LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT
School of Engineering Science

Software Engineering

Master's Programme in Software Engineering and Digital Transformation

MODERNIZATION OF LEGACY SYSTEMS AND MIGRATIONS TO MICROSERVICE ARCHITECTURE

Kristian Tuusjärvi

Master Thesis

Examiners: Associate professor Jussi Kasurinen

Assistant professor Antti Knutas

TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT

School of Engineering Science

Tietotekniikan koulutusohjelma

Ohjelmistosuunnittelun ja Digitaalisen Transformaation Maisteriohjelma

Kristian Tuusjärvi

MODERNIZATION OF LEGACY SYSTEMS AND MIGRATIONS TO MICROSERVICE ARCHITECTURE

Diplomityö

2021

50 sivua, 12 kuvaa, 1 taulukko, 1 liite

Työn tarkastajat: Apulaisprofessori Jussi Kasurinen

Apulaisprofessori Antti Knutas

Hakusanat: Legacy Systeemit, Mikropalveluarkkitehtuuri, Monoliittinen Arkkitehtuuri, Mikropalvelun Siirtäminen

Nykyinen siirtyminen kohti digitalisaatiota pakottaa yritykset modernisoimaan vanhentuvia järjestelmiään. Tämä diplomityö antaa kattavan kuvauksen vanhoista järjestelmistä. Lisäksi diplomityössä tehtiin systemaattinen kirjallisuuskatsaus, joka tutki vanhojen systeemien modernisointia mikropalveluarkkitehtuuriin. Tutkimuksessa tarkasteltiin tutkimuksia, jotka liittyvät vanhan systeemin modernisoimiseen mikropalveluarkkitehtuuriin. Tutkimuksia tutkittiin lähestymistavan, tutkimusalueen ja tutkimuskontribuution näkökulmasta. Yksi tutkimuksen päähavainnoista on se, että mikropalveluihin siirtymistä käsittelevistä empiirisistä tutkimuksista on puutetta.

ABSTRACT

Lappeenranta-Lahti University of Technology

School of Engineering Science

Software Engineering

Master's Programme in Software Engineering and Digital Transformation

Kristian Tuusjärvi

MODERNIZATION OF LEGACY SYSTEMS AND MIGRATIONS TO MICROSERVICE ARCHITECTURE

Master’s Thesis

2021

50 pages, 12 figures, 1 table, 1 appendix

Examiners: Associate professor Jussi Kasurinen

Assistant professor Antti Knutas

Keywords: Legacy Systems, Microservice Architecture, Monolithic Architecture, Microservice Migration

The current shift toward digitalization is pushing companies to modernize their legacy systems. This thesis gives a comprehensive summary of legacy systems. Furthermore, a systematic mapping study was performed to study a relatively new legacy system modernization trend – the trend of migrating legacy software systems to microservice architecture. The study examines the existing research literature according to approach, research area and contribution. One of the main findings from the study is that there is a lack of empirical studies attempting to explain the microservice migration.

ACKNOWLEDGEMENTS

This thesis has taken nearly eight months to complete. It has been a great learning experience with many challenges. Firstly, I would like to thank Associate Prof. Jussi Kasurinen for his guidance throughout the writing process. Associate Prof. Kasurinen was especially supportive toward my working situation and timeline. The subject of this thesis is an area of interest for me, and I hope to continue working with it in the future.

Secondly, I would thank my friends and family for their continuous support – for without them I could not have accomplished this thesis. Lastly, a special thanks to Henni Hyvärinen and Kaarina Tuusjärvi for proof reading and correcting my grammar.

Table of contents

1. Introduction

Companies spend a lot of money developing software systems for their needs. These systems are often the backbone of their business models. Understandably, companies want to maximize the return on their investment and will try to keep their software systems running for as long as possible. Software systems may be used for as long as 10 years or even longer. Naturally, there will be maintenance to these software systems, which is required to satisfy business and customer needs, and changes in the underlying hardware, as well as to fix bugs (Sommerville, 2016). Lehman's first law states, software needs to be updated continually or it will become obsolete (Lehman, et al., 1998).

Maintaining a software system works well for a while, but as the time passes, the system becomes increasingly complex and out of date, and eventually maintenance will no longer be efficient or cost effective. Business needs dictate that the system needs to be modernized to improve the system's maintainability, performance, and business value. Modernization takes much more effort to accomplish compared to maintenance. In the end, when a software system can no longer be updated, it will need to be replaced (Comella-Dorda, et al., 2000).

This study attempts to give a comprehensive summary of legacy software, what it is and how to manage it. Additionally, a systematic mapping study (SMS) was performed to study a relatively new legacy software modernization trend – the trend of migrating monolithic legacy software systems towards microservice architecture. Microservice architecture (MSA) increases maintainability, scalability and decreases coupling (Lithicum, 2016). MSA has become popular due to new technological advancements, such as cloud computing and dockerization. Furthermore, the overall digitalization of the business world has forced companies to search for more dynamic software architecture.

2. Research method

This thesis conducts research in the form of SMS. SMS is a research methodology that summarizes a pool of research papers and results by gategorizing them. The results are visualized to create the map of the researched subject (Kitchenham, et al., 2007). SMS is recommended as the research methodology in software engineering when the research area is still emerging, and there isn't yet substantial quantity of high quality studies (Kitchenham, et al., 2007). This is the case with the literature of migrating legacy software to MSA. Systematic mapping studies are more commonly used in medical research and are less popular on the field of software engineering (Bailey, et al., 2007).

2.1 Research process

The SM study approach that is used in this thesis is adapted from the approach used by Petersen, et al. (2008) in their paper. The main phases of the SMS study are definition of research questions, conducting the search, screening of papers, keywording and data extraction and mapping, as can be seen in Figure 1. All the phases have an output that is forwarded to the next phase and the last output is the systematic map.

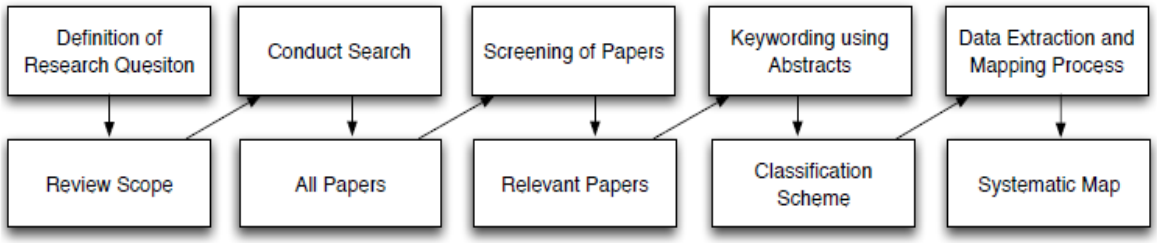

Figure 1. Systematic mapping study process (Petersen, et al., 2008)

2.2 Research questions and scope

The goal of SMS is to provide a general view of the research subject (Petersen, et al., 2008). In this thesis the research subject is migration of monolithic systems to MSA. The research questions and their sub questions reflect the goal of this SMS. Research questions are listed below.

