conducted in this thesis, as well as answers to our research questions.

2. Background and motivation

This chapter details the motivation and background for modernizing a monolithic legacy system into a more modern, microservice-centric system. We go through what legacy systems are and what they mean to companies, as well as what costs they can generate for them. After that, we take a look at what a modern architecture is, what microservices are, different approaches to modernization and the variety of different tools and frameworks that can be used for modernization, and how some of them were or are used within Case A.

2.1. Motivation

The main motivation for this thesis came from the fact that while Case A had had several smaller modernization efforts in the past, these efforts had been done with a business-focus approach, based upon the software's production performance and customer feedback, within Case A: Dotnet was taken into considerable use within the software during the first quarter of 2017, and was used as a standard within Case A afterwards. This eventually evolved into taking .Net Core as an object-relational mapper (O/RM) and EFCore into full use, and EFCore mappings were implemented in the code and database in early 2018.

After that had been done, the planning of microservices was taken into more focus, and these plans were well on their way in mid-2018, and by August 2018 tasks for microservices that would handle business logic were already being made. This was when the larger modernization project was taken into planning.

Adding parameters to EFCore was still continuing in early 2019, and the Business Logic, which is a C# library to share business logic code across applications, was being improved just before the start of the big modernization that was the motivator of this thesis: This modernization started in earnest in mid-2019.

When this modernization project was initially taken into consideration in 2018, there weren't many internal guidelines or documentation on best practices on how to practically do this modernization, so it took time for the modernization to properly start. This was especially true as this was also Case A's big foray into using microservices, which Case A in general had no previous experiences in.

This larger modernization effort brought several changes into Case A, the largest of which was the aforementioned usage of microservices, but also included the usage of Docker Containers, Microsoft Azure, Syncfusion and Google's Material Design. Of the ones mentioned, Docker and Azure required a great

deal of planning and education for the developers and other employees within Case A. A great many of new, upcoming features were also planned to take advantage of these new technologies, and as such represented a unique challenge to Case A. To combat this, new employees with specific expertise in the required technologies were hired to facilitate their adoption.

On top of this modernization effort on technology and techniques, a large change to Case A's working methods was also started alongside it: Restructuring of the teams according to the Spotify Model [KIS12] to Tribes, Squads, Chapters and Guilds. As this coincided with the other changes, it compounded the challenges of modernization within Case A, despite Case A having had similar team restructuring efforts before.

2.2. Legacy systems

A legacy system refers to an old computer system, method, hardware or application that is still in active use within an organization, and can be characterized by having been implemented with old technological methods or outdated programming approaches. Legacy software systems are often monolithic in nature [Kal17], where modifying one part of the system necessitates updating the system as a whole, and as such are poor at scalability and maintainability. Layered monolithic architectures are widely used in networking and operating systems [Kal17], and are split into a UI layer, a service logic layer and a data access layer which accesses a single database and handles the application's data; This configuration is shown in Figure 1. In general, legacy systems are very resistant to the evolution of software [Mal10], but are often critical to day-to-day operations of the organization.

Monolithic architectures refer to application architectures with a single codebase, which can also operate on a single database system. This is considered to be a standard way to start development for most developers, and due to everything being contained in one location, is very efficient as long as the codebase stays small and manageable [Kal17]. However, there are numerous problems with monolithic architectures that become more and more apparent as the codebase grows and becomes more tightly coupled, some problems of which we cover in the next subchapter.

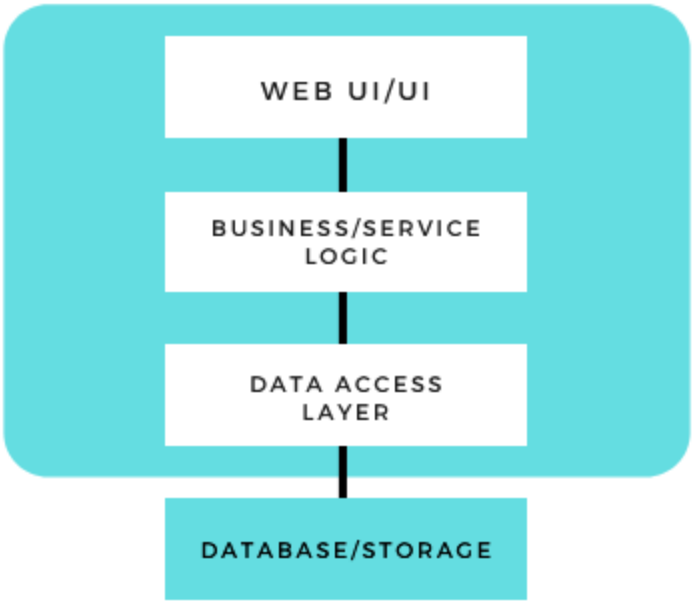

Figure 1: Standard monolithic application architecture with n-tiers. All parts of the application reside within the single monolith. Figure adapted from [Kal17].

Legacy systems are most common in older companies [Alt19], where updating or replacing them has been deemed to be less cost-effective than maintaining the existing legacy approaches. Other reasons for not replacing a legacy system with more up-to-date systems include not having the time or the resources to train or hire personnel to use a new system, difficulty in integrating data from the legacy system to a new system, the possibility of having multiple legacy systems within one company and prioritizing them accordingly, the difficulty of dealing with a great amount of additional codelines and the requirement of having the skills to do so among other issues [Ber99,Alt19]. Many challenges of software modernization have persisted for decades, such as issues relating to training personnel and management, while some challenges are relatively recent, such as those pertaining to cloud-computing and microservices.

For many companies still using legacy systems, a move away from them takes a great amount of resources and time that could be used in developing new functionalities in the base product, and can be seen as a waste. There is also a fear of failure when it comes to updating legacy systems [Ber99]. We will take a closer look at the risks of updating a legacy system in chapter 3.1.3.

2.2.1. Costs of legacy systems

Legacy systems have a number of costs to companies still relying upon them [Alt19], with a great many of these costs caused by the fact that most legacy systems are implemented with a monolithic architecture

over a number of years, becoming extremely large and complex. This can cause the software to fossilize, having accumulated enough technical debt to cause the software to become extremely difficult to maintain and update [FB19].

Monolithic architectures are common and useful in the beginning of the lifecycle of a software product [Kal17], but over time they can become huge burdens for the organization. Because monolithic architectures are developed on a single codebase, for most language-bases the developers cannot simply deploy a single change to the codebase as its own unique item: They must deploy every part of the application, which harms continuous deployment, as any background tasks must be interrupted and parts of the application that are not updated may fail [Ric15]. This constant re-deployment also makes iterative development slow, as it lengthens the release cycle of the product [Ric15,Kal17]. All of this makes deployment of a large monolithic application into a cumbersome process, which is prone to issues unless strict rules are followed every deployment.

This development can be exacerbated by the fact that developers within a team change over time, and this can result in the necessary knowledge on keeping a legacy software product maintained disappearing with old employees. This is especially challenging for an organization if there are only very few developers who know how the legacy systems work; There exists some correlation between technical debt and developers leaving a company [SS21].

The previous issue can be alleviated with proper documentation [Alt19], but this only highlights another problem with many legacy systems: The lack of documentation. While not an issue with all of the organizations using legacy software, the lack of documentation can greatly decrease productivity, as a lot of resources has to be expended on training and retraining developers to use the legacy software that exists within the organization. Even with proper documentation in place, legacy software systems may use outdated languages and methods which many modern developers are no longer familiar with, which necessitates finding specialist developers for the post or investing in time-consuming training sessions [Ber99,Alt19].

With legacy systems, yet another cost comes from integrations with other systems: Due to legacy systems using outdated technologies, methods and techniques, it becomes very challenging to integrate them with newer systems. This may necessitate the use of third-party tools, or a great amount of custom code [Alt19], to act as interfaces between the two systems, which automatically causes another layer of risk because it adds another layer to the software that must be kept track of and maintained. This difficulty in integrating with new ease-of-use functionalities such as geolocation, user authentication and data-sharing

can cause an organization to miss out on business opportunities because the cost of creating the needed connection with the legacy system is too high [Alt19]. There can also be issues with compliance to industry standards [Cap19], especially in heavily regulated sectors. For example, GDPR (General Data Protection Regulation) caused a great deal of issues for many companies, especially those still using outdated technologies: In a report by the CapGemini Research Institute [Cap19], 38% of the responding IT executives reported that the most complex barrier to aligning their organization to GDPR, were their existing legacy systems. 42% of these executives considered legacy IT as their most critical challenge.

Another issue with legacy systems is the difficulty to maintain comprehensive tests for the different parts of the software due to the codebase slowly becoming less comprehensible for the developers over time [Kal17], which is often a symptom of long-term legacy systems. This is of course not always the case, as it can be mitigated with disciplined documentation and the creation of tests by the developers and software architects, but is still one of the risks of using legacy systems.

Security also becomes an issue, as legacy systems may be less resistant to harmful programs and cyber attacks. This can be due to the sheer age of the components of the legacy system, which translates to more knowledge about its exploits, or even because the vendors no longer support the components [Alt19].

From a less technical point of view, using legacy systems can translate to lost business opportunities [Alt19], as prospective clients may think twice if a company they are negotiating with is using archaic methods and technologies. One of the biggest reasons to modernize for some is, indeed, the ability to satisfy customer expectations and thus not lose business revenue due to old technologies.

2.3. Modern software architecture

Modern software architecture in the context of this thesis consists of modular and scalable techniques and technologies that improve the development cycle and maintainability of a software product or a system. Many of these techniques and technologies have become widely used by many software companies in recent years, as the benefits of using them far outweigh the necessary development and training time it takes for a company to adopt them.

The technologies we will mainly be exploring in this thesis are Microsoft Azure, Docker containers, Material Design by Google, OpenAPI, Dotnet Core, Entity Framework Core and Asp.net Core.

We also discuss different strategies on modernization, and what choices Case A took, as well as how the development team structure changed within Case A to facilitate better communications and compartmentalized working practices.

2.3.1. Microservices

Microservice architectures are cloud-native architectures that focus on breaking down a software system into small, modular applications that can work both independently and interconnected with each other [LF14, BHJ16]; An example of what a microservices architecture can look like is presented in Figure 2. Each distinct microservice can be on a completely different platform, working on a different programming language, and are connected to each other and the UI via APIs so they can work together in unison.

Microservices facilitate easier Continuous Integration (CI) and Continuous Delivery (CD), and they are helpful in a more iterative development cycle, as well as iterative modernization for the whole software base [BHJ16].

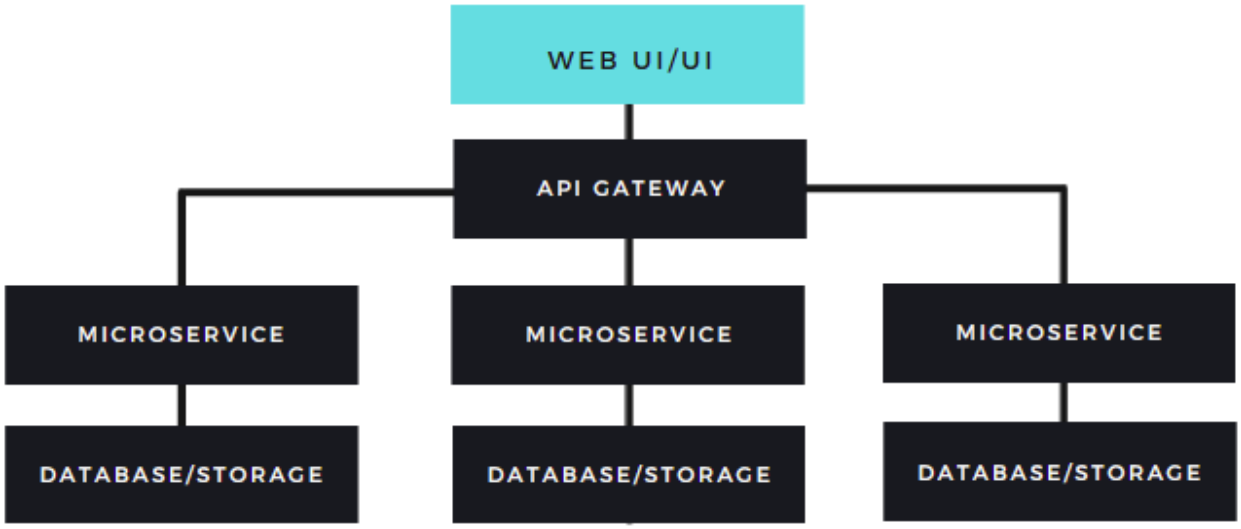

Figure 2: An example of a microservice architecture, using an API gateway and three separate microservices with their own databases/storages. Figure adapted from [MW21].

The reasons for adopting a microservices architectural approach over a monolithic one are several: As each microservice is an independent modular component of the software, they can be developed independently of each other, helping to decouple developers and development teams from each other. This way, an issue in the code or logic that would prevent further development for one team does not necessarily

hamper the progress of another team in any way, and accidental breaches between different services' fields become rarer [Kal17]. This also helps with testing, as each microservice can be tested independently, speeding up the testing process. Both of the aforementioned reasons to use microservices also contribute to making development cycles faster: Because the development of microservices can be asynchronous, it contributes to the flexibility of development of the system as a whole [OS19,Aka20].

As microservices also scale much easier than monolithic systems [OS19], they are very attractive to organizations undergoing growth or other organizational changes.

For the end-user, the use of microservices architecture can appear as improved stability [Aka20], as the unavailability of one service does not necessarily mean that the system as a whole is completely unusable, as not all end-users require the use of all microservices in a single system. This also translates to improved performance, as only the parts necessary for the end-user are loaded and used when using the system.

End-users also benefit from the flexibility from a business standpoint: Because each microservice is independent, they can pick and choose what services they wish to have from the system to themselves, improving the flexibility of the product's pricing model [BHJ15].

2.3.2. Tools and Frameworks

Here we take a look at some of the technologies that can be used to enable modernization for companies and developers that are using less efficient legacy methods. Many current software solutions and applications already use some of these technologies in some capability, as they are easily available and have extensive documentation and software support.

Microsoft Azure

Microsoft Azure is a flexible cloud computing service provided by Microsoft that offers the ability to build, test, deploy and manage applications and microservices in a cloud. It has a number of services that can be subscribed to individually, making the platform very customizable, as it can offer SaaS, PaaS and IaaS according to its customers' needs. It was originally launched in 2008 under the name of Windows Azure, but was renamed in 2014 to Microsoft Azure, and has since expanded greatly in the services it offers [SrA08,Azu10].

As mentioned, Microsoft Azure offers a great breadth of services within it, including products for AI and machine learning, analytics, blockchain, containerization, devops, integration, authentication and others

[Azd21]; A comprehensive list of their services will not be listed here, as we are not focusing solely on Azure in this thesis. In short, it can generally be said that if a company or developer requires a comprehensive and customizable cloud computing package, they can find it on Microsoft Azure.

For Case A, the main use cases for Azure are monitoring and error investigations as well as the ease with which APIs can be created and configured using it.