RQ1: What are the research approaches?

RQ2: What areas of migrating to MSA are addressed?

RQ3: What are the contributions to modernization from monolithic legacy systems to MSA?

2.3 Search strategy

Google Scholar's research database was used to get research material for the SMS. It was chosen because it looks for research material from many different publishers, such as ACM and IEEE. The search was conducted using a search term, which was developed by testing keywords against the database. The goal of testing different search terms was to find a keyword combo that would yield a high number of search results (300 – 500 results) and would reflect the research area. The search results were evaluated manually to estimate if they matched the research area. The evaluation was done by reading through the search result summaries and seeing if they fitted the research area. As suggested by Petersen, et al., (2008) the search terms do reflect the research questions. The final search combo consisted of the following words: microservices, legacy software, migration and modernization.

2.4 Inclusion and exclusion

The less relevant research papers were filtered out from the results. This was done by using inclusion and exclusion criteria. The research criteria for inclusion were the following: research papers, conference papers and journal articles from 2015 to 2020, with more than 4 pages of text, English as the primary language, public access and no master's or bachelor's thesis. Additionally, the research paper must explicitly discuss the theme of this SMS (i.e., migration of legacy software systems to microservices architecture) in the topic, abstract or introduction of the paper. The number of citations was not considered as it would have given a less realistic view of the research area.

2.5 Classification schemas

The classification scheme is developed by using a systematic process introduced by Petersen, et al., (2008). The process utilizes keywording to craft the classifications. There is two steps in the keywording process. First, abstracts are read through to look for keywords and trends that reflect the contribution of the research paper. While doing this it is also possible to identify the context of the research. Once this is done, the sets of keywords are unified to create a higher level representation of the research. From the high level representation it is easier to create classification scheme that represent the underlying

populations. If the abstract is not informative enough the introduction and conclusion would be read. Finally, a set of keywords is chosen and combined to create the categories for the systematic map (Petersen, et al., 2008).

The keywording process for this study yielded three main categories. These categories are research contribution, research type and the research area. The research contribution is derived from the keywords. The research contribution can be a tool, data, information, processes and so on. The research area schema describes the area in focus in relation to the topic. For example, decomposition of a legacy system, challenges, and benefits of the migration (Petersen, et al., 2008). The research type was categorized by using Wieringa's (2006) categorization method, which is illustrated in Table 1. This categorization method is general and does not depend on any specific research field (Petersen, et al., 2008).

Table 1. Classification for research types by (Wieringa, et al., 2006).

Category	**Description**
Validation Research	Techniques investigated are novel and have not yet been implemented in practice. Techniques used are for example experiments, i.e., work done in the lab.
Evaluation Research	Techniques are implemented in practice and an evaluation of the technique is conducted. That means, it is shown how the technique is implemented in practice (solution implementation) and what are the consequences of the implementation in terms of benefits and drawbacks (implementation evaluation). This also includes to identify problems in industry.
Solution Proposal	A solution for a problem is proposed, the solution can be either novel or significant extension of an existing technique. The potential benefits and the applicability of the solution is shown by a small example or a good line of argumentation.

Philosophical Papers	These papers sketch a new way of looking at existing things by structuring the field in form of a taxonomy or conceptual framework.
Opinion Papers	These papers express the opinion of somebody whether a certain technique is good or bad, or how things should be done. They do not rely on related work and research methodologies.
Experience Papers	Experience papers explain on what and how something has been done in practice. It must be the personal experience of the author.

2.6 Data extraction and systematic map

After finishing the classification scheme, data from research papers is extracted and the papers are sorted into the scheme. The scheme keeps evolving through the data extraction process, which means that the categories can be merged or added depending on the data (Petersen, et al., 2008). The data extraction process works in three steps. First, a research paper is read through and key information is stored into Excel sheet. Second, the information is analyzed, and the appropriate classification is added to the paper's information. Finally, the classification information is transformed into graphs to visualize the 'map' of the research topic.

3. Review of legacy software

In this chapter the notion of legacy software is explored. The aim is to explain how legacy software is born, what is the life cycle of a software system and how to modernize legacy systems. Additionally, a review of legacy software related literature is conducted.

3.1 Software stages

The life cycle of a software can be divided into five stages: (1) initial development, (2) evolution, servicing and (3) phasing out and closedown. During the initial development, the first versions of the software are developed. After the initial development phase, the evolution of the software product takes place. In this phase, the abilities and functionalities of the software product are shaped to meet customer needs. This phase may include major upgrades to the software. When the product has taken shape and only requires minor changes and tweaks, it enters the servicing phase of its life cycle. In servicing phase, no major upgrades are applied, only minor changes and bug fixes. The two final phases of the product's life cycle are phaseout and closedown. In phaseout stage, the software product is no longer cost effective and so the software will enter closedown phase, where it will be replaced or removed completely from the market. If replaced, the customers are directed towards the replacing software product (Vaclac, et al., 2000).

3.2 Software aging

In theory, software is composed of logical operations based on math, and math does not change, so software should not age. However, the opposite is true, and software does in fact age. This is caused by the changes in the world around the software. The world is constantly shifting and changing, which demands the software to change with it or be obsolete. To be precise, it is not the physical software that is aging. It is the purpose of the software that changes, and to account for that the software needs to change too (Parnas, 1994).

3.2.1 Causes

There are two main causes for software aging. Firstly, developers fail to implement changes to the software. If the software is not frequently updated to match the needs of the customers, the customers will become dissatisfied and start looking for alternative solutions. In theory, a software from 1950s could be used today, but no customer would be

satisfied by its performance or features. Secondly, the changes are successfully implemented to the software, but these changes are done by developers who do not understand the original design of the software. This will cause the structure of the software to degrade. If the changes are inconsistent with the original design, the resulting software will have two possibly colliding designs implemented on it. The next developer trying to make changes must now understand the original design and the design of the changes. This addition of complexity to a software can go on for the whole life cycle of the software, accumulating complexity and decreasing maintainability. The problem is aggravated by lack of documentation or incorrect documentation (Parnas, 1994).

3.3 Legacy systems

In the 1960s, digitalization of companies started and has continued to this date. Old software systems were made using the languages, hardware and supporting software available at the time, and the support for those is long gone. Most of these systems had to evolve because of the changes in business models and user requirements, as well as in technological advancements (Liu, et al.,1998). These old software systems are business critical and long lived. They are often referred to as legacy software systems (Dayini-Fard, et al., 1999).

The dictionary defines legacy systems as “systems that are in one way or another outdated, but still in use” (YourDictionary, 2020). There are many signs that can tell a system is a legacy system: there is no replacement hardware, the documentation is non-existing or invalid, the source code has gone missing, change history has not been managed and the original developers are no longer working in the company (Pressman, 2015).

3.3.1 Legacy system components

A legacy software system can be divided into system hardware, support software, application software, application data, business processes and business rules. Replacing one piece of this complex system will affect all the other parts. The hardware that the legacy system was developed with are outdated and expensive. Special skills are required to maintain the hardware and it is not compatible with other hardware used in the system. The support software such as OS, drivers used by hardware and development environments used for developing the system are out of date and contain security vulnerabilities. The application software is the main application that provides the business services to clients. It

is often made up of many applications that have been created in different times by many different teams. The application data refers to the data that the application uses to function.

Old legacy systems often have huge amounts of accumulated data. The data can be partly corrupted or inconsistent, because of the changes made to the application during years of maintenance. The business processes are used by businesses to accomplish a certain business goal. Business processes are often built around legacy systems that can enable and constrain their performance. The business rules define how a company does some part of their business (Sommerville, 2016). For example, bank can embed rules for giving out loans into their application. Sometimes companies lose their documentation of the business rules, and the only place they exist in is the application.

3.3.2 Existing legacy code

It is very difficult to estimate how much legacy software and systems are still in use. Some indication can be drawn from the estimations that in year 2000 there was still 200 billion lines of COBOL in use (Kizior, et al., 2000). According to the survey done by Computerworld (2012), 60% of the companies surveyed used COBOL more than they used C++ and Visual basic. COBOL is an old software language used from 1960s to 1990s, especially in the financial industry. There are still many software systems built with COBOL and if they keep working there is no need to replace them. However, there is an increasing risk since the support for old systems is running out. COBOL is no more a mainstream language and there are no new developers that would learn it. In the next decades old COBOL systems need to be replaced with modern versions (Kizior, et al., 2000). COBOL is just one of the many examples when it comes to legacy software systems.

3.3.3 Types of legacy systems

According to Langer (2016), “a legacy system is an existing application system in operation”. What this means is that there can be generations of legacy systems in an organization. The generations of legacy systems closely follow the generations of programming languages. There are five of these programming language generations (Langer, 2016):

1. First generation: machine code performs actions using binary symbols between the machine and the programming language. It is very unlikely to encounter a legacy system with first generation programming language anymore.
2. Second generation: assembler languages. These languages translate higher level languages to machine code that the hardware can understand. Some mainframe computers are still running on assembly code.
3. Third generation: higher level symbolic languages such as: COBOL, FORTRAN and BASIC. These languages use English keywords and are often specialized. For example, FORTRAN is specialized for mathematics and scientific work, while COBOL was designed for business applications.
4. Fourth generation: even higher-level languages that use English keywords, are more focused on the output of a program rather than how statements need to be written. Because of this, these languages were easier to learn for less technical people. Examples of 4th gen programming languages are Visual Basic, C++ and Delphi. These languages have features such as database querying, code generation and graphic screen generation.
5. Fifth generation: these are known as the rule-based code generation which means artificial intelligence software. These software use knowledge-based programming, where developers do not tell the program how to solve the problem, but rather the programs learn on their own.

In most cases, the legacy system is made with either 3rd or 4th generation programming language (Langer, 2016). There are different tools and practices involved in modernizing and replacing the different generations of legacy systems.

3.3.4 Risks of modernizing legacy system

Legacy systems seldom have a complete documentation specifying the whole system with all its functions and use cases. In most cases, the documentation is badly out of date or missing completely. Many years of maintenance under different developers and managers can cause the documentation to degrade. Without up-to-date documentation specifying the legacy system, it is hard to create a new system that would function identically to the previous one. Companies have often coupled their legacy software with their business processes. Changing legacy software can cause unpredictable consequences to the business

process that relies on it, which in term can cost a lot of money to the company. Legacy systems have business rules as part of the software. Business rules are constraints that companies use to regulate their business functions. For example, a casino might have embedded its rules of winning into its slot machine. If the machines are replaced without the same rules of winning, the company might lose money to the players. The nature of replacing legacy systems with new ones is risky, since the new system can be more expensive than expected and there can be problems with delivering it on time (Sommerville, 2016).

3.3.5 Risk of staying with legacy system

First, legacy system is not compatible with new software, and it cannot interface with internet and other more modern software. There are also too few developers that know old languages or none. Therefore, maintenance needs to be bought from specialized companies, which is expensive (Dogru, et al., 2011).

In some cases, documentation is out of date or completely missing. It might even be that there is no source code, in which case developers needs to map out the functionalities of the software and try to create a new system with similar capabilities (Dogru, et al., 2011).

Sometimes the source code of the legacy system may have degraded over the years. It might be collection of programs from different time working together with questionable logic. Furthermore, the code could have been written in a more optimized manner that new developers do not understand, since it is not necessary with modern hardware and languages (Dogru, et al., 2011).

Legacy systems may also have many different data files from different time periods with duplicate data that can be out of date or corrupted. At worst, the databases might not even be compatible with each other. At some point, the cost of maintaining legacy systems will exceed the cost of replacing them with a new system (Dogru, et al., 2011).

3.3.6 Strategies for updating legacy system

There are three strategies for dealing with legacy systems: scrap the legacy system, keep maintaining the system or replace the whole system (Malinova, 2010). Because companies have limited amount of money to spend on their legacy systems, they need to carefully assess the best option to get the best return of investment. Scrapping the system can be an option when the company has found out that the legacy system no longer creates any value

and is not critical part of any business process. In this case, the best solution is simply getting rid of the legacy system. Continuing to maintain the legacy system can be done when the system is still stable, and maintenance is cost effective. Improving the legacy system is possible when the system has degraded a bit, but it can still be maintained. The idea is to add new interfaces to make the system easier to maintain. Lastly, replacing the legacy system all together should be attempted when there is no longer hardware to support for the system, maintenance is too expensive, and the cost of a new system is not too high (Seacord, et al., 2003).

3.3.7 Deciding what to do

When company is assessing the state of their legacy system to determine what to do with it, it is important to evaluate the legacy system from a technical perspective and from business perspective (Warren, 1998). The technical perspective means that the quality of the legacy system is evaluated. The business perspective is used to determine if the business needs of the company require a new software to perform optimally. By combining these two perspectives, a company can decide on whether they should invest into the legacy system. If a legacy system has low quality and low business value, the system should be removed. Legacy system with low quality but high business value should not be removed, but because of the low quality, its maintenance is expensive, so it should be replaced with a modern system. Systems with high quality and low business value do not create much value, but they do not cost much to maintain. Thus, these systems can be left running. Legacy systems with high quality and high business value are the best possible option. They do not cost much to maintain, and they are creating value to the company. These systems should be left running with normal maintenance (Sommerville, 2016).

3.4 Modernization

System modernization is a similar act of transforming software as maintenance is. However, modernization differs in the sense that it is much more extensive process than maintenance because modernizations often incorporate restructuring, functional changes, and new software attributes. A legacy system can be modernized to lower its maintenance costs if it still has business value.

There are two types of modernization strategies that can be used: white-box modernization and black-box modernization. The key difference between them is the amount of

knowledge of the legacy systems needed to complete the modernization process. White-box modernization requires a lot of technical information about the internals of the legacy system. Contrary to that, black-box modernization only requires information about the external interfaces of the legacy system. Sometimes these methods are not enough, and a legacy system is so outdated that it is not cost effective to modernize it. Thus, it should be replaced (Comella-Dorda, et al., 2000).