As Microsoft Azure is a cloud computing service, a short explanation of cloud computing services is in order: Cloud computing services operate on external data centers provided by external service providers, rather than being provided by their users in-house [CaV20]. A cloud provider has the hardware, servers, virtualization and other infrastructure as a shared resource in the cloud, as illustrated by Figure 3, and users of the cloud computing service can choose what resources they want to use from the cloud. This distribution of cloud services is generally automated by the cloud provider, offering a very dynamic service for the users of the cloud computing service, as the users can choose to subscribe to the services they need on-demand. This allows cloud providers to have very flexible billing models, something that was more difficult to do with in-house data centers, or even with external non-cloud data centers. In short, cloud computing is an utility, like electricity, but for applications, storage or virtual machines. Docker utilizes this to great effect due to how it is structured [CaV20].

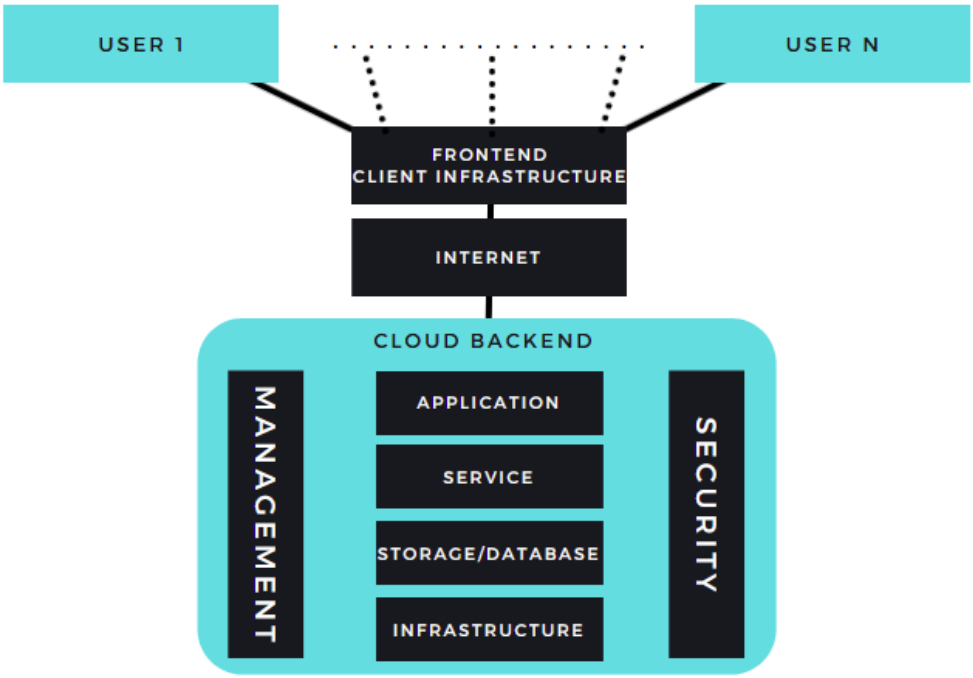

Figure 3: A general example of a cloud architecture. Multiple users can access and use the cloud backend, regardless of their own hardware/software, as long as they can access the frontend. Figure adapted from [Jen21].

There are some drawbacks to cloud services, such as security and privacy concerns, data mobility issues and responsiveness in the face of technical problems among other issues [AA13,PV19]. But in many cases, the pros of utilizing cloud services for large web-based applications outweigh the cons, as their cost-effectiveness over conventional data centers is quite high, especially when the up-front costs of setting up an in-house data center are taken into account. In the case of microservices, cloud services become even more effective, as is illustrated more in the next technology: Docker.

The most interesting aspect of Azure for Case A was its ability to deliver granular data about transactions and other logs within Case A's software package, and between its microservices. Azure's microservice and API deployment handling could also be important for Case A, as it offers a simple and unified platform to do this from. Using Azure still offers a simple API configuration and subscription utility for Case A, even though API deployment in Case A is being handled by Jenkins [Jen11], an open-source automation server.

Docker

Docker is a set of products that allow its users to have multiple small independent containers for each application in isolation [PoN19,PoN20,Doc17], allowing easier development in different environments.

It works as a combination of multiple platform-as-a-service (PaaS) products that use OS-level virtualization to provide similar services as virtual machines, but unlike virtual machines, they do not need to simulate a complete OS and machine for each separate application within a single server. Instead, Docker uses software packages called containers, which contain the software system, libraries and other necessary files to run a single application, working on top of one shared OS and Kernel. Containers are isolated slices of the OS located on the machine, rather than completely separate virtual machines that simulate a whole computer system.

Docker containers are an organized collection of namespaces [PoN18], which wrap system resources into abstractions that make them appear to the processes within the same namespace as isolated instances instead of a global resource that is used by other processes; An example of this is shown in Figure 4. There are eight types of namespaces [KeM21]:

1. Cgroup: Control group root directory.
2. IPC (Inter-process communication): Message queues, System IPC, POSIX.
3. Network: Network devices, stacks, ports.
4. Mount/Filesystem: Mount points.
5. PID (Process Identification): Process IDs.
6. Time: Boot and monotonic clocks.
7. User: User and group IDs.
8. UTS (UNIX time-sharing): Hostname and NIS (Network Information Service) domain name.

Fortunately, Windows calls its own OS isolation namespaces as well, so we can avoid a lot of cross-platform confusion, and Docker works the same way regardless of whether or not the underlying OS is Linux or Windows.

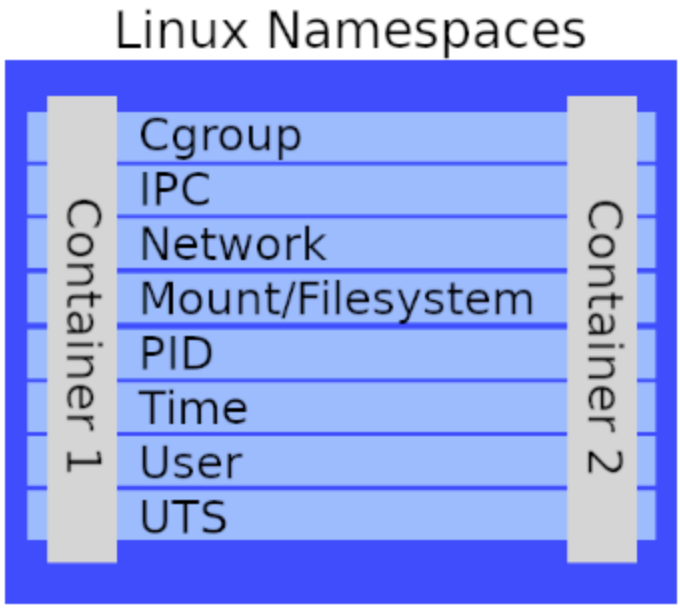

Figure 4: Illustration on how a Docker container resides "on top of" the linux namespaces: Container 1 and container 2 are each their own separate groups of namespaces, isolated from each other [PoN18].

What this all means in practice is that each container uses a lot less resources on a server than a virtual machine, and as such more of them can be run on a single server without appreciable loss of performance. For software companies managing software-as-a-service (SaaS) products, this can create great opportunities for boosting performance and saving resources, especially servers, which can easily become a bottleneck in a growing software system. This also helps in making software development platform-independent, as the minimum that is required from a specific computer is the ability to create a Dockerfile, which is a set of build instructions for Docker to build an app and its dependencies into a container image [PoN20]. So a developer might develop this at the office or in the cloud with minimal setup required for the environment.

Due to how Docker works with containers, it is extremely useful for developing microservices, as each application can be more efficiently sliced to run in their own specific environments within a server.

Taking Docker, and microservices, into use was one of the main changes that came with the modernization effort, and was one the technologies with the least previous experience within Case A.

Material Design and Syncfusion

Material Design is a design language developed by Google with the intent of creating designs that mimic the style of paper and ink [Mat14]: A viewer of a page created with the principles of Material Design can

decipher at a glance what can and cannot be interacted with, using seams and shadows of the buttons and other components to highlight the interactable parts, as if they were different physical materials and parts of the page rather than flat buttons and links.

It is used to provide a framework for building easily identifiable and interactable frontend user interfaces for many applications, including many of Google's own applications.

Along with Syncfusion, Material Design was used to provide a new way to design the frontend user interface for Case A, as previous methods for user interface design were more ad hoc within Case A.

Syncfusion is a developer platform that offers hundreds of different user interface components for .NET and Javascript, as well as comprehensive support for each of its components [Syn01].

It is primarily aimed towards enterprise usage, and offers several different packages depending on how many developers are going to use it. As a developer platform it supports ASP.NET MVC, ASP.NET Core, ASP.NET Web Forms, Angular, Javascript and jQuery among many others, and makes further forays into user interface development that much easier, especially for organizations who have not had an organized UI platform beforehand.

For Case A, Material Design and Syncfusion were chosen to better organize and streamline the design of the product's interface, and to provide a guideline on future development of the UI.

OpenAPI and Swagger

Swagger is a tooling ecosystem for the development and description of APIs that use the OpenAPI Specification, in particular it is used in RESTful web services [SmB16,OAS17]. Swagger tooling can also be used to describe APIs outside of the OpenAPI standard, but when used together with the standard it can create OpenAPI documentation directly from the code itself, streamlining any documentation tasks. It is also possible to create SDKs directly from the OpenAPI documentation itself using Swagger Codegen [SmB16].

Swagger was initially created in November 2015 along with the advent of the OpenAPI Initiative by a company called SmartBear Software, which had created the Swagger API before this, with several other large companies acting as founding members, such as Google and Microsoft. This initiative eventually led to the rebranding of Swagger to OpenAPI specification in January 2016 [SwG15].

Swagger API 2.0 was renamed to OpenAPI 2.0 in 2016, which can cause some confusion, but regardless of whether or not someone is talking about Swagger or OpenAPI, they are often talking about the same thing: OpenAPI [MHR20]. In this thesis, we will be using the terms OpenAPI and Swagger interchangeably.

In Case A Swagger was taken into use in late 2017, and remains in use to this day as the tooling ecosystem for Case A's APIs. The main reason why Swagger was taken into use was to provide a development interface for the customers, as the integration between Case A's product and the customer's software was being developed simultaneously on both sides. Another big reason to have taken Swagger into use, was the ability to create documentation straight out of the code itself, rather than having to document everything by hand. Previously, Case A used to have paper documentations on its software integrations, which mostly pertained to customer use-cases rather than internal technical comments: Those were handled in the code itself and in comments of the code. Swagger allows documentation to be created in a much more simple fashion, and thus increases developer efficiency.

Dotnet Core

Dotnet Core, which was renamed to .NET in late 2020, is an open-source general-purpose development platform provided by Microsoft that has cross-platform functionality and is lightweight, fast and modular [DNC16, DNC20]. It is used to create software solutions that require to be run across multiple devices, from backend servers to mobile devices, as it works agnostically over all of these devices. It uses NuGet packages to achieve architectural modularity, which can be added or removed to projects as required. Dotnet Core is a lighter version of .NET with an emphasis on performance and modularity.

Dotnet Core fully supports C#/C++/CLI and F# languages, and can be used with Visual Studio 2017/2019 or later. Other IDEs are also supported, such as Visual Studio Code and VIM [DNC20].

In Case A, Dotnet Core was taken into full use in 2017, and has been in use ever since, with an update from .NET Core 2.x to .NET Core 3.x taking effect in late 2020. It was added to the software stack because JSONAPI needed to be very fast. It also served as a useful springboard with other tech-modernizations.

Entity Framework Core

Entity Framework Core (EFCore) is Microsoft's cross-platform data-access framework for .NET that can function as an object-relational mapper (O/RM), which enables developers to work with databases by

using .NET objects, rather than having to create a great amount of custom data-access code to do the same [LeJ20,EFC20].

Case A had used PETAPOCO for nearly all of its object-relational mappings since early 2015's, before it slowly started moving towards using EFCore, which started happening with the adoption of JsonApi-DotNetCore in 2017. This shift picked up more speed in 2018, when Case A started to use specialized business logic organization within the software, and took EFCore as the main focus for its O/RM needs. For the modernization that happened in 2019-2020, EFCore was used as one of the key components in conjunction with microservices, and is now in more wide-spread use within the codebase.

For Case A's modernization, the micro O/RM Dapper [Dap20] was also used as a way to streamline some issues and legacy tables during the process.

Asp.net Core

Asp.net Core is an open-source cross-platform web framework provided by Microsoft, and it runs on .NET, .NET Framework and on Windows [ANC16]. Its main uses are building web apps and services, internet of things (IoT) applications and mobile backends, and is platform-agnostic for ease of use. It can be used to deploy directly into the cloud or local servers. It is a direct successor, and combination of, ASP.NET MVC and ASP.NET Web API.

Case A adopted Asp.net Core 3.1 for the modernization effort's backend framework in 2019, and the current plans are to upgrade to 6.0 in the near future.

2.3.3. Developer team structure

Before the modernization, Case A had been using a team-based approach, with a sprint-based rotation that allowed all of Case A's developers and testers to get familiarized with all of the parts of Case A's product. But with the onset of the modernization, this came under scrutiny, as the new methods, especially in relation to microservices, allowed for more modular and segregated development practices: Specialization on specific parts of the software was now more viable, and doing so would allow for faster turn-around on new features, testing and bug-fixing. This would also simplify role-division, as specialization would enable more stability between sprints.

The structure for the teams was given consideration, and what was eventually chosen was a modified Squad based structure used by Spotify [KIS12]:

- Squad: A modified agile Scrum-team, designed to mimic a small start-up, where each member of the squad works with each other in close proximity who self-organize and decide on their own methods on how to go about development. The members of a squad have the tools, skills and knowledge to design, develop, test and deploy their part of the software into production independently. A squad has a long-term mission of some kind within the software, such as improving and scaling the UI, handling the back-end scaling or developing a specific business logic part of the application. Squads do not generally have a formal lead, but they do have a product owner, whose primary duty is the prioritization of the squads workload. These product owners communicate with other product owners within the organization to maintain a roadmap of overall development for the organization, and they also maintain matching product backlogs for their respective squads.
- Tribe: A tribe is a collection of squads, preferably amounting to less than 100 people or so. Each tribe has a tribe lead, who will work to provide the best habitat for each of the squads within the tribe.
- Chapter: A chapter is a cross-squad team within the same tribe, with similar areas of competence and responsibility, such as: Testing chapter, UI chapter, backend chapter.
- Guild: A guild is a more general and flexible chapter, which can comprise members from multiple squads from multiple tribes. A guild comprises people who have the same areas of inte- rest, who want to share resources and knowledge about this area of interest. This area can be about testing, agile coaching, web technologies etc.

While the Spotify squad model was taken as a basis for the new structure within Case A, it ended up getting modified towards the end of the modernization: Case A ended up mainly using squads and chapters. This was mostly due to Case A being relatively small, so adopting tribes and guilds would not have been very feasible for the organization.

After the modernization, Case A has 4 squads: Two specialized software development squads for the two main parts of the software, infrastructure software development squad as well as a sales and management squad. Each development squad also includes testers and support personnel.