3.4.1 White-box modernization

In order to do white-box modernization, it is necessary to reverse engineer the legacy system in order to build up a higher level representation of the system, its relationships and components (Chikofsky, et al., 1990). Program understanding is often used to reverse engineer legacy systems in white-box modernization. In essence, program understanding is the act of examining and studying the legacy system, modeling the system domain, using reverse engineering tools to study the code base, and developing abstract models that help to understand the system (Comella-Dorda, et al., 2000). The task of reverse engineering a system is a time consuming and expensive operation. It is even more difficult if the system is large and complex, and the documentation is outdated. Once there is a sufficient understanding of the legacy system, the restructuring of the code and the system will commence. Restructuring is transforming the code to increase performance, maintainability and other quality attributes, while maintaining the functionality (Chikofsky, et al., 1990).

3.4.2 Blackbox modernization

Black-box modernization does not attempt to uncover the inner workings of the legacy system, since only the inputs and outputs of the system must be examined. This makes the modernization process easier compared to white-box modernization (Comella-Dorda, et al., 2000). Black-box modernization builds new interfaces around the legacy system rather than trying to restructure it.

3.4.3 Replacement

Replacement of a legacy system is needed when the system is so badly outdated that it is not possible or cost effective to modernize it. When replacing legacy system, a new system needs to be built from the ground up to take its place. Building a new system requires a lot of resources and knowledge. Since most developers work on maintenance, they may not

know how to design new systems. Another risk is that some of the business knowledge that is embedded in the legacy system will be lost. Furthermore, the new system might not work as well as the old one (Comella-Dorda, et al., 2000).

3.5 Related research

A Mapping study by Agilar et al. (2016) reviewed 44 software modernization related research papers published between 1995 and 2015. The study tried to find out what are the characteristics of (1) legacy software modernization, (2) what techniques and tools are used to modernize software, and (3) what causes companies to try to modernize software?

3.5.1 Characteristics of modernizing legacy software

According to the study, software modernization of a legacy system can be defined as the evolution of systems, where new features are added according to customer needs and business requirements, and updates are made to remove errors (Agilar, et al., 2016). This definition is based on many existing models about software life cycle. For example, Weiderman et al. (1997) suggests a model, where software evolution is split into three stages: maintenance, modernization and replacement.

3.5.2 Tools for modernization

Regarding to the second question, the study observed three main modernization strategies: white-box modernization, black-box modernization, and replacement. According to the study, approximately 73 % of the companies mentioned in the research papers used white-box methods, while black-box was used by approx. 23 %, and only approx. 3 % used replacement methods. This is surprising, since the popular opinion seems to be that black-box modernization is easier than white-box modernization (Comella-Dorda, et al., 2000). It might be indication that the tools for white-box methods are getting better (Ransom, et al., 1998).

3.5.3 Reasons for modernization

In order to answer the third question, analysis of the of publications revealed four main reasons for modernization of legacy software. First of all, legacy systems are hard to integrate to other software systems, since they were not designed for highly integrated usage that is required in the world of today. Integration allows companies to better control their resources, remove duplicate business rules, re-use existing software and reduce cost of development. The second reason for modernization is that legacy systems are expensive

to maintain, and the maintenance costs will only increase as the system ages (Agilar, et al., 2016). The effort needed to keep legacy systems running often takes away resources from other projects. The maintainance effort for legacy systems can take 50 % to 70 % of the IT budget in a large company (Cimitile, et al., 2000). Often the reasons for rising maintenance costs are bad documentations and the lack of understanding of the legacy systems (Bisbal, et al., 1999). This leads to the problem of choosing between ever more rising maintenance costs and expensive replacement of the legacy system (Bennett, 1994). Third reason exposed by the study is the shrinking knowledge concerning the legacy system (Agilar, et al., 2016). To be able to apply changes to any system, it is necessary to understand the system (Bennett, 1994). Often legacy system's original developers are gone, its documentation is degraded and there is nobody willing to learn its design. Other way to think about it is that legacy systems are like rare diamonds, and just like a rare diamond, they cost a lot. The fourth reason for modernization are the frequent errors and bugs that plague legacy systems. This is caused by the poor documentation and lack of understanding (Bennett, 1994). The study also uncovered that 63 % of the research papers were solution proposals, not empirical studies, pointing out a need for empirical studies from this topic (Agilar, et al., 2016).

4. Basics of microservices architecture

Microservices have gained a lot of popularity recently. In this chapter, the history, and the current state of the microservices are described. It is also explained how microservices differs from traditional monolithic architectures and what are its quality attributes.

4.1 Microservice architecture versus monolithic architecture

The term ´monolithic software´ is used to describe software application of which modules cannot be executed separately, as illustrated in Figure 2. Example of monolithic architecture. Monolithic architecture style has been the norm in software applications for a long time. However, there are some general issues related to monolithic architecture, which are causing migration to microservices (Dragoni, et al., 2017). List of issues is below:

1. Monolithic applications tend to grow indefinitely. This causes the complexity to also increase, which makes maintaining monolithic applications gradually hard. Things like finding bugs and creating new features takes a long time (Dragoni, et al., 2017).
2. If at some point a part of monolithic application needs to be updated, the whole application needs to be restarted. This is fine for small applications, but for large applications this might be a considerable downtime (Dragoni, et al., 2017).
3. Monolithic applications are harder to deploy since some parts of the application might have different requirements to other parts. This means that some parts are computationally heavy, and others are memory heavy. The developers must choose a one-size-fits-all environment to satisfy all requirements, which is expensive and suboptimal (Dragoni, et al., 2017).
4. Handling scalability is another weakness of monolithic applications. When an application is getting a spike in inbound requests, the strategy usually is to create more instances of the application to handle the increased load. However, monolithic applications work as a single monolithic structure and cannot be split. This means that even the parts of the application, that do not need the increased load, still get it, which wastes resources (Dragoni, et al., 2017).

5. Monolithic systems are locked in the same technology and cannot evolve fast. The same programming language and framework must be used from the first to the last module, even though better new options would have appeared in the market (Dragoni, et al., 2017).

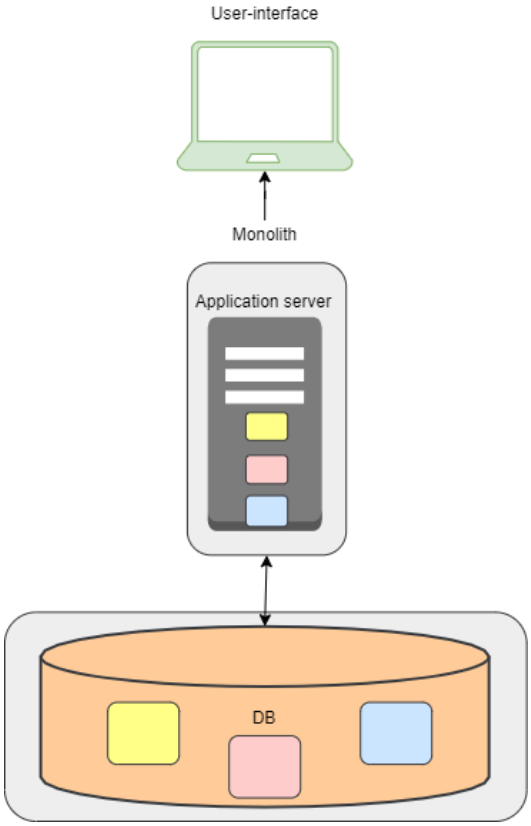

Figure 2. Example of monolithic architecture. Server application and database.

Microservices are cohesive and self-sufficient processes that interact with each other to form a distributed application, as illustrated in Figure 3**.** Example of microservice architecture.. They are small, independent services that have their own isolated environment with operating system, databases, and other support software. Basically, all the modules in an MSA application are microservices. An example of a microservice is a web shop that has a microservice for handling their customer information. All that the microservice does is add, delete, update and list customer information for the web shop. The microservice has no other functions, and it does not know anything else. It is purely focused on the narrow task of managing customer information. Adding multiple

microservices together forms a distributed application. Microservices often use message passing to communicate with each other. This means that a microservice can be built with different programming languages and different environments depending on the requirements (Dragoni, et al., 2017).

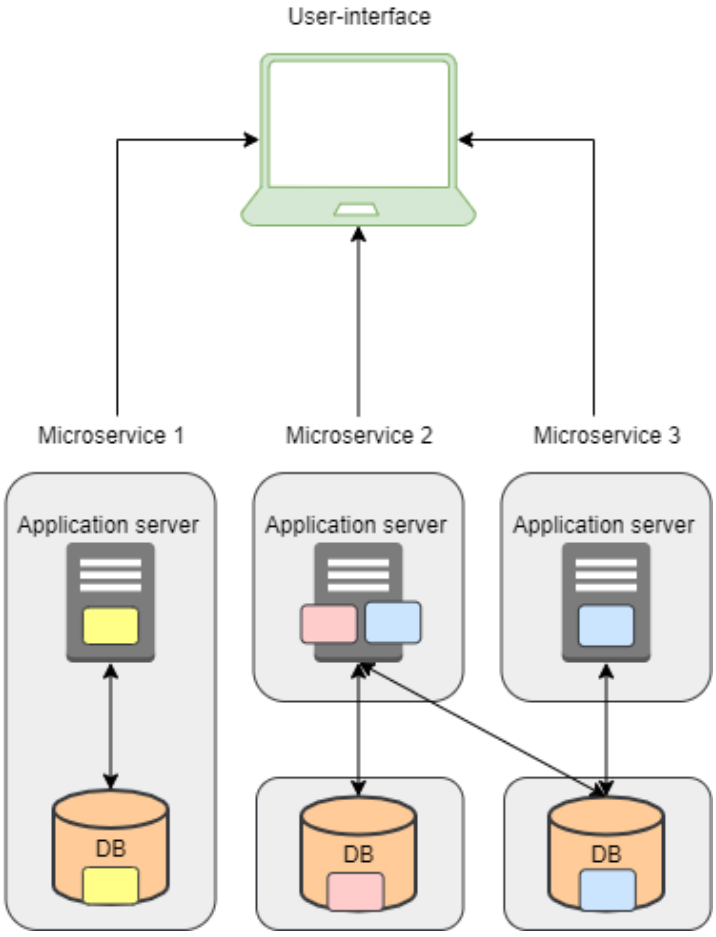

Figure 3. Example of microservice architecture. Multiple microservices applications and databases.

MSA is a guideline for developers and managers, which helps them to develop and implement distributed applications. By following MSA principle, developers work on small, well specified functionalities that are well separated from other parts of the application. This is also true for the higher level microservices that only coordinate the work of other microservices (Dragoni, et al., 2017).

In an earlier part of this chapter, the issues of monolithic architecture were listed. Here is how MSA attempts to cope with the same issues:

1. Contrary to monolithic applications, microservices only implement one or couple of specific functionalities. Because of this, the code bases are small and simple, and finding bugs is easier. Additionally, developers can test their implementations in isolation, because microservices are decoupled form the rest of the system (Dragoni, et al., 2017).
2. When microservices need to be updated to a newer version, there is no need to reboot the whole system. One microservice can be shut down, updated and restarted with minimal downtime (Dragoni, et al., 2017).
3. Microservice applications are usually deployed using containerization. This means that they run on virtualized environments that can be quickly deployed or taken down. Thanks to containerization, they can be configured to ensure the best performance by reducing costs and improving quality (Dragoni, et al., 2017).
4. Because microservices are fundamentally decoupled services, they scale well. There is no need to duplicate the whole application to increase the performance on one part of it. New instances of containerized microservices can be spooled up as need be, this is effective and reduces costs (Maurizio Gabbrielli, et al., 2016).
5. With the exception of communication protocols, there are no language or technology lock-ins imposed on MSA. Developers can choose the best programming language and the best supporting software to support the functionalities of a given microservice (Dragoni, et al., 2017).

4.2 History of microservices architecture

Architecture is what provides a structure to the software development. Using well designed architectures allows software to evolve throughout its life span. Software architecture is bridging the gap between software requirements and the functionality of the software (Dragoni, et al., 2017). In this chapter, the history of software architecture is explored from the early days of object-oriented programming to the emergence of microservices. The whole timeline can be seen in Figure 4.

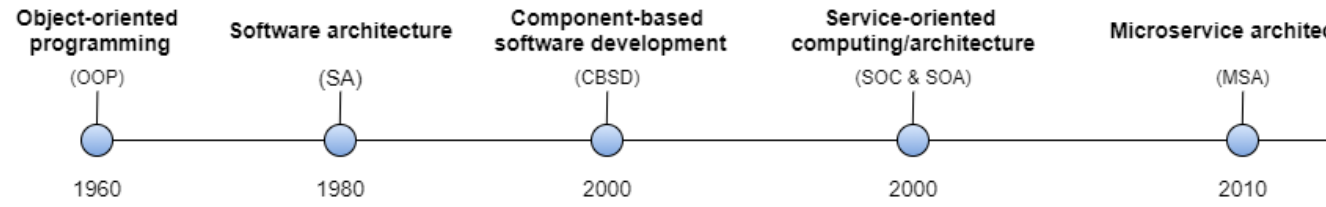

Figure 4. History of microservices (Dragoni, et al., 2017).