3. Research Methods

In this chapter we cover what research methods we used to find information on modernization and legacy systems, as well as what methods we used to gain data about the modernization efforts of Case A in specific, and what the experiences of the employees were on this modernization.

3.1. Research environment

The study plan for the thesis consisted of both an e-mail survey and several in-depth questions, as well as the literature reviews from the background material as data collection methods.

Case A was introduced briefly before, but it is useful to add in additional information about the company, as it was the research environment for the thesis: Case A is a small-sized company, with around 24 employees, accounting for small fluctuations. The software development teams account for about 70% of this personnel, including testers and the infrastructure team. Case A began its operations in the early 1990's, but it pivoted to being mainly a software development company in 2016, so considering its long operational history, Case A's period of time as a software developer has been relatively short.

3.2. Survey and in-depth questionnaire

We used an email survey and several in-depth questions with the developers and other employees within Case A to gather information on the modernization effort within Case A.

The survey was originally sent to 24 of the employees within Case A, 12 of whom were developers and 12 who were in software support or lead positions. Out of the 12 developers, 3 were senior developers, one was the CIO of Case A, one was the solution architect of Case A, 2 were lead developers, one was a reporting specialist, while the remaining 4 were software developers. The exact composition of survey recipients and other details is shown in Table 1. Software Developer 1 and 2 are marked separate from the others, as they answered the in-depth questionnaire.

The survey covered questions on what effects the new tools and methods had on the employees' work and what were their previous experience-levels on the tools and methods. The survey also covered opinions

on whether the new tools were helpful, and how effective the restructuring of the teams within Case A was for them.

In addition to the surveys, there were 6 in-depth questionnaires sent over email, which had more specific questions about the modernization change and their experiences with it. These were sent to 2 software developers, Software Developer 1 and Software Developer 2; The in-depth surveys were also sent to 2 other software developers, a senior software developer and the CTO, but without replies from these recipients.

Table 1: Table of the survey recipients' job titles in Case A, categorized between software/architecture development, software support and management personnel. Rows listed in approximation of the recipient's duty-load in Case A in descending order.

Development	**Software support**	**Management**
CIO	Senior Customer Service Manager	COO
Solution Architect	Customer Service Expert	CFO
Senior Software Developer (3)	Quality Assurance Specialist (3)	CCO
Lead Developer (2)		Senior Development Manager
Software Developer (2)		Project Manager
Software Developer 1		Product Owner
Software Developer 2		Director of People and Culture
Reporting Specialist		

Survey composition

The very first question of the survey was to establish the role the recipient has within Case A, be it Developer, Tester, Product Owner or other, as this sets the basis of the weight on how the new technologies and methods used in modernization might affect them.

The email survey was then composed of questions with a rating of 0 to 10, asking the recipients on the effect of the new technologies and methods on their work, and how much previous experience they had in using those technologies and methods. They were also asked to rate on how much of an effect the general modernization effort had on their work, and to rate their general opinion on the new tools and methods.

The technologies and methods that were covered in the survey were Microsoft Azure, Angular, Docker, Syncfusion, Material Design, OpenAPI/Swagger and Dotnet Core. We also asked the recipients to rate how the team restructuring affected them and how they felt about it.

Additionally, there was a free comment question at the end of the survey, for any additional comments and notes the recipients might have had.

The full survey is attached to this thesis as Appendix A.

Questionnaire protocol

The in-depth questionnaires sent to the recipients focused on getting more granular answers from specific recipients. The in-depth questions consisted of three main questions, and one free comment field. In the questions, we use the word modernization to refer to the software modernization in specific. The questions were:

> Q1: In general, what are your opinions on using the new tools after the software modernization finished?
> Q2: Why was modernization important for the company?
> Q3: If another modernization change should take place in the future, would you like to change anything in the implementation or planning of such a change?

With these questions, we aimed to gauge the acceptance of the new tools and the opinion on modernization and how it should move forward.

The in-depth questions were sent specifically to the development team, the titles of which are shown in Table 1 the Development column, as they could provide an expert's opinion on the matter.

The in-depth questionnaire, and its answers, is attached to this thesis as Appendix B.

3.3. Literature review

For the purposes of this thesis, we used a literature review to find out more about previous attempts at modernization, what pitfalls they can have and what methods and techniques could be used for such a change. The search for literature to use in this thesis started by using the Scopus citation database [Sco04] with "(usability OR 'user experience') AND agile" as the initial search terms, with another search term "modernization AND 'computer science' " used afterwards. These initial searches together yielded a good amount of starting literature from which to begin research, including [Kal17] and [Ber99]. In addition, white papers published by companies specializing in software modernization were used to gather further data, such as [Alt19]. Information was also gathered directly about the tools from the specific companies'

websites providing the tools, such as [Azu10] and [Mat14]. Supplemental data was gathered mostly through snowballing backwards and forwards from the original materials' references.

Modernization strategies

As several enterprises and companies have become keenly aware of the demerits of clinging to old technologies, several different modernization strategies, some concerned with a decomposition of a monolith into microservices [Kal17,FB19], have appeared over the last two decades [Mal10,Alt19]. In particular, Balalaie et al. [BHJ15,BHJ16] outline several microservice migration patterns, including the problems, solutions and situations where they could be used.

Because different companies are often in differing situations, with varying team skills, resources, requirements etc., there is no silver bullet strategy that can work with all situations, so a situational-method-engineering (SME) [HaA97] approach is usually needed to find the correct strategy for the scenario [BHJ15].

Seacord et al. [SPL03] outline three high-level modernization strategies that can be used for software modernization:

- Maintenance:
    - This is an iterative and incremental strategy that involves UI/UX improvements, performance optimization and database migration. These changes are usually small, and business logic and core architecture is usually unchanged. Migration to using cloud-based services can happen in this strategy.
- Modernization:
    - More extensive than Maintenance, but still conserves a significant part of the existing system. Seeks to do enhancements to the core product: Architecture optimization, code refactoring, UX updates or general performance optimizations are included, without significant changes to the core business logic. May include restructuring of the existing software.
- Replacement:
    - Existing legacy or outdated software features of the software system are identified, and then the features that are to be kept are re-created from the ground up, using modern technologies for improved performance. This is the most time and resource consuming strategy, as it requires the complete re-creation of existing features. This strategy can be

implemented either incrementally over time, or by doing it all as one large project and getting all the features replaced when the project is finished.

These high-level strategies offer a guideline on modernization, and there are multiple ways to perform each of these, depending on the needs of the software system and company in question.

These strategies can further be categorized into two different categories: Black-box modernization and white-box modernization [SPL03]: Black-box modernization involves evaluation and investigation of the inputs and outputs of the system that is being modernized, and knowing how its interfaces work. Using this information, the system can be, for example, wrapped by a new software layer that hides the old system's complexity and can then use a modern interface [SPL03]. This can be achieved by using Azure or other cloud-based solutions, and having the outputs be interpreted and analyzed by them; Azure Application Insights can be used for this. While black-box modernization can only involve inputs and outputs of the system, it may become necessary to alter the underlying code, to add telemetry event hooks for Azure to use, for example. In this case, the line between black-box and white-box modernization can become less distinct.

White-box modernization on the other hand is more involved than black-box modernization: This is the process that is often referred to as software reengineering, and white-box modernization requires understanding the internal procedures of the system to be modernized, rather than only its inputs and outputs. Software reengineering in the case of modernization can be defined in a way that the old software system is to be fully evaluated, and then rebuilt in a new more modern format. This procedure can take the form of replacing the entire system, or only parts of it with the new, rebuilt parts, and then this new form in its entirety is implemented into production use [Mal10].

Modernization methods and techniques

Aside from the general strategies and categories in which they belong, there are also multiple different modernization and migration patterns, methods and techniques that developers and management can choose from when they are implementing and planning the modernization. We cover some of these techniques next, as there is a good deal of variance in the ways a modernization can be implemented with them.

There are multiple different modernization techniques that can be used in modernization, fitting different situations and used technologies, as explained by Altexsoft [Alt19]. The first modernization technique is encapsulation, which is a wrapping modernization technique [SPL03]. In this technique, the old legacy

system is left mostly intact, and is then accessed via an API. This is the least invasive technique of modernization, but it does not address problems already present in the old system itself. It is a useful technique when the underlying code of the old system is of high value, and there are no significant issues in continuing to use it.

Another technique that can be used is rehosting, which is a data migration technique, where the old system is hosted on a different physical, virtual or cloud-based infrastructure without making changes in the underlying code. This makes it a relatively fast technique to use in modernization. It offers flexibility, lower physical space constraints, greater scalability and better stability, depending on what hosting solution is used: Rehosting to a cloud-based infrastructure can offer the greatest benefits these days, and can also be used for replatforming and refactoring techniques. Cloud-based infrastructures can have their own issues, however, as addressed in chapter 2.3.2.1.

There is another data migration technique aside from rehosting: Replatforming. In this technique, the platform of the system is migrated to a new runtime platform while making only minimal changes to the underlying code, leaving its structure unchanged. This technique can be used for cloud migration without making the system being modernized completely cloud-native, which would also include refactoring the underlying code.

Which leads us to the next technique, that of refactoring: This technique involves restructuring and optimizing the existing code without changing its external behavior. This can remove or mitigate fossilization and technical debt from the codebase. By updating the codebase, the organization undergoing modernization can take full advantage of, for example, cloud-based features, if cloud-migration is also a consideration during the modernization.

Yet another technique for modernization is rearchitecting, which is the most synonymous technique to re-engineering, and involves gathering requirements from existing legacy codebase/application and redeveloping them on new platforms using new techniques [Mal10]. Examples of this technique would be re-engineering to use service-oriented architectures or microservice architecture.

There are two modernization techniques that seek to remake the application from the very beginning: Rebuilding and replacing. Rebuilding involves rewriting the application completely, while retaining the scope and specification of the original application. This allows for redesigning the application to involve new features, functionalities and processes that can be used with the new technologies being adopted.

Replacing is more drastic than Rebuilding, as it will replace the existing application and codebase wholesale, with a new scope, specification and design, rather than attempting to modernize its parts.

Table 2: Table on different modernization techniques and methods [SPL03, Mal10, Alt19].

	Description	**Pros**	**Cons**
Encapsulation	Old system is left intact, and an API is added to access it.	Structure of the code is left intact. Fastest and least invasive technique.	Does not fix problems present in the system.
Rehosting	Old system is hosted on a new infrastructure, without modifying the internal code.	Structure of the code is left intact. Fast and flexible, and can lower physical space constraints as well as improve scalability and stability.	Does not fix problems present in the system. New infrastructures can introduce additional issues.
Replatform	The platform of the system is changed, with minimal changes to the underlying code.	Structure of the code is left intact. New platforms can offer new solutions, such as cloud-based platforms.	Does not fix problems present in the system. New platforms can introduce additional issues.
Refactoring	Restructures and optimizes existing underlying code without altering external behavior.	Can address many problems present in the system, removing fossilization. New code can take advantage of more modern features, tools and methods.	Time-consuming. Requires having the legacy code under control.
Rearchitecting	Redevelops the required functionalities of the old system on new platforms using new technologies.	Can address many problems present in the system, removing fossilization. New code can take advantage of more modern features, tools and methods.	Time-consuming. Requires having the legacy code under control.
Rebuilding	Rewrites the application in question from scratch, using the existing scope and requirements.	Can address all of the issues present that originate from the underlying code. New code can take advantage of more modern features, tools and methods. Only requires the original scope, requirements and design.	Very time and resource consuming. Does not address issues stemming from scope, specification or design of the system.
Replacing	Rewrites the application in question from scratch, using the new scope and requirements.	Can address all of the issues of the system. New code can take advantage of more modern features, tools and methods. Requires no prior knowledge of the original system.	The most time and resource consuming technique.

As can be seen, there are multiple different techniques on how to approach modernization, and they impact the application in different ways: Encapsulation, rehosting and replatforming concern the technology platform of the application, while refactoring and rearchitecting can solve issues in the codebase and architecture itself. Rebuilding and replacing on the other hand are more comprehensive changes to the application, which can involve changes in all aspects of the system and its peripherals [Alt19].

The choice of modernization technique depends on the application being modernized: Encapsulation, rehosting and replatforming fit applications that have no issues in the existing codebase, whereas refactoring and rearchitecting can be used to fix underlying code-issues, as mentioned before. If there are deeper issues with the application, rebuilding can fix these issues, as long as the overall design and specifications are still valid; If not, then replacing the application wholesale should be considered, as the necessary designs and specifications can be much different at the time of modernization than they were at the onset of the application. A short summary of the modernization techniques is listed in Table 2.

Many third-party companies offer commercial-off-the-shelf (COTS) solutions to modernization, which can mix and match many of the technologies and strategies examined here, but these packages can be limiting should the organization undergoing the modernization wish to reuse existing business logic or have a more direct hand in the modernization [Mal10].

Microservice migration patterns

Modernization through migration to the cloud by using microservices is a very difficult task in itself, even without considering the other techniques and strategies of modernizing software otherwise. As with modernization in general, microservice migration can happen in many different situations, in different kinds of organizations with differing skills and technologies in use, and as such the situational-method-engineering (SME) is proposed [HaA97]. To facilitate this, Balalaie et al. [BHJ15,BHJ16] propose 15 migration patterns with associated technology suggestions from which the method engineer in charge can select and compose the required ones, based on the modernization project's requirements, constraints, scope and other requirements; From this, an overall migration plan that is specific to the current situation can be crafted.

The following microservice migration patterns, or more accurately steps to help finish a microservice migration, are defined by Balalaie et al. [BHJ15]:

Table 3: Table on microservice migration patterns [BHJ15].

Pattern	**Description**	**Technology Stack**
Enabling the Continuous Integration	Continuous integration (CI) allows the developers to integrate their work with other developers' work as early as possible, helping to prevent future conflicts with each others' work. This is the first step towards continuous delivery (CD).	Gitlab, Artifactory, Nexus, Jenkins, GoCD, Travis, Bamboo, Teamcity

Table 3: Continued

Recover the Current Architecture	Understanding the current architecture of the system is important in planning for migration, as it allows the developers and planners of the migration to understand the big picture scope of the system and information is sufficient for migration planning. In specific, understanding the components, service, technology and deployment architectures are important to consolidate a common understanding of the whole system. Documenting the current architecture is optional, as it will cease to exist after the modernization; Rather it is important that the team understands the architecture while the modernization is taking place [BHJ15].	
Decompose the Monolith	Decomposing a monolithic system with a complex domain into smaller chunks is an important task, and how to do this and how big the resulting chunks should be is a non-trivial issue. For example, domain-driven design can be used to identify subdomains of the business the system is operating in, as is a good candidate for the initial decomposition. After the initial decomposition, each subdomain can constitute separate bounded contexts which represent deployable units, the size of which can vary and change.	
Decompose the Monolith Based on Data Ownership	Decomposing a monolithic system based on data ownership involves finding distinct sets of data entities in the monolith that can be grouped together and have a single unique owner, which are then packaged together into being a service. Each of these services can then be modified and created by only their corresponding service. The size of the services created through data ownership decomposition are heavily contextual, and depends on the entities located within the system.	
Change Code Dependency to Service Call	After a software system has been decomposed into a microservices architecture, it becomes important to decide when it is and when it isn't appropriate to change any code-level dependencies to service-level dependencies. Keeping the services code separate is recommended, as build processes of other services may fail as a result of shared code. Sharing functionalities as separate, isolated services or as dependent services is a good practice, as this also allows for independent scaling of the services and decouples the services code.	
Introduce Service Discovery	Service discovery stores addresses of each service instance, as the new microservices need to be able to locate each other dynamically: When a service is first initiated it registers itself and then sends periodic heartbeat signals to the service discovery. This allows the registry to contain a list of all available services at all times, which can then be easily located from an edge server, load balancer or any of the other connected services.	Eureka, Consul, Apache Zookeeper, etcd
Introduce Service Discovery Client	Each service should know the address of the service discovery and register itself to it during the service's initiation. They should then follow the heartbeat signal procedure as above, and as long as the heartbeat signal is sent, it is maintained in the service discovery's list of services. The service instance can then be terminated by stopping the heartbeat signal from an instance, or explicitly informing service discovery of the instance's termination.	Eureka is a Service Discovery which has a Java client implementation for its server version.
Introduce Internal Load Balancer	An internal load balancer can be implemented in each of the services, which fetches a list of available services from service discovery. The load balancer can then balance the load between instances using local metrics, so that none of the services are under undue stress: Using this, each service can have their own load balancing mechanism that is specific to their context. This method is not centralized code-wise, and a separate load balancer needs to be made for each of the programming languages in use amongst the services.	Ribbon is an internal load balancer for Java that works well with Eureka, a Service Discovery.

Table 3: Continued

Introduce External Load Balancer	Alternatively to the above, an external load balancer is implemented as a separate component, such as a proxy or an instance address locator, which still uses service discovery to gather the list of service instances. There are some service discovery options with their own load balancer built in, which could be used instead of another separate component. With an external load balancer local metrics cannot be used, and neither can different clients have their own customized load balancing strategies.	Amazon ELB, Nginx, HAProxy, Eureka
Introduce Circuit Breaker	A circuit breaker can be a proxy to a remote client which monitors the recent responses from the service provider. This solves an issue with failing fast in cases where a service is unavailable, as when a service is available the circuit breaker does nothing and is in a closed state, but when a service is unavailable a response or an error-message is sent to the end-user: This state is called an open circuit, and is achieved when enough failed responses have passed a user-defined failure threshold. A retry connection can also be included in the circuit breaker, after a user-defined timeout.	Hystrix
Introduce Configuration Server	To modify the running service instances without redeploying them after every change, a separate repository for software configurations can be used. While there may be a need for synchronizing the repository with the source code repository when changes happen in the configuration keys, they should evolve independently from each other. Any changes in the configuration repository should be propagated to the corresponding running service instances.	Spring Config Server, Archaius
Introduce Edge Server	Having an edge server applies a layer of abstraction that can handle dynamic routing based on predefined configurations, with the service instance addresses for routing the incoming traffic being fixed, hard-coded or fetched from service discovery. This solves the issue of hiding the complexity of the system from the end-user, as they would only be interacting with this abstraction layer, and as such would never be exposed to the complexities of the internal structure of the system. Usage and status of the system can also be monitored within this edge server layer, as all traffic between the users and the system goes through this layer.	Zuul
Containerize the Services	Containerization is the act of creating container images for the services and storing them in a repository during the continuous integration pipeline. This can solve issues stemming from developers having different software environments, as the container images can be run on any environment and produce the same results in each of them [PoN20].	Docker
Deploy into a Cluster and Orchestrate Containers	A cluster of computing nodes is difficult to manage, therefore a management system that can deploy the services container images on-demand, with a specified number of instances on different nodes is recommended. This cluster management tool should also be able to handle service instance failures and provide a means for auto-scaling the services it manages, as well as providing the means to define the deployment architecture of services declaratively.	Mesos + Marathon, Kubernetes
Monitor the System and Provide Feedback	Monitoring and logging to gather information, e.g. CPU and RAM usage, and sending them to a monitoring server should be added for each of the services. In the monitoring server, this information can then be parsed and aggregated into information that can be queried efficiently by the services team. This helps development to refactor the architecture and code to remove performance bottlenecks and other anomalies.	Collectd + Logstash + ElasticSearch + Kibana

The patterns suggested in Table 3 follow a step-by-step process of microservices migration process, with technologies and methods of how to avoid possible pitfalls in a generic microservices migration process. While the process is contextual for each specific system, as there is no silver bullet for all systems, it does

provide a good guideline to follow on what steps should be taken during the process: The method engineer, or the team of planners for the migration, can select what patterns they require and use only those from the repository illustrated here; It is important that the planners and engineers understand the contextual needs of the migration. Adding to the repository of these migration patterns is encouraged, as new patterns and contexts can only augment the set described here [BHJ15]. The examples in the proposed technology stack for each pattern is very flexible, for example there are a great many monitoring solutions available, such as Azure Application Insights [Azu10] as used by Case A.

4. Risks and challenges of software modernization

As with many large changes in operational and technological practices, modernization can carry a great amount of risks and challenges that must be taken into account and overcome. In general these can be separated to organizational challenges and architectural or technological challenges.

Bergey et al. [Ber99] list several key reasons why modernization efforts can fail:

Table 4: Table on the key reasons for modernization effort failure, as well as the sub-chapter where it is covered[Ber99].

Reason for failure	**Subchapter(s) that covers the failure**
The organization inadvertently adopts a flawed or incomplete reengineering strategy	Subchapters 4.1. and 4.5.
The organization makes inappropriate use of outside consultants and outside contractors	Subchapters 4.2. and 4.7.
The work force is tied to old technologies with inadequate training programs	Subchapter 4.2.
The organization does not have its legacy system under control	Subchapter 4.3.
There is too little elicitation and validation of requirements	Subchapters 4.3. and 4.4.
Software architecture is not a primary reengineering consideration	Subchapter 4.8.
There is no notion of a separate and distinct reengineering process	Subchapter 4.9.
There is inadequate planning or inadequate resolve to follow the plans	Subchapters 4.1. and 4.7.
Management lacks long-term commitment	Subchapters 4.4. and 4.7.
Management predetermines technical decisions	Subchapters 4.4. and 4.7.

We will cover the reasons listed in Table 4 in more detail, along with other challenges and risks in the following subchapters.

4.1. Modernization strategy and ensuring improvements

One of the initial greater risks of going forward in modernization, is going forward without a clear planned-out strategy [BHJ15]. In such a situation, the process becomes more trial-and-error, and is prone

to wasting time and even leading to a wrong solution [BHJ15] that either does not work correctly or does not improve upon the existing system.

A flawed or incomplete modernization strategy can lead to severe issues in the modernization: Sometimes, the problems that occur during modernization stem from the fact that wrong issues are being addressed instead of issues that were the actual problem [Ber99], and sometimes all of the components and steps aren't being addressed in the strategy.

This leads to another risk: Ensuring that the planned modernization improves the product in some aspect. A good strategy greatly mitigates the risk of this happening, but as modernization and reengineering can take a long time depending on the scale of the product being modernized, it is possible that by the time the modernization process ends the planned improvements do not match the current situation, and another round of modernization has to take place.

In general, if the starting strategy adopted for the modernization is flawed in some respect, it can have wide-ranging consequences to the modernization effort at large. As these strategies represent high-level choices for the modernization, it is paramount that a modernization effort is well-thought out at its onset [Ber99]. Analyzing and assessing the state of the legacy software should be a part of this strategy, as the legacy software system might actually serve as a viable basis for the modernization instead of something to replace wholesale during modernization [Alt19].

4.2. Personnel and training

A challenge and a bottleneck for many modernization efforts can be the personnel of the organization: Developers, testers, product owners, project managers etc. must all be trained and vetted in the use of the new technologies and methods adopted by the organization during modernization [Ber99]. Coaching and motivating personnel to adopt new technologies and methods can sometimes even create pushback from the personnel [Alt19], especially in the case of a 'big bang' modernization, where the modernization is conducted in one large chunk as a single project. The risks originating from personnel adoption and compliance can be lessened by employing continuous learning (CL) [NaA17], which is also known as lifelong learning, methods and using incremental and iterative development methods for modernization rather than 'big bang' style methods, which focus on making all the changes as one single project, as this allows all the personnel to learn the new methods in smaller pieces, which helps adoption of the new methods [Ber99].

New personnel and consultants brought specifically for the modernization effort can also prove a challenge [Ber99], as they can lack the business-specific knowledge of the software required for proper implementation of the modernization project. This can then translate to these personnel requiring more training about the base software and the business practices, as well as having to be carefully monitored by employees and managers who have business-knowledge of the base software.

4.3. Documentation

One of the biggest risks and a challenge in any decently-sized change in operational procedures or adoption of new technologies, is documentation [Ber99]. Even in the case of smaller scale changes in software and practices, good documentation and knowledge-base that is available to the personnel can greatly alleviate problems with adoption and usage of those new methods. On the flipside, missing and incomplete documentation can cause severe breaks in communication between the personnel, as there can be cases where only one person actually knows how the new methods work, causing their workload to increase as they must then teach the other personnel. The worst case scenario in situations like this is, if the only person who knows the new process leaves an organization undergoing modernization, either permanently or temporarily: This can cause a gridlock effect within the organization if any problems arise with the new methods, as there is no-one knowledgeable enough to investigate and fix it.

While the above low-level documentation pertaining software and technologies are a challenge even in normal development, in a modernization process the lack of a documented project plan or a road map can prove an even larger challenge [Ber99]. Because tactical management must be focused on a higher level than the technologies used in a modernization effort, which is often planned by a team of interdisciplinary engineers and experts, poor documentation of the implementation plan and roadmap of the modernization can lead to severe issues during or after the modernization. This may be a result of members of the organization changing [Ber99], or even from simple human error, as undocumented plans that only exist in the teams' heads can decay and be misremembered.

4.4. Communication

Related to the challenge of documentation is communication: If the plans and roadmaps of the initial team that plans the modernization are not adequately spread amongst the other members of the organization, it is very possible that misunderstandings, lack of required knowledge and even disgruntlement can occur [Ber99]: Poor and incomplete communication distorts the whole modernization effort. This can occur bet-

ween different levels in the organization: If the management does not properly communicate with the developers, or if the organization does not communicate properly with the users of the software, there is a distinct risk of problems occurring. For example, the management or outside consultants can choose what technical solutions, schedule, cost and performance the modernization effort should use by themselves without sufficient project team input or communication. Without grassroots communication, this approach can cause the modernization effort to fail [Ber99], as detailed planning of schedules and milestones can only be reliably determined by studying the technical parameters of the system, which requires understanding the system and having specific business knowledge of the existing system.

4.5. Accessibility

One of the larger challenges for the modernization within Case A was that the software product was required to be continuously accessible by current users and customers, without too much downtime that might result from adopting new technologies and methods. In general, this means that an organization going forward with modernization must allocate manpower for both the modernization and also the upkeep of the current system. Without hiring new people, or external consultants and experts, this places an extra burden on the current developers and their teams, which can result in slower production cycles and less work done.

Accessibility can also become an issue when it is time to integrate the new technologies and methods to the old system, or when moving the whole system onto different frameworks: A poorly planned-out integration can cause significant downtime or errors for the customers who are using the system [Ber99].

This is an important concern for SaaS products, such as what Case A provides, as customers who have signed deals with such providers expect to receive what was prescribed to them in the contract they made with the provider, regardless of any internal reorganizations or modernizations.

4.6. Integration

Integrating the new technologies coming from a modernization effort with any existing systems the organization has can become a roadblock for the whole project, especially if the organization does not have its legacy system under control [Ber99]. This lack of control can be attributed to lack of documentation of the legacy system, but can also stem from lack of understanding it. Aside from documentation, there

should be data on the maintenance costs [Ber99], configuration management, planning and project management capabilities of the legacy system.

This thinking also extends to the other systems that the organization is using, as there can be non-legacy systems that need to communicate with the new technologies the modernization effort brings into use, and they also need to be understood and documented just as well as the legacy systems that are likely being phased out with the modernization.

4.6.1 Databases

Integrating and migrating data to new databases, and providing new database solutions during the modernization effort can prove to be costly and time-consuming, as this is often manual work requiring gathering and systemizing all the legacy data [Alt19].

Choosing a correct database solution is very important to avoid problems down the line [Kal17], and for microservices in specific, there are two main database patterns: Database by service and shared database [Ric15a, Ric15b].

Database by service pattern keeps each microservice's data private to only that service, and can only be accessed via its own API, and the service's transactions only concern its own database. No other services can access this service's database. This can be achieved by having each service own a set of tables accessed by that service, each service having a database schema that is only for that service or each service having its very own database server. There are several benefits to this database pattern: All services are ensured to be loosely coupled and each service has a specific type of database best suited for their needs. There are also several drawbacks as well, such as increased complexity in implementing business transactions between different services [Ric15a], implementing join queries between multiple databases is difficult and the complexity of having to manage multiple different databases.

Shared database pattern in contrast to database by service uses a single database that is shared by multiple services, and each of those services are free to access the database and data of other services. The benefits of this pattern include simplicity, as everything is handled by a single database, as well as having a good rate of data consistency. The drawbacks, however, are that the services will be more tightly coupled, which has to be handled during development and runtime, as each service has the chance to interfere with other services. The chosen database might not be well-suited for all services either [Ric15b], which brings additional concerns regarding data storage and access requirements.

4.6.2 Testing

There are also new challenges that stem from testing, as the old tests, which normally consist of unit tests, integration tests and end-to-end tests, from before the modernization need to be remade and tested in turn before they can be put into use with the new system. This is in addition to completely new tests that the new system might require, which in the case of microservices would include component and contract testing on top of the existing tests. While the basic tests do not differ greatly between microservice architectures and monolithic architectures [Kal17], it is necessary to understand that they are still different.

As the microservices need to communicate with each other, databases and APIs, they introduce new communication channels and boundaries between them [Kal17], which have to be taken into account during the formulation of the testing strategy. Also, because these new communication channels and boundaries can introduce new performance issues [Kal17], performance testing of different grades needs to be considered for the testing strategies: End-to-end service calls should be considered as well as service calls between single services.

Automated tests may need to be completely changed during modernization, but a modernization project is also a good time to review any manual tests that an organization uses, and see if those can be automated for further efficiency: Automated testing is more important in a microservices architecture, as the release cycles are likely going to be shorter than on a monolithic architecture [Kal17], dissuading manual testing. But as with all parts of the modernization, testing strategies must be carefully formulated at their inception, so as not to create flawed strategies that may not fit the result.

4.7. Economy

Large-scale modernization efforts can be very costly for an organization, depending on the overall strategy chosen for the modernization. As with other challenges, a flawed strategy can end up costing the organization a great deal of time, money and effort [Ber99]. These costs may not be reciprocated in the results of the project, and this can make management apprehensive of investing in modernization, as it can be seen as throwing money into a black hole. As project costs, schedules and deliverables can be predetermined by the management [Ber99], it can be very tempting for them to set specific deliverables and timetables without consulting the project team, which can lead to poor results and a loss of resources.