4.2.1 Software architecture

In 1960s, the first large-scale software projects started to appear. Developing large-scale software projects was a complex job with many difficulties (Brooks, 1995). During these times, software design was not yet a tightly coupled part of the implementation process – rather it was a separate activity with its special set of tools and documentation. The differentiation made large-scale software hard to develop. This led to growing interest in software design from the research community, who saw its importance in software development. In 1980s, references to software architecture started to emerge. However, it took another decade until the first foundations for software architecture were created by Perry and Wolf in 1992 (Perry, et al., 1992). The definition of software architecture by Perry and Wolf differed from software design, and has since gathered large research community around it. The arrival of object-oriented programming (OOP) in the 1980s and 1990s, added to the software architecture research field. In OOP software architecture was translated to code using software patterns (Dragoni, et al., 2017).

4.2.2 Component-based software development

In the early 2000s, software systems were becoming increasingly large, complex and hard to manage. This led to a decrease in productivity, increase in development costs, and low quality of software that was too risky to migrate towards new technologies (Pour, 1999a). As a consequence, there was a need for a more efficient programming paradigm that would give attention to separation of concerns. To answer this need, component-based software development (CBSD) approach emerged. The idea of CBSD is to built software products using off-the-shelf components with well defined software architecture (Pour, 1998). CBSD is able to reduce development costs, time-to-market and improve maintainability, reliability, and the overall quality of the software (Pour, 1999b).

4.2.3 Service-oriented computing

In the last decade, there has been a shift towards service first paradigm, which further evolved to MSA (W3C, 2004). Service-oriented computing (SOC) is an emerging software architecture paradigm for distributed systems. It's origins can be traced back to OOP and component-based computing (CBC). SOC attempts to solve the complexity of distributed systems and the problems of integrating multiple applications together (MacKenzie, et al., 2006). SOC is based around services, which can connect and understand each other. Connecting can be done via message passing using HTTP and TCP. Services can understand each other by using the same format, for example, XML and JSON. The philosophy behind SOC is to connect large enterprise systems together and make them interoperable (Munindar, et al., 2004).

The benefits of SOC are as follows: dynamism, modularity, distributed development, and integration of legacy systems and heterogenous systems. Dynamism means that new instances of a service can be launched to increase the capacity of the system. Modularity allows services to be constructed by combining other services together, which increases code reuse. Because of well defined interfaces, services can be built apart from each other allowing for distributed development. Communication between services is done with standard protocols, which means that legacy sytems only need to implement the protocols to be able to communicate with new services (Dragoni, et al., 2017).

The component based computing used in SOC has its origins in OOP literature. However, there are some critical differences that led to the separation of the research paths. SOC was and still is built on top of OOP languages. This is caused by the broad merging of the two paradigms in the early 2000s. However, one should be careful when comparing the objects and the services. The philosophy of object is to focus on encapsulation and to work on shared memory, whereas service's philosophy is independent deployment, and it works using message passing. Both paradigms share the common idea of componentization (Dragoni, et al., 2017).

Microservices are the second generation of SOC and service oriented architecture (SOA). It attempts to fix many of the issues of the earlier generations by stripping away unneeded complexity, and squarely focus on the development of simple services that only implement one functionality. Microservices uses the same service-oriented languages as SOA did (Dragoni, et al., 2017).

4.3 Microservice architecture

Microservices architecture is a new service-oriented paradigm. MSA works by combining simple light weight services together with message passing. Microservices are small applications with few functionalities running on isolated environments. The MSA is built on similar principles as SOA (MacKenzie, et al., 2006). ‘Microservice’ as a term was first coined by a workshop group in 2011 to describe an architectural style that they had been recently examining (Fowler, et al., 2014). The MSA had been used before this, but with a different name. For example, Netflix used the term ‘fine-grained SOA’ for their microservice-like architecture (Wang, et al., 2013).

Microservices are combatting the complexity of large software systems by splitting off their functionalities to small, easily managed services. Microservices are independent in development and deployment, making MSA applications highly modular. High degree of modularization has many benefits, such as low coupling, high cohesion, maintainability, scalability and faster development and deployment (Dragoni, et al., 2017).

MSA is often confused as a branch of SOA and to a degree that is correct (Fowler, et al., 2014). However, there are some key characteristics that differentiate MSA from SOA, for example, size, bounded context, and independency (Dragoni, et al., 2017). Microservices are small, and they are supposed to only support a single business capability. This is contrary to SOA service, where a typical service can range from a small to large enterprise service. Component sharing is another difference in MSA. While SOA attempts to share as many of the same components as possible, MSA tries to minimize it with bounded context (Tuli, 2018). Bounded context allows microservices to have different meanings for different data models depending on the context. This means that multiple versions of a certain entity can exists in the microservice system without clashing (Fowler, BoundedContext, 2014). Lastly, all microservices operate independently from each other, only communicating through published interfaces (Dragoni, et al., 2017).

MSA has become popular due to its key characteristics, flexibility, modularity, and evolution. Being flexible and able to change fast depending business needs, gives companies that adopt MSA a competitive advantage. Modular design allows companies to create new services and products for their customers and provide customized solutions.

Low coupling and high maintainability allow developers to evolve their microservice systems, and create new features for customers (Dragoni, et al., 2017).

4.3.1 Development teams

In MSA, teams are created to be cross-functional, as shown in Figure 5, meaning that a single team will have all the skills needed to develop a microservice. This includes user-interface (UI), storage persistence and server-side development. Microservices are divided according to business capabilities. The reason for this organization is Conway's law (Fowler, et al., 2014). The law states the following: "*Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.*" - Melvin Conway (1968). In practice, this means that traditionally teams are split along the technology layer, as illustrated in Figure 6. For example, there can be UI teams, server teams and database teams. This is a fragile way to organize teams, since even small changes require cross-team project, which is ineffective and expensive. Often this forces teams to implement changes to the first technology layer that they have access to. This means that before long there is logic in all the layers. Because developers write logic to technology layer that is closest to them (Fowler, et al., 2014).

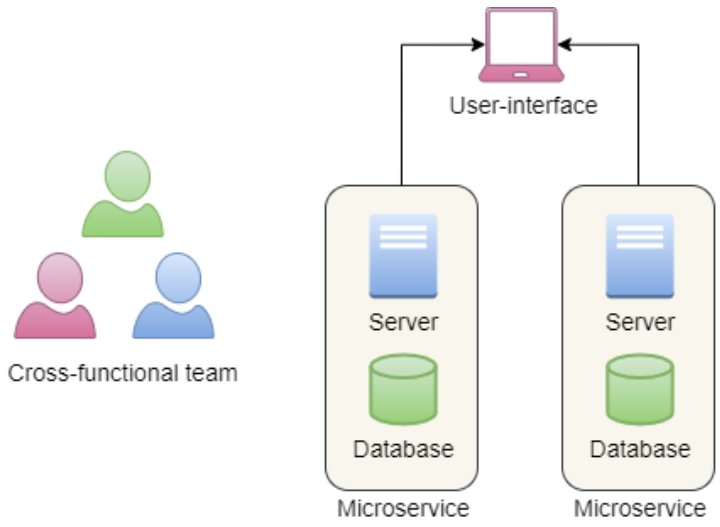

Figure 5. Cross-functional team organized based on business capabilities.