Improper use of consultants and contractors can result in a strain on resources: While outside consultants can provide great benefits to a modernization effort, they rarely have as much business knowledge as the insiders of the organization [Ber99], which can lead to misunderstandings that can prove costly for the organization. This can also be reflected in management using consultants improperly: If multiple consultants are used by an organization, they may all find the same or similar issues, but even after they have been found, the issues persist. This may be because their reports on the issues are rejected for being biased [Ber99], because the consultants do not have enough experience or credibility, or even because they are not given enough time to actually address the found issues.

4.8. Software architecture is not a primary consideration

If there is no consideration for methodically evaluating the software architectures of the new and the old system, and having that be one of the driving forces behind the modernization effort, it is possible that a failure may occur in the modernization effort [Ber99]. Evaluation of the old architecture is important, and even necessary, to know whether or not the old architecture is viable for further development; If the architecture is not viable, starting over from scratch can be considered. This further links to documentation, as per subchapter 4.3., because the legacy architecture needs to be understood, and documentation enables this: If the documentation on the old architecture is reliable, it can then be extended to encompass the new architecture.

If there is no evaluation of the old architecture, it is highly likely that there will be inconsistencies between the legacy and target systems, which leads to problems [Ber99].

4.9. There is no notion of a separate and distinct reengineering process

There are four critical elements to any modernization effort: People, technology, process, and available resources [Ber99]. Should even one of these elements be lacking, the end-product of the modernization effort may be of dubious quality. While software modernization parallels normal software development, it must be considered separate from it [Ber99], on a different development track. If it is considered normal development rather than its own process, it can be easy to overlook one of the elements critical to a successful modernization effort, as normal development may not have such elements to consider.

4.10. Summary

There are a great number of complex challenges to consider in a modernization effort, especially in a move from a monolithic legacy software to a microservices. Many of these challenges can be seen to stem from faulty starting strategies and planning, while others can occur at any point within the modernization effort.

From the challenges posed in this chapter, the greatest pertains to implementing a proper modernization strategy at the outset of the modernization effort [Ber99, BHJ15]: As this strategy sets the roadmap, milestones and even low-level technological choices among other considerations, it is the key for either a successful modernization project, or a failed one; All other challenges and risks are somehow tied to the overall strategy, and as such it needs to be done correctly to ensure success. This is not to say that a modernization strategy is set in stone once made, even if it is a very good one, as the situation can change during modernization, necessitating changes or amendments to the strategy: Personnel can change, data may be lost, some key technology may become unavailable etc. A good modernization strategy therefore should have some flexibility in its implementation. For example, the technologies to be used in the project should have alternatives to choose from if the initial choices become unviable, or that there are backup personnel to fulfill project roles if the original members become unavailable. These can be accounted for by the modernization strategy, but it requires certain flexibility from it instead of a rigid adherence to the set rules.

With a good modernization strategy that is properly scoped and flexible to the modernization needs of the organization, the chances of a successful modernization are much increased, but it is still not the modernization strategy in itself that guarantees success: It is important to keep in mind that problems can occur whenever before, during or even after the modernization, and there is no silver bullet solution to them all.

5. Results

In this chapter we collate the results gained from the survey and in-depth questionnaire, as well as compare the gained results from these to the guidelines and advice gathered from the literature reviews.

5.1. Survey results

Out of the 24 employees the email survey was sent to, 7 replied to the emails. While this is only a third of the wanted replies, it does allow us to create a dataset that can be analyzed. A more thorough breakdown of the recipients of the survey can be found in chapter 3.2, Table 1. As the survey was sent out during a busy time of the modernization effort, it can be speculated that many of the recipients did not answer due to having no time.

The breakdown of the 7 respondents can be seen in Table 5, and the recipients who did not respond to the survey can be seen in Table 6.

Table 5: Table of the survey respondents' job titles in Case A, ordered as per Table 1.

Development	**Software support**	**Management**
Software Developer (2)	Customer Service Expert	Project Manager
Software Developer 1		Product Owner
Software Developer 2		

Table 6: Table of the recipients job titles of the survey who did not respond to the survey, ordered as per Table 1.

Development	**Software support**	**Management**
CIO	Senior Customer Service Manager	COO
Solution Architect	Quality Assurance Specialist (3)	CFO
Senior Software Developer (3)		CCO
Lead Developer (2)		Senior Development Manager
Reporting Specialist		Director of People and Culture

Due to the varied positions of the respondents, the survey results are charted in two parts: All respondents and only developers. This is done to have more granularity in the results, as non-developers may have no experience or experience no effect from the adoption of new technologies and methods. But as we also

included more general questions in the survey relating to the modernization, their responses are also important to consider.

The data used for the averaging and calculation of the results in this chapter are attached in Appendix C.

5.1.1 All respondents' results

From the results of the survey on how effective on the respondents' working methods the change to new technologies and methods were, we can see that the change to using Azure and OpenApi/Swagger had the most effects, when all respondents are considered, as illustrated by Figure 5. For the purposes of this thesis, we use an average usage effect estimate in Figures 5, and 7, which is the average of the respondents' own estimates on a scale of 1 to 10 on how much the listed technology or method impacted their work. In Figures 6 and 8 we use an average previous experience estimate, which is the average of the respondents' own estimates on a scale of 1 to 10 on how much previous experience they have had with the listed technology or method. The resulting values are rounded to the first decimal for easier readability.

The estimated average effect for the change on using both Azure and OpenApi/Swagger were 6.6, when averaged against all of the respondents' answers, so they had a marked effect on the practices within Case A. Docker and Angular had the highest effect after Azure and OpenApi/Swagger, with Docker averaging 4.7 and Angular averaging 4.6 on their effectiveness, according to the respondents of the survey. Among all the respondents, the three lowest effectiveness estimates were Dotnet at 3.7, Syncfusion at 3.6 and Material Design at 2.7.

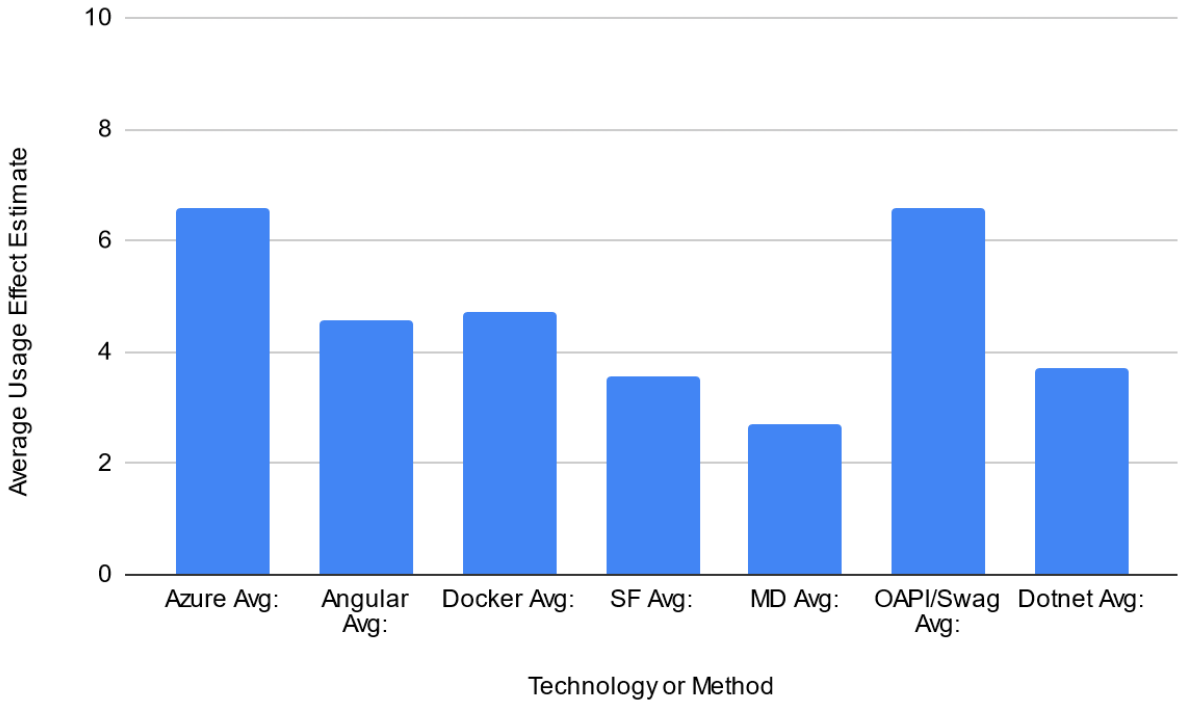

Figure 5: Average effectiveness on working methods among all respondents.

As for the previous experiences all of the respondents had from the new technologies and methods, very few of the respondents had much experience with the new tools, with the exception of OpenApi/Swagger, as can be seen from Figure 6.

From Figure 6 it is clear to see that OpenApi/Swagger was the one technology all respondents were most experienced with, with an average value of 4.4. The closest next technology was Dotnet at 2.7, with Docker, Angular and Azure tailing them with 2.1, 2 and 1.4 respectively. Material Design at 0.9 and Syncfusion at 0.6 were the technologies the respondents were the least familiar with.

In general, we can see that the majority of the respondents were either unfamiliar with many of the technologies and methods covered by the modernization, with the most knowledge being with the ones experimented with previously by Case A, with OpenApi/Swagger being the standout technology.

Despite the lack of experience as listed in Figure 6, we can see from Figure 5 that the effectiveness of the modernization on work had an appreciable impact.

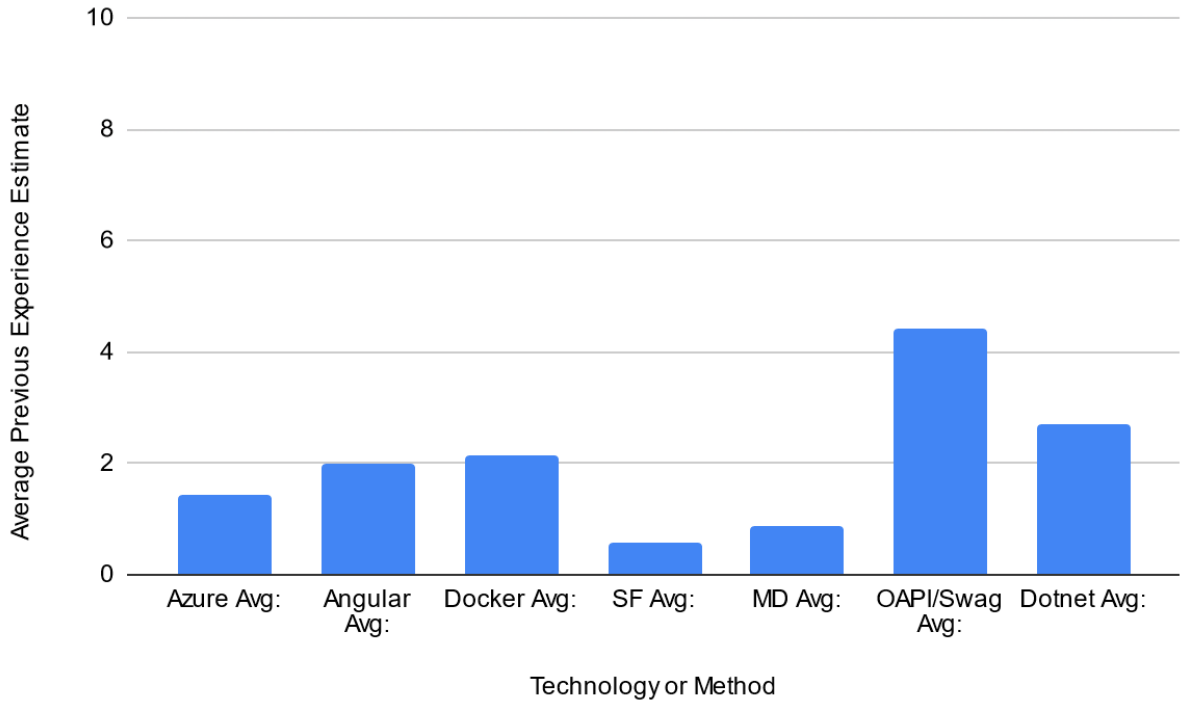

Figure 6: Average previous experience on technology/method among all respondents.

5.1.2 Developers' results

As many of the new changes in technologies and methods pertained to the usage of new programming tools and interfaces, the developers reported overall higher usage effect estimates on nearly all of the technologies, as can be seen in Figure 7.

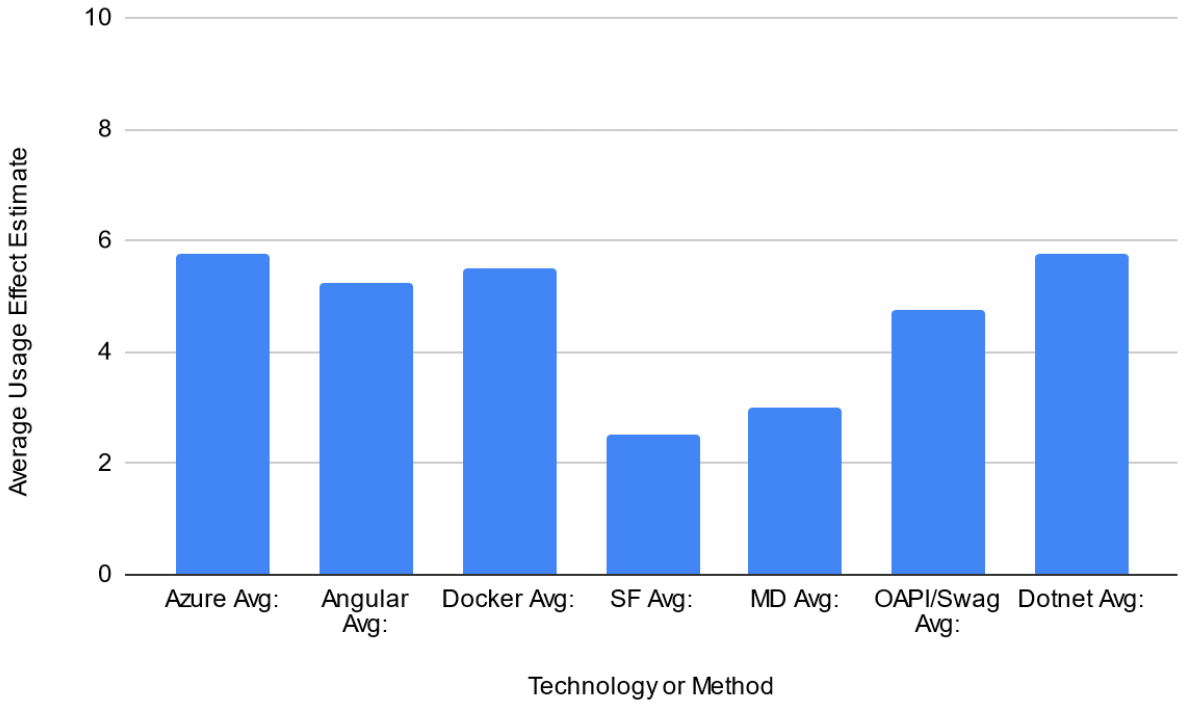

Figure 7: Average effectiveness on working methods among developers.