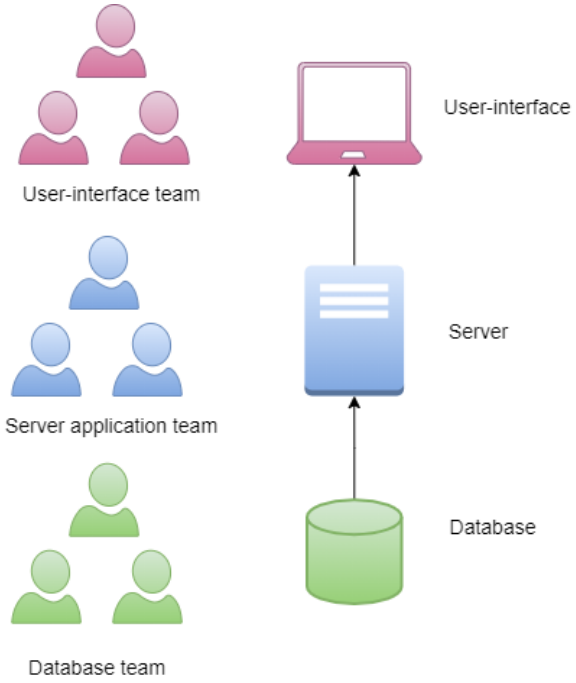

Figure 6. Development teams that have been grouped by technology layer.

4.3.2 Pipeline automation

There has been a huge surge of development of infrastructure technology in the past ten years. Cloud service platforms like AWS and Azure have made operating and deploying microservice based applications easier and cheaper. This allows teams to build microservices using continuous integration (CI) and continuous delivery (CD) pipelines. An example of this can be seen in Figure 7. The pipeline automates testing, building, deploying and delivering the microservices. This makes developing and deploying new features easier and faster, since developers can rely on their pipeline to handle the validation of code and deployment to end-users (Fowler, et al., 2014). A microservice can be updated independently without distrupting the functioning of other microservices. MSA was developed after the popularization of CI and CL. Furthermore, MSA is designed to work with CI and CL to automize the software development lifecycle (Dragoni, et al., 2017).

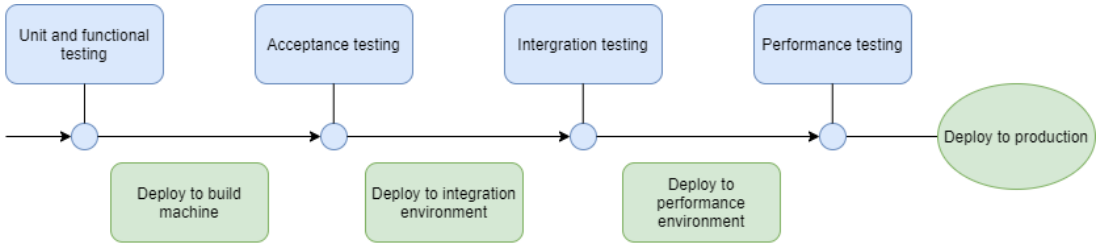

Figure 7. Automation pipeline example.

4.3.3 Availability of microservices

For microservices availability is critical. Because microservices depend on each other, the availability of the whole system can be evaluated by examining the availability of a single microservice. A single microservice failing can cause failures in other services that are part of the same microservice system (Dragoni, et al., 2017). It has been found out that small software components, such as microservices, have high fault density (Hatton, 1997). However, the fault density also grows when the component's size grows (Emam, et al., 2002). Microservices are designed to be small, and if they grow too much, they should be split into multiple microservices. This should keep microservices in the optimal size regarding fault density (Dragoni, et al., 2017).

4.3.4 Reliability of microservices

MSA is distributed by design, since microservices communicate through message passing. Because of this, reliable communication is critical between microservices (Dragoni, et al., 2017). MSA systems are built from small components with clearly defined interfaces. This is a way to achieve high reliability in a system (Raymond, 2003). For communication, the microservices community favors the 'smart endpoint and bum pipers' approach. Applications designed with MSA try to be as decoupled and cohesive as possible; they have their own domain logic that processes the incoming request returning a response. The most used communication protocols with MSA are HTTP and Protocol Buffers (Fowler, et al., 2014).

4.3.5 Maintainability of microservices

Often applications are developed in a project model, where development team builds a software application. Once they are finished with it, the application is handed over to a maintenance team. However, amongst the microservices community, this model is not

favored. Instead, a common practice for microservices, development is for development teams to own their products throughout the product's lifetime (Fowler, et al., 2014). This idea is inspired by Amazon's "you build it, you run it" philosophy (Vogels 2006). Developers working daily with their software and communicating with the customers creates a feedback loop that helps further improve the software (Fowler, et al., 2014). Microservices systems are by design decoupled, meaning that there are no strong dependencies between individual microservices. This adds to maintainability by reducing the cost of modifications, fixing problems, and creating new features (Dragoni, et al., 2017).

4.3.6 Performance of microservices

The greatest performance weakness for microservices system is the communication that happens over a network. As a comparison, monolithic application communicates inside a single process using the same shared memory. However, microservices must communicate across the network to pass messages. This decrease of performance is dependent on the number of external connections each microservice has. If a microservice system has well defined bounded contexts, it will experience less of a performance hit (Dragoni, et al., 2017). The issues related to microservice connectivity can be mitigated in two ways, making network calls less frequently and using asynchronous calls. The latter way means that asynchronous network calls should be made parallel to each other, so that only the slowest of the calls needs to be waited (Fowler, 2015).

4.3.7 Security of microservices

MSA have the same security vulnerabilities as any other distributed architecture. MSA has the same security issues as its predecessor SOA (O'Brien, et al., 2005). Microservices transfer data across network to other microservices, which can always be a point of attack. This means that there needs to be additional encryption of data that is send between the microservices. Furthermore, the distributed nature of microservices means that there is more surface on which security breaches can be targeted. Another security issue can raise from the complexity of microservice systems. Even a moderately sized monolithic application can be split into tens or hundreds of microservices. This is a lot of applications to take care of, and creates opportunities for attackers (Dragoni, et al., 2017).

5. Results of the systematic mapping study

This chapter reports on the results of the systematic mapping study. The goal of the study was to discover trends related to legacy software modernization, more specifically, trends in migrating from legacy applications to microservices. The study was conducted as a systematic mapping study. A pool of 450 research papers was the starting point. After exclusion analysis, 44 of them were chosen for further classification. Three classification schemas were developed to map out the research subject: research type, research area and research tool. These classifications are directly related to the research questions.

5.1 Results to the first research question

Regarding the first research question "(RQ1) What are the research approaches?", the research types were analyzed using a modified version of the classification schema by Wieringa (2006). The classification schema consists of the following gategories: validation research, evaluation research, solution proposal, philosophical papers, opinion papers and experience papers, as listed in Table 1 in section 2.5. The distributions of research types can be seen from the pie chart in Figure 8.