Much like the results of the all respondents on effectiveness of the modernization, the developers reported that the usage of Azure had a high effectiveness on their work, but a lesser impact from OpenApi/Swagger. The latter is likely a result of Case A having been using OpenApi/Swagger among developers for a longer time, so the modernization only added to its usage, rather than introducing something completely new for the developers. This is also reflected in figures 6 and 8, where the overall previous experience with OpenApi/Swagger is by far the greatest.

Out of the new technologies, Microsoft Azure and Dotnet Core had the highest effectiveness in developers' work, with an average of 5.8 each. Docker had the next most effect on work, with an average of 5.5, and Angular was close behind it in effectiveness with an average of 5.3.

OpenApi/Swagger had an average effectiveness among developers of 4.8, with Material Design and Syncfusion coming in with having the least effect in the developers work with 3 and 2.5 respectively.

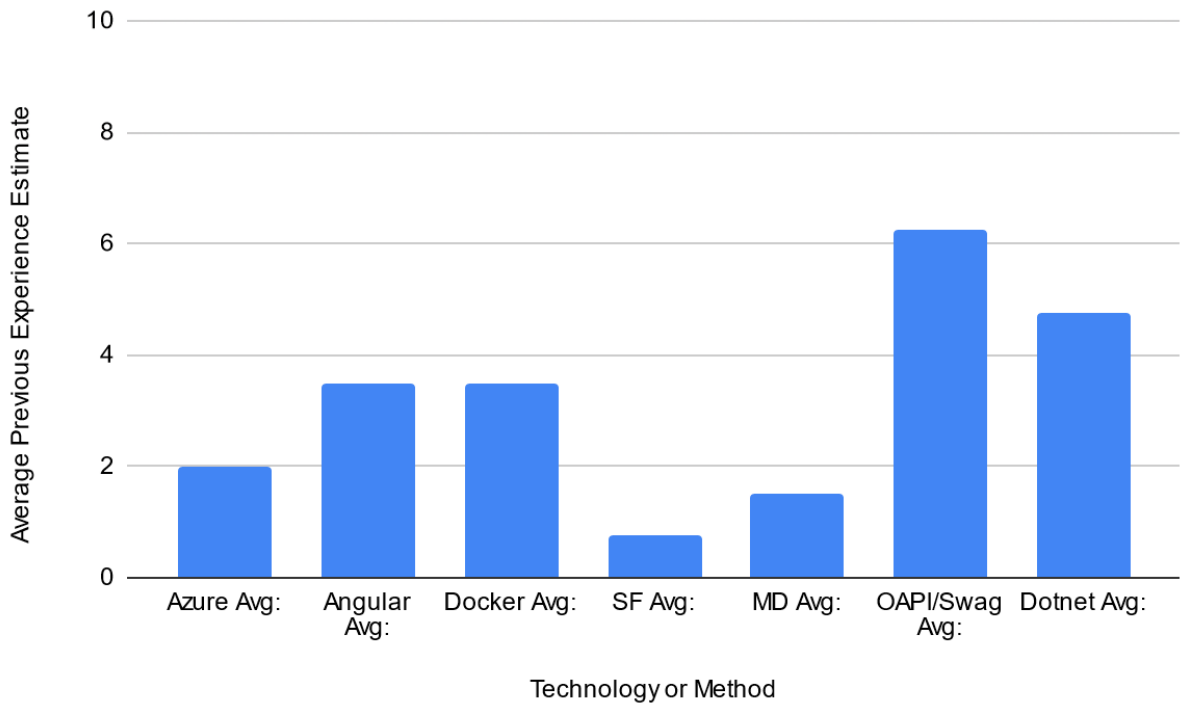

Figure 8: Average previous experience on technology/method among developers.

As was mentioned, we can see from Figure 8 that the developers had the most experience with OpenApi/Swagger before the modernization with a respondent average score of 6.3; As OpenApi/Swagger had been previously used by Case A by the developers in a more limited fashion, these results are not unexpected. Experience with Dotnet came in second with an average of 4.8, and Angular and Docker were evaluated at nearly the same experience levels, with both of them averaging out at 3.5.

Material Design and Syncfusion were the tools that the developers had the least previous experience with, coming in at 1.5 and 0.8 respectively; Experience with UI, UX and frontend elements among Case A's developers were not very high at the time of the survey, which explains the low numbers here.

5.2. In-depth questionnaire results

Out of the 4 employees the email in-depth questionnaire was conducted with, 2 replied with comprehensive answers. As with the survey, this allows us to create a small dataset from the answers. Both of the questioned employees were developers, one of them a senior developer.

The questions and answers from each of the respondents will be covered in order, with a short per question analysis based on their answer. We shall call the two respondents Software Developer 1 and Software Developer 2 respectively, as per Table 1 and Table 5.

For the first question concerning the respondents' opinions on using the new tools after the software modernization had finished, both respondents answered that new tools and technologies are of importance to the future well-being of organizations and its employees, bringing up improvements to development time and employee learning and future prospects; They generally held a positive outlook on new technologies and methods.

On the second question, concerning why the modernization was important for Case A, Software Developer 1 emphasized the importance for a SaaS company with a long history to transition to new technologies, as this could help build the software faster and more efficiently. Software Developer 2 pointed out the importance of modernization on remaining competitive and efficient in the long-term; From the answers, both developers seem to view modernization as important for the future of Case A.

As for the final question on what should possibly be changed on the implementation and planning in any future modernizations concerning Case A, both developers highlight the importance of a more iterative system of modernization, where smaller modernization efforts are constantly ongoing instead of making a large project in a short period of time. Software Developer 1 brings up the importance of planning, while keeping in mind the resources and time available.

In general based on their answers, both recipients agree that continuous development and iteration on using and finding new tools is very important.

It is interesting to notice, that both of the recipients also agree that continuous, smaller improvements and modernization via degrees is likely a good course in the future, rather than implementing larger modernization efforts. This is in line with the thinking found in iterative development, where each feature is developed in small chunks and tested continuously.

The full in-depth questions are included in Appendix B.

5.3. Overall modernization results

From the data in figures 6 and 8, we can see that while the overall experience level with many of the newly adopted technologies was low to average, they did have an appreciable effect on work, as can be

seen in figures 5 and 7. This may be exactly because the organization in general did not have much experience with the adopted technologies, as picking up new knowledge on how to use these technologies already causes an effect in work due to having to learn to use them.

Both Azure and OpenApi/Swagger were both relatively approachable by all of the respondents, whether they were developers or not, as both of these tools offer easily human-readable analytics and other data from the product, and OpenApi/Swagger had been in use as an event logger beforehand, offering insights into transactions happening between the users and the databases.

Event logging specifically changed a great deal within the organization, as previously it had been handled with a combination of Swagger logging from an external site, and holding internal logs within Case A's own database. With the advent of Azure, both of these old logging systems started getting phased out, and were being replaced with Azure's built-in event and transaction logging system: Application Insights. This was further reinforced with the usage of custom transaction logging codes within the codebase itself, enabling developers to track specific events. Swagger logging still remains in use, as taking Azure AI (Application Insights) into use everywhere takes effort and time.

5.4. Usability before and after modernization

From the customer usage perspective, we can infer the impact the modernization had on usability, performance and stability from the number of support tickets in Case A's Jira system that were issued by the customers on non-project issues: For the purposes of this thesis, we used the overall number of created customer tickets from before and after the modernization to measure the impact of the modernization on the customer.

Before the modernization started, in September 2018, there were 241 tickets created by the customers on a number of issues, which included technical issues that needed programming support to solve, guidance on how to use the system properly as well as issues that needed to be solved by a meeting with the customer and the Product Owners. Out of these tickets, 169 were resolved within that timescale. At the very start of the modernization, between the 1st and the 30th of January 2019, the tickets created rose to a high of 369, out of which 357 were solved during this time. This bump also includes the seasonal post-holiday high, which explains some of the tickets.

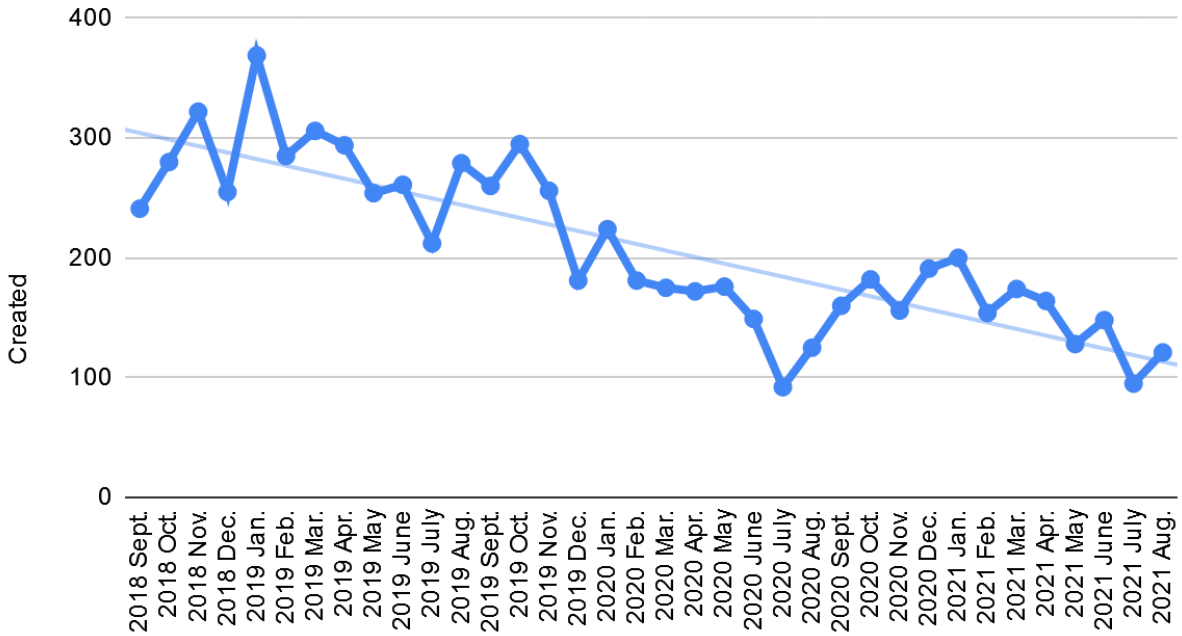

Figure 9: Trend of Jira ticket generation at the start of the modernization and after it.The dips in July, June, August and December are due to seasonal holidays, with peaks of activity following them.

After a good deal of the modernization was already done, between 1st and 31st of July 2020, the numbers were down to 92 created and 98 resolved. Finally, at the end of the tracking for this thesis, between the 1st and the 31st of August 2021, the numbers were 121 created tickets and 108 resolved tickets. An illustration on the trend throughout the modernization can be seen in Figure 9.

While it would be tempting to infer that this correlation between the modernization time and the created ticket count is a direct result from the modernization itself, there are other factors that are at play here that are more difficult to calculate the impact of: A medium-sized customer of Case A ended its contract with Case A in 2019-2020, which had its own impact on the ticket count, but the analysis on how large this impact was is difficult to calculate, as during this time period new contracts were also made with new customers. Another reason for the impact on ticket counts could be the hiring of a new customer service expert to handle the tickets, but this was also tempered with a small company restructuring that ended with a smaller amount of help desk personnel. There is also the possibility that long-time clients of the software were gaining experience in using the software, or that they had better internal guidelines on using it; This is difficult to quantify, however. And finally, we cannot discount the possibility of the

COVID-19 pandemic having its own effect on the number of tickets, as it would have a small effect on nearly everything.

But even with these other factors in play, the overall trend during and after the modernization seems to have been a lessened amount of tickets created, which can be inferred to mean that the usability of Case A's software has increased for the end-user. While promising, this inference should be confirmed from the customer through some manner of feedback cycle in the future, whether it is through surveys, interviews or other methods; This is outside of the scope of this thesis, however.

6. Discussion

The main research question of this thesis was about the impact of modernization on the developers of the product, the end-users of the product as well as the product itself, and what challenges such modernization efforts can face and how they can be mitigated with sufficient measures. While many of these questions did get answered within the framework of this thesis, there are still pitfalls and challenges related to modernization of software products that could not be covered completely. In this chapter, we will go through the parts of the thesis that are open to more discussion, such as how the thesis managed to cover its research material, what limitations it had, why the research was conducted as it was as well as what could possibly be improved in a future thesis on the subject.

6.1. Surveys and in-depth questionnaires

The decision to use in-depth questions and surveys was initially based on the fact that a modernization effort of a software system required a varied set of multidisciplinary skills, and because those skills were largely present within Case A itself, it was deemed to be useful to use that as a resource. This would also allow the thesis to get a more grass-roots experience and information on the subject matter, as well as opinions on the modernization itself straight from the source.

The questionnaire and survey process was completed in 2019, right in the middle of the modernization, which also accommodated the chance of accumulating pertinent data.

The gathered answers and data provided a decent look into how the other employees, developers and otherwise, saw the modernization, as all of them saw the modernization end-goal in a very positive light. The new methods and technologies had an impact on most of them as well, especially on the developers, who were the ones most directly influenced by the changes. Even though the experience level in some of the technologies was initially low among the employees, especially concerning front-end UI development tools like Material Design and Syncfusion, the overall results and effects remained impactful on each of the respondents' answers.

Unfortunately, due to the high attrition rate of the survey and in-depth questionnaire, with only 7 out of 24 answering the survey and 2 out of 4 answering the in-depth questions, the dataset for that specific part was not large enough to draw on concrete conclusions; Out of the respondents, 4 were developers, one was a product owner, one a project manager and one a customer relations manager. The answers provided

still allowed us to find out several interesting pieces of information, when looked at in conjunction with the material from the literature review.

This could be much expanded upon in a follow-up survey and in-depth questionnaire, especially if there would be a continuation on this thesis, as a 29.2% coverage with the surveys could use a great deal of improvement.

The questions in the survey and the in-depth questionnaire were a bit more broad than we would eventually have wanted them to be, and could have used more prototyping, with perhaps a round of demo surveys. Expanding more on the actual research questions rather than honing in on the technologies would have allowed for a more precise scope on the subject matter. Also, giving the customers and other end-users a survey on the usability of the modernized product would have given much more insights to the product's usability than simply following the data provided by customers through Jira. This would be a very useful follow-up course of action, as precise data from multiple customers on the product's usability after modernization could provide a great deal of insights into how the product can be developed in the future, and could be something that Case A could explore.

Overall, the usage of a survey and an in-depth questionnaire was a useful tool for the thesis, but had many problems in the end, mostly due to low participation rate. Pursuing respondents more aggressively could have provided more results and answers, perhaps with follow-up emails for those who did not answer, as it is possible that the initial message was lost or forgotten; In the case of senior developers or other senior personnel, it may be that they were too busy with their other duties to take the time to answer. For non-developers, it may be that the subject of the survey was too much outside of their field, and writing satisfactory answers and ratings may not have been comfortable for them.

As we could not get results from all of the employees within Case A, the results from the surveys and in-depth questionnaires are to be considered guidelines on prevailing trends within Case A rather than the whole picture.

While the respondents from the development side all had solid background and experience in back-end development, none of the developers came specifically from front-end and UI/UX backgrounds, which also reflects in the results of the surveys and the in-depth questionnaires.

6.2. Literature review

In this thesis we have performed a literature review over multiple different sources, with some of the main sources having been [Ber99] by Bergey et al., [Kal17] by Kalske, M., as well as several white papers by companies specializing in modernization such as Altexsoft [Alt19] and Capgemini [Cap19]. We also used several papers and product documentations on the technologies used, such as with Docker and Dotnet Core, as background material for explaining the impacts of the used technologies and new methodologies.

The usage of a literature review was decided because just having a survey and in-depth questionnaire would not have given a thorough enough look into the background of modernization, and how it can affect work in a company going through it. Reading through the references was very helpful in opening up the subject matter, and using it broadened the scope of the thesis.

6.3. Limitations

As the thesis was based on an ongoing software modernization in a company, we could have a good look at how modernization flows in practice, and the respondents to the in-depth questionnaire and survey all had ongoing experience with the modernization. This gave a more comprehensive base for the answers in the in-depth questionnaire and survey, as it was not based on a what-if scenario, but rather the ongoing modernization. Unfortunately, as mentioned, we only managed to reach only less than a third of the intended respondents, which decreased the accuracy of the conclusions that could be drawn from the remaining answers.

The data gathered about end-user usage statistics from Jira were used to indicate improved usability on the end-user side. This metric was decided upon due to it allowing us to see how the users could be using the software throughout the modernization, and how often they were raising tickets about issues within the software. While the figures and data show that the number of tickets created were steadily decreasing throughout the modernization, this is only a single data point correlation, and we listed several other reasons that could cause this downtick in created tickets. Whether or not this correlation is correct as we surmise in this thesis, or whether it is due to other factors that were discussed upon when going through the data, would require more research that is unfortunately out of scope for this thesis at the current time.

As said in the literature review part, we used a broad base of literature to try and accurately gauge the impacts of modernization, and how it can be properly handled. As software modernization is a constantly

evolving and progressing concept, it is necessary to consider the impact of even single pieces of technology in a whole modernization stack. The difficult part in having a very broad scope in terms of literature review, both in breadth and age, is that it can run the risk of losing focus on the important parts. Hopefully the focus of the thesis, software modernization, its challenges and how to pursue it, were clear to the reader. In the future, this is one area where we should keep the scope more concise, as while having a lot of sources and references is useful, it can easily become difficult to find the important information.

6.4. Case A ongoing modernization

During the software modernization outlined in this thesis Case A decided to implement a constant, iterative software modernization working in parallel to normal development; This would ensure that the software product stays up to date and utilizes the newest technologies as they emerge to maintain efficiency and usability. To facilitate this, a new development team was conceived within Case A in early 2020, which would focus mostly on achieving this goal. This team consisted of several senior developers, with other members of the development team rotating in and out of it to facilitate continuous learning for all of the developers working for Case A.

The ongoing modernization team was tasked with several parts of any future modernization efforts: Researching viable technologies, choosing the most appropriate one for the software product and then implementing them in both the back- and front-end of the software product: At the time of writing, the software modernization team has already implemented many of the proposed technologies for the purposes of comprehensive development demos they chose, and the team-rotation has almost finished for all of the developers in Case A.

The technologies adopted and moved to the demo-stage include RavenDB [Rav10] which is a NoSQL based document database, EventStoreDB [Eve12] for event sourcing and handling and Svelte UI [Sve16] for building front-end user interfaces; Case A is moving to implement more of the CQRS, command query responsibility segregation, pattern for all of its programming design using these new technologies.

In addition to the above technologies in use, the Jira Atlassian issue tracker was updated to Jira Software Cloud [Jir02] for a more decentralized solution, and Case A’s knowledge base wiki was migrated to use Atlassian’s Confluence wiki [Con04].

7. Conclusions

In this thesis we have explored the impact that modernization of technologies, methods and processes have had on a SaaS product using a real-life company as a case study. For this purpose, we used a survey and in-depth questionnaire with the employees of that company as well as background literature reviews to find out that impact in specific.

As RQ1, “How may software modernization affect a software product?”, is very broad in scope, all of the conclusions of this chapter on the other research questions also apply to answering this question.

From the material and research done on the subject, we can conclude that software modernization has had a significant impact on Case A’s product: The modernization of technology has opened doors for additional modernization in the future, as newer technologies are more modular and supported by more commonly used libraries and apps than legacy technologies.

As such, modernization has several benefits that have been discussed in the previous chapters of this thesis, but how do these benefits transfer to the end-user as improved usability, stability and performance?

Covering RQ2, “How may software modernization affect the usability of a software product for the end user?”, the adoption of new standards and techniques in UI design and standardization of such designs, such as with taking Google’s Material Design [Mat14], Syncfusion [Syn01] or Svelte [Sve16], all contribute to creating a design for applications that are easy to use and easy to understand. This translates to a more cohesive usage experience for the end-users. And even if the methods are not completely adopted for usage in the app, they provide guidelines on how to make the UI better. We can infer some of the effects from usability from chapter 5.4., where the overall number of tickets created by users has a steady downwards trend as the modernization progresses, and while this has some caveats as covered by the chapter, there does seem to be a positive effect to the usability from the customer’s point of view following the modernization. This conclusion is reinforced by [Ric15c], which emphasizes how microservices can be used to create less tightly coupled services, whose functionality is not dependent on the functionality of other services within the same system: The failure of one service does not cause the whole system to be incapacitated, such as can happen in monolithic systems with tight coupling. Improved logging and problem-tracking through the new tools, Microsoft Azure Application Insights, OpenApi and

Swagger, also contribute to improving the stability of the system, as they reduce the time it takes for the developers to locate issues in the system.

From a purely developers' point of view, the adoption of new tools has at the same time made updating the software simpler, as the monolith has been decomposed to smaller services that are easier to handle, and the developer workflow was improved through those same technologies. Usage of Azure for event logging and configuration of services has simplified both of these procedures, albeit the creation of custom Azure event logging is an ongoing process within Case A, so Seq [Seq13] logs are still in active use; It is likely that this will continue for a while, as iterative development of the custom logs takes extra development time. Containerization of services with Docker has allowed for looser coupling of services within the software system, and thus improved development by the developers not having to consider every single connection between the different services while developing and testing the differing services. All of these taken together answers RQ3, "How may software modernization affect the development of a software product from a developer's point of view?", on the developers' side.

As we covered in chapter 4 in depth, there are a myriad different challenges that a product undergoing software modernization can face, which help in answering RQ4 "What challenges can modernizing a legacy system that is still in production use pose?": Be they challenges from a software perspective to challenges coming from the environment, such as management, personnel or documentational challenges. The challenges to software modernization documented in this thesis alone can make the task into a risky endeavor, yet there is a high possibility of encountering unique challenges depending entirely on the context of the software modernization task: Updating the platform of a multiplayer video game is much different from updating the core software of critical medical devices, for example. Some of the challenges relate to challenges included in any modernization endeavor, software or otherwise, such as the aforementioned management and personnel challenges, and have existed for as long as any kind of modernization concepts have existed, while others are uniquely scoped to software development as it is now: People are generally change-resistant, and may pose challenges regardless of what year it is and where it happens, while challenges relating to moving a software from local servers to a cloud-based platform as microservices is uniquely challenging for software in use today. Software in production use is no exception, for while there are general challenges from software modernization that applies to it, it has its own unique challenges, such as having to be constantly available to the users using it, and not breaking down for these users while the software is being modernized.

In chapters 3.3 we covered several modernization strategies, methods and techniques as well as microservice migration patterns, and in covering modernization challenges in chapter 4 we arrived at several different solutions to RQ5 “How can the challenges of modernizing a legacy system that is still in production use be mitigated?”; The main solution to mitigating challenges to a modernization effort is a solid, correctly formulated modernization strategy that is communicated throughout the whole company going through it [BHJ15], as this strategy will affect all parts of modernization. The context of the modernization dictates what methods and techniques are optimal for each modernization, and choosing the correct technique for the context from encapsulation, rehosting, replatforming, refactoring, rearchitecting, rebuilding or replacing is a key step in any modernization strategy. For this reason, it is important to gather as much information about the original system as possible before modernization, as well as gathering and documenting what the requirements of the post-modernization system are; Documentation takes an important role here, for both pre- and post-modernization purposes. Maintaining training for all employees on the modernization and all adopted new technologies and methods is important for mitigating issues originating directly from personnel [Ber99]; A maintained and comprehensive knowledge base on the modernization also helps personnel to keep up-to-date on it, and can be used to facilitate information comprehensively throughout the workers of the company.

For systems that are in production use, one of the most important challenges to overcome is accessibility: Maintaining a balance between development and maintenance of the SaaS -product while at the same time working on the modernization effort. For this purpose, as earlier explained, a clear and planned-out modernization strategy is extremely important, as putting extra effort in at the planning stage will help mitigate challenges down the line. Case A planned this issue using a specific modernization team that worked alongside the other development teams, which concentrated only on the issue of modernization; This modernization team had several consultants working on it, while the development teams continued working normally on the product’s development and maintenance: This approach worked out relatively well for Case A, as the SaaS -product kept improving simultaneously while the modernization progressed. Most of the other solutions to challenges from and during modernization were introduced in chapter 4, so other than the key-solutions above, we shall refrain from repeating those solutions here; What bears repeating, however, is that all modernization situations are different, and while some of the solutions here may work in specific cases, there are cases where they may be wrong: There is no silver bullet for modernization challenges.

The results for Case A from modernization have so far been encouraging. While there have been challenges with training personnel to use the new technologies and methods, the impact on customer-

generated tickets about the software as well as the effect on the product's development cycle have been positive enough to offset these initial challenges. The adoption of the microservices pattern has allowed the development teams to work in a less tightly coupled fashion, enabling better parallel development, and the new tools and continuous, iterative modernization have made the product more viable for future development.

There still remain challenges to overcome in continuing forward with modernization, chief among them the allocation of resources to maintain and develop the base product, while simultaneously researching and developing new technologies to keep the product competitive with the other products on the market.

From Case A, we have seen that modernization generally benefits software products, as long as there is a well-thought out modernization strategy to follow when implementing it: While there are many challenges set out for modernization, they can be avoided or mitigated with proper planning and research. We can also see from Case A that large batch modernizations can take a long time and place an undue effort on the party implementing it, so having an iterative modernization development that works in parallel to normal development can help mitigate this impact: This type of modernization can help a product stay competitive for a long time into the future.

Bibliography

AA13 Apostu, A., Puican, F., Ularu, G., Suciu, G., Todoran, G., Study on advantages and disadvantages of Cloud Computing - the advantages of Telemetry Applications in the Cloud, 2013. https://www.semanticscholar.org/paper/Study-on-advantages-and-disadvantages-of-Cloud-%E2%80%93-of-Apostu-Puican/da6249af5e0df8ed36334b34c2f1010ed19c53f4 [20.02.2022]

Aka20 Akana by Perforce, Microservices: What, Why, and How, 06.04.2020. https://www.akana.com/blog/microservices-what-why-and-how [05.12.2021]

Alt19 Altexsoft, Legacy System Modernization: How to Transform the Enterprise for Digital Future white paper, 2019. https://www.altexsoft.com/whitepapers/legacy-system-modernization-how-to-transform-the-enterprise-for-digital-future/ [26.11.2019]

ANC16 Microsoft, Asp.net Core, 27.06.2016. https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-3.1 [10.03.2020]

Azd21 Microsoft, Microsoft Azure Documentation, 2021. https://docs.microsoft.com/en-us/azure/?product=featured [21.05.2021]

Azu10 Microsoft, Microsoft Azure, 01.02.2010. https://azure.microsoft.com/en-us/ [28.11.2019]

Ber99 Bergey, J; Smith, D; Tilley, S; Weiderman, N; Woods, S., Why Reengineering Projects Fail, 1999. https://apps.dtic.mil/docs/citations/ADA362725 [26.11.2019]

BHJ15 Balalaie, A., Heydarnoori, A. and Jamshidi, P., Microservices Migration Patterns, Technical Report No. 1, TR-SUT-CE-ASE-2015-01, Automated Software Engineering Group, Sharif University of Technology, 10.2015. http://ase.ce.sharif.edu/pubs/techreports/TR-SUT-CE-ASE-2015-01-Microservices.pdf [11.06.2021]

BHJ16 Balalaie, A., Heydarnoori, A. and Jamshidi, P., Microservices architecture enables devops: migration to a cloud-native architecture, 05.2016. IEEE Software, 33,3(2016), pages 42–52. [28.05.2021]

Cap19	Jiang, Z., Tolido, R., Jones, S., Hunt, G., Budor, I., Bartoli, E., Van der Linden, P., Buvat, J., Theisler, J., Wortmann, A., Cherian, S., Khemka, Y., Championing Data Protection and Privacy, 09.03.2019. https://www.capgemini.com/gb-en/wp-content/uploads/sites/3/2019/09/Report-%E2%80%93-GDPR.pdf [28.11.2021]
CaV20	Catrinescu, V., Microsoft Azure Cloud Concepts, 17.07.2020 https://app.pluralsight.com/library/courses/microsoft-azure-cloud-concepts/table-of-contents [19.05.2021]
Con04	Atlassian, Confluence wiki, 25.03.2004 https://www.atlassian.com/software/confluence/use-cases/wiki [16.10.2022]
Dap20	Dapper, Dapper ORM 05.04.2020. https://github.com/DapperLib/Dapper [05.12.2021]
DNC16	Microsoft, Dotnet Core, 27.06.2016. https://dotnet.microsoft.com/ [10.03.2020]
DNC20	TutorialsTeacher, .NET Core Overview, 2020. https://www.tutorialsteacher.com/core/dotnet-core [20.08.2021]
Doc17	Docker Inc., Docker, 2013. https://www.docker.com/ [28.11.2019]
EFC20	Microsoft, Entity Framework Core documentation, 20.09.2020. https://docs.microsoft.com/en-us/ef/core/ [12.05.2021]
Eve12	Event Store, EventStoreDB, 2012. https://www.eventstore.com/eventstoredb [13.07.2022]
FB19	Fritzsch, J; Bogner, J; Zimmermann, A; Wagner, S., J.-M. Bruel et al. (Eds.): DEVOPS 2018, LNCS 11350, pp. 128–141, From Monolith to Microservices: A Classification of Refactoring Approaches, 2019. https://www.researchgate.net/publication/326646378_From_Monolith_to_Microservices_A_Classification_of_Refactoring_Approaches [06.08.2020]
Fo15	Fowler, M., Microservice Trade-Offs, 01.07.2015. https://martinfowler.com/articles/microservice-trade-offs.html [06.02.2022]
Gar06	Gardner, D., Not just a nip and tuck, application modernization extends the lifecycle of legacy code assets, 06.10.2006. https://www.zdnet.com/article/not-just-a-nip-and-tuck-application-modernization-extends-the-lifecycle-of-legacy-code-assets/ [10.12.2019]

GiA20 Gillis, S. A., RESTful API definition, 09.2020. https://searchapparchitecture.techtarget.com/definition/RESTful-API [28.05.2021]

HaA97 Harmsen, A.F., Situational Method Engineering, 31.01.1997. ISBN: 90-75498-10-1. [11.06.2021]

Jen11 Kawaguchi, K., Jenkins, 02.02.2011. https://www.jenkins.io/ [22.11.2021]

Jen21 Jena, S., Architecture of Cloud Computing, 25.03.2021. https://www.geeksforgeeks.org/architecture-of-cloud-computing/ [06.02.2022]

Jir02 Atlassian, Jira Software Cloud, 2002. https://www.atlassian.com/software/jira/whats-new/cloud [13.07.2022]

Kal17 Kalske, M., Transforming monolithic architecture towards microservice architecture, 19.11.2017. https://helda.helsinki.fi/handle/10138/234239 [26.11.2019]

KeM21 Biederman, E.W.; Kerrisk, M., Linux Programmer's Manual, 22.03.2021. https://man7.org/linux/man-pages/man7/namespaces.7.html [29.04.2021]

KIS12 Kniberg, H.; Ivarsson, A., Scaling Agile @ Spotify with Tribes, Squads, Chapters & Guilds, 10.2012. https://blog.crisp.se/wp-content/uploads/2012/11/SpotifyScaling.pdf [28.05.2021]

LeJ20 Lerman, J., Entity Framework Core: Getting Started, 15.12.2020. https://app.pluralsight.com/library/courses/entity-framework-core-get-started [12.05.2021]

LF14 Lewis, J; Fowler, M., Microservices a definition of this new architectural term, 25.03.2014. https://martinfowler.com/articles/microservices.html [05.12.2019]

Mal10 Malinova, A., Approaches and techniques for legacy software modernization, 01.2019. https://www.researchgate.net/publication/267181092_Approaches_and_techniques_for_legacy_software_modernization [26.11.2019]

Mat14 Google, Material Design, 25.05.2014. https://material.io/ [28.11.2019]

MHR20 Monson-Haefel, R., Getting Started with Swagger Tools, 08.17.2020. https://app.pluralsight.com/library/courses/getting-started-swagger-tools/table-of-contents [27.05.2021]

MW21 MiddleWare Team, What Are Microservices? How Microservices Architecture Works, 23.09.2021. https://middleware.io/blog/microservices-architecture/ [06.02.2022]

NaA17 Nagpal, A., 7 Reasons Why Continuous Learning is Important, 30.06.2017. https://www.linkedin.com/pulse/7-reasons-why-continuous-learning-important-amit-nagpal [05.08.2021]

OAS17 Linux Foundation, OpenAPI specification, 26.07.2017. https://www.openapis.org/ [10.03.2020]

OS19 Opensource.com, What are microservices?, 2019. https://opensource.com/resources/what-are-microservices [19.05.2020]

PoN18 Poulton, N., Pluralsight web course: Docker Deep Dive, 01.04.2018. https://app.pluralsight.com/library/courses/docker-deep-dive-update/table-of-contents [09.04.2019]

PoN19 Poulton, N., Pluralsight web course: Docker and Kubernetes: The Big Picture, 02.13.2019. https://app.pluralsight.com/library/courses/docker-kubernetes-big-picture/table-of-contents [09.04.2019]

PoN20 Poulton, N., Pluralsight web course: Getting Started with Docker, 27.10.2020. https://app.pluralsight.com/library/courses/getting-started-docker/table-of-contents [09.04.2019]

PV19 Peshraw, A., A., Varol, A., Advantages to Disadvantages of Cloud Computing for Small-Sized Business, 2019. https://ieeexplore.ieee.org/document/8757549 [20.02.2022]

Rav10 Hibernating Rhinos, RavenDB, 2010. https://ravendb.net/ [10.07.2022]

Ric15 Richardson, C., Monolithic architecture, 2015. http://microservices.io/patterns/monolithic.html [24.08.2020]

Ric15a Richardson, C., Microservices Database per service, 2015. https://microservices.io/patterns/data/database-per-service.html [06.08.2021]

Ric15b Richardson, C., Microservices Shared database, 2015. https://microservices.io/patterns/data/shared-database.html [06.08.2021]

Ric15c Richardson, C., Microservices decompose by business capability, 2015. https://microservices.io/patterns/decomposition/decompose-by-business-capability.html [12.12.2021]

Sco04	Elsevier, Scopus, 2004. https://www.scopus.com/home.uri [06.03.2022]
Seq13	datalust, Seq, 2013. https://datalust.co/seq [15.05.2022]
SmB16	SmartBear Software, Swagger/OpenAPI specification, 2016. https://swagger.io/ [27.05.2021]
SPL03	Seacord, R., D. Plakosh, G. Lewis, Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices, Addison-Wesley, 2003. [05.09.2021]
SrA08	Srivastava, A., Introducing Windows Azure, 27.10.2008 https://archive.is/M0vvP [21.05.2021]
SS21	StepSize, State of Technical Debt 2021, 09.2021. https://www.stepsize.com/report [06.02.2022]
Sve16	Harris, R., Svelte, 29.11.2016. https://svelte.dev/ [21.12.2021]
SwG15	Linux_Foundation, New Collaborative Project to Extend Swagger Specification for Building Connected Applications and Services, 05.11.2015. https://web.archive.org/web/20160427104213/http://www.linuxfoundation.org/news-media/announcements/2015/11/new-collaborative-project-extend-swagger-specification-building [27.05.2021]
Syn01	Syncfusion Inc., Syncfusion, 06.2001. https://www.syncfusion.com/ [28.11.2019]
TH99	Thomas, David; Hunt, Andrew. Pragmatic Programmer, 10.1999. [09.06.2020]
ToC21	Torre, C., Modernize existing .NET applications with Azure cloud and Windows Containers, 07.01.2021. https://docs.microsoft.com/en-us/dotnet/architecture/modernize-with-azure-containers/ [28.05.2021]

Glossary

CD	Continuous Delivery: “Continuous delivery is a software development practice where code changes are automatically prepared for a release to production. A pillar of modern application development, continuous delivery expands upon continuous integration by deploying all code changes to a testing environment and/or a production environment after the build stage. When properly implemented, developers will always have a deployment-ready build artifact that has passed through a standardized test process.” Amazon AWS, https://aws.amazon.com/devops/continuous-delivery/ [24.09.2021].
CI	Continuous Integration: “Continuous integration is a DevOps software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. Continuous integration most often refers to the build or integration stage of the software release process and entails both an automation component (e.g. a CI or build service) and a cultural component (e.g. learning to integrate frequently). The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates.” Amazon AWS, https://aws.amazon.com/devops/continuous-integration/ [24.09.2021].
CL	Continuous Learning: The process of continuously and consistently learning new skills and knowledge. Can relate to casual learning, tutoring, attending classes etc. This can happen personally as lifelong learning, or within an organization as a result of a learning strategy set up by the organization. Also known as lifelong learning [NaA17].
ERP	Enterprise Resource Planning: “Enterprise resource planning (ERP) refers to a type of software that organizations use to manage day-to-day business activities such as accounting, procurement, project management, risk management and compliance, and supply chain operations. A complete ERP suite also includes enterprise performance management, software that helps plan, budget, predict, and report on an organization’s financial results.”. Oracle, https://www.oracle.com/applications/erp/what-is-erp.html [07.04.2020].
IFRS	International Financial Reporting Standards: “International Financial Reporting

	Standards, commonly called IFRS, are accounting standards issued by the IFRS Foundation and the International Accounting Standards Board (IASB). They constitute a standardised way of describing the company's financial performance so that company financial statements are understandable and comparable across international boundaries. They are particularly relevant for companies with shares or securities listed on a public stock exchange.". Wikipedia, https://en.wikipedia.org/wiki/International_Financial_Reporting_Standards [07.04.2020].
PaaS	Platform as a service: "Platform as a service (PaaS) or application platform as a service (aPaaS) or platform-based service is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app.". Wikipedia, https://en.wikipedia.org/wiki/Platform_as_a_service. [07.04.2020]
O/RM	Object-relational mapper: Reduces friction on how data is structured in a relational database and how it is defined in classes. Without an ORM, you have to write a lot of code to transform database results to transform them to be usable by our code. ORMs allow us to express our database queries using classes, building and executing the query we want. Also, ORMs allow us to store, update and delete data from the database [LeJ20, EFC20].
REST	Representational State Transfer: This is an architectural style often used in the context of RESTful APIs as a part of a web service or communications over the web. A RESTful API uses HTTP requests to GET, UPDATE, INSERT and DELETE data types, which in turn do the above operations on data resources [GiA20].
SaaS	Software as a service: "Software as a service (SaaS /sæs/) is a software licensing and delivery model in which software is licensed on a subscription basis and is centrally hosted. It is sometimes referred to as "on-demand software", and was formerly referred to as "software plus services" by Microsoft.". Wikipedia, https://en.wikipedia.org/wiki/Software_as_a_service [07.04.2020].
SOA	Service-oriented architecture: "A style of software design where services are provided to the other components by application components, through a communication protocol over a network.". Wikipedia, https://en.wikipedia.org/wiki/Service-oriented_architecture [03.12.2019].

Appendix A

Hi!

I am currently doing my master's thesis on the subject of software modernization within Fatman (and in general), and would greatly appreciate it if you had the time to answer a quick survey on the subject.

It shouldn't take more than 5-10 minutes of your time to fill out, and it would really help me out.

There are also a few more open-ended interview questions after the survey, and I hope you can find the time to answer those as well.

With best regards,

Janne Kauhanen

Company Survey

- What is your role in the company?
    - Developer
    - Tester
    - Quality Assurance
    - System Administrator
    - Project Owner
    - Project Manager
    - Other (what?)
- On a scale of 0 to 10, with 0 being "No effect" and 10 being "Large effect", how did the change to use the new tools and frameworks affect your job?
    - Azure
    - Angular
    - Docker
    - Syncfusion
    - Material Design
    - OpenAPI / Swagger

    - Dotnet Core
- On a scale of 0 to 10, with 0 being “No experience” and 10 being “Very experienced”, rate your level of experience with the following tools before the change:
    - Azure
    - Angular
    - Docker
    - Syncfusion
    - Material design
    - OpenAPI / Swagger
    - Dotnet Core
- What are your opinions on using the new tools after the change happened?
    - On a scale of 0 to 10, with 0 being “Very negative” and 10 being “Very positive”.
- How did the restructuring of teams within the company affect your work?
    - On a scale of 0 to 10, with 0 being “Very negatively” and 10 being “Very positively”.
- Is there anything else you would like to say on the topic?

Interview

- In general, what are your opinions on using the new tools after the software modernization finished?
- Why was modernization important for the company?
- If another modernization change should take place in the future, would you like to change anything in the implementation or planning of such a change?
- Is there anything else you would like to say on the topic?

Appendix B

In-depth questionnaire questions and answers:

In-depth questionnaire, Software Developer 1:

Q: In general, what are your opinions on using the new tools after the software modernization finished?

A: New Tools and technologies are improving and emerging faster than ever before. So it is important to use the new technologies, the benefits are three fold; the tools and technologies are helpful in terms of faster development time and improved services, second the employees are able to keep up with new technologies and are always learning new things and tools and organization is able to switch to newer tools faster according to the industry standards, third it is important for the employee career.

Q: Why was modernization important for the company?

A: The company has been delivering SaaS to the customer for a long time and has a lot of legacy applications. Thus, it was important to transition to newer tools and technologies to build faster (both in terms of development time and software) and improved software.

Q: If another modernization change should take place in the future, would you like to change anything in the implementation or planning of such a change?

A: Modernization is an ongoing process as the technologies and tools are keep (sic) on evolving for better, faster and faster. So, it is important to keep modernizing. The implementation could be improved by planning in advance while keeping in mind the required resources and time for change.

In-depth questionnaire, Software Developer 2:

Q: In general, what are your opinions on using the new tools after the software modernization finished?

A: New tools are generally a good idea. Even if it might cause some issues and even if its decided that the new tool is dropped. Using new tools and technologies brings new perspectives and potentially even fixes problems you didn’t know you had. Generally it’s a good idea for a developer to be actively on a lookout for new tools.

Q: Why was modernization important for the company?

A: To remain competitive and increasingly profitable companies seek venues of improvement. Part of companies (sic) continuous improvement, they attempt to more efficiently use available resources. Modernization is a form of such improvement.

Q: If another modernization change should take place in the future, would you like to change anything in the implementation or planning of such a change?

A: Considering the given parameters, best decisions were made. In future It would be good to do smaller improvements more frequently. Rather then (sic) a big change in a relatively short time period.

Appendix C

	Person 1	Person 2	Person 3	Person 4	Person 5	Person 6	Person 7		All	Dev Only
Role	Developer	Developer	Developer	Developer	Product Owner	Project Manager	Customer Relations Manager			
Effect on work										
Azure	6	8	6	3	7	8	8	Azure Avg:	6.571	5.75
Angular	5	6	7	3	3	0	8	Angular Avg:	4.571	5.25
Docker	7	3	8	4	3	0	8	Docker Avg:	4.714	5.5
Syncfusion	4	0	3	3	7	0	8	SF Avg:	3.571	2.5
Material Design	7	0	3	2	7	0	0	MD Avg:	2.714	3
OpenAPI / Swagger	8	0	6	5	7	10	10	OAPI/Swag Avg:	6.571	4.75
Dotnet Core	9	3	8	3	3	0	0	Dotnet Avg:	3.714	5.75
Prev. Experience										
Azure	5	0	3	0	1	1	0	Azure Avg:	1.429	2
Angular	4	5	2	3	0	0	0	Angular Avg:	2	3.5
Docker	6	1	4	3	1	0	0	Docker Avg:	2.143	3.5
Syncfusion	3	0	0	0	1	0	0	SF Avg:	0.571	0.75
Material Design	6	0	0	0	0	0	0	MD Avg:	0.857	1.5
OpenAPI / Swagger	7	8	5	5	1	5	0	OAPI/Swag Avg:	4.429	6.25
Dotnet Core	8	1	7	3	0	0	0	Dotnet Avg:	2.714	4.75
New tool opinion	8	9	8	10	8	7	9			
Restructure effect	9	9	6	7	8	8	8			

Avg. Effect	4.633	Avg. Effect (Dev,Backend)	5.4
Avg. Effect (Dev)	4.31	Avg. Effect (Dev,Frontend)	2.75
Prev. Exp.	3.786	Prev. Exp. (Dev,Backend)	4
Prev. Exp. (Dev)	3.333	Prev. Exp. (Dev,Frontend)	1.125