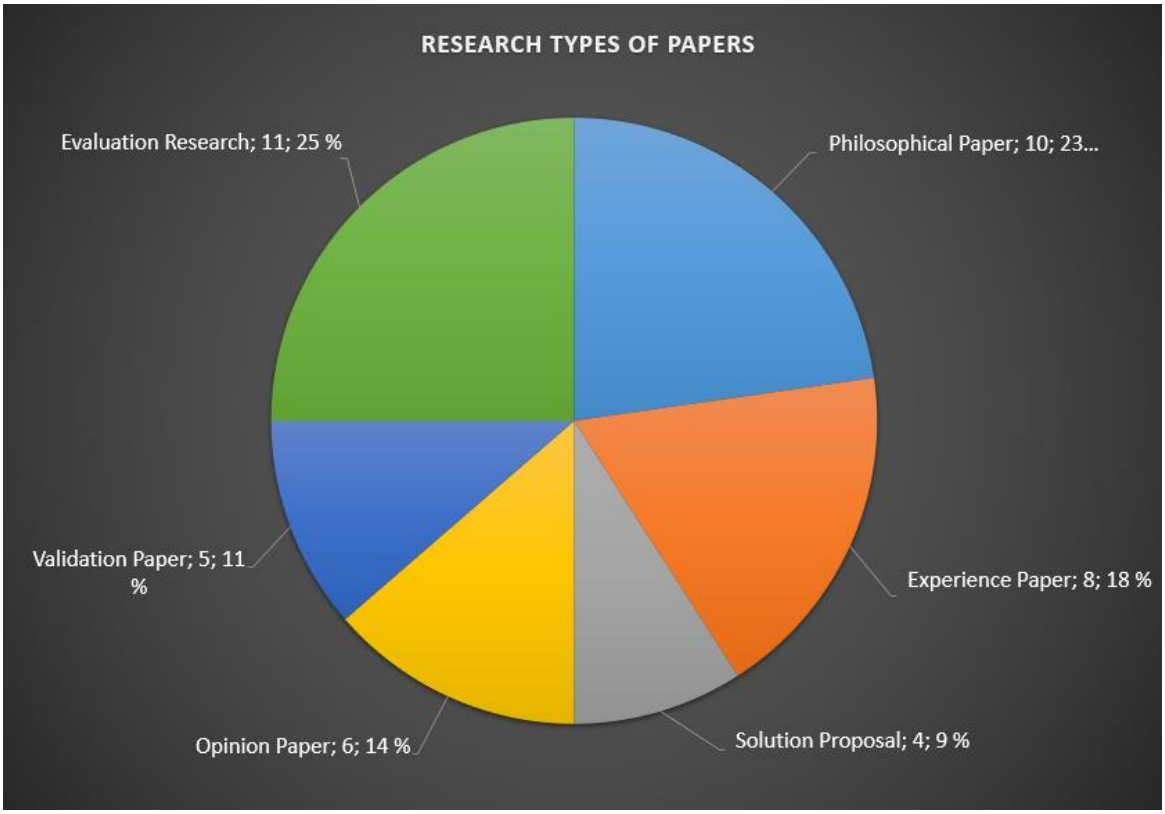

Figure 8. The distribution of research types.

5.1.1 Implications

The research types can be divided into empirical and non-empirical research. Non-empirical types are solution proposal, opinion papers, experience papers and philosophical papers. Empirical study types are validation and evaluation research. Most of the studies (64 %) are non-empirical studies (Figure 9). This suggests a lack of empirical studies (36 %), which use verified data and observations in order to support research results. Empirical studies are critical for validating theories, models, tools and other migration related artifacts. The lack of empirical studies might be because researchers have a limited access to modernization projects.

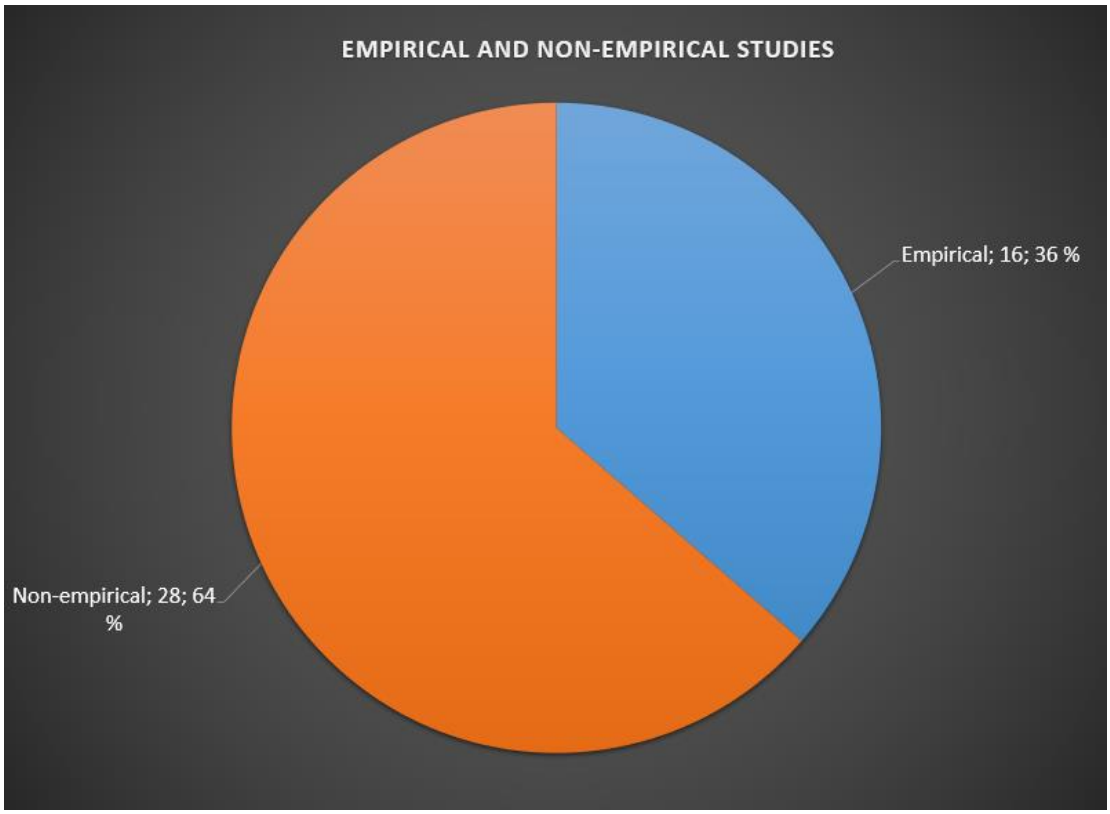

Figure 9. Research types divided by empirical and non-empirical studies.

5.2 Results to the second research question

Second research question deals with the research area: (RQ2) what areas of migrating to MSA are addressed? The research area reflects specific problems and trends that researchers are trying to figure out within the context of migrating to microservices. The distribution of research areas can be seen in Figure 10. It is worth noting that a research

paper can cover multiple research areas, hence the sum of all research areas (48) is greater than the sum of all the research papers (44).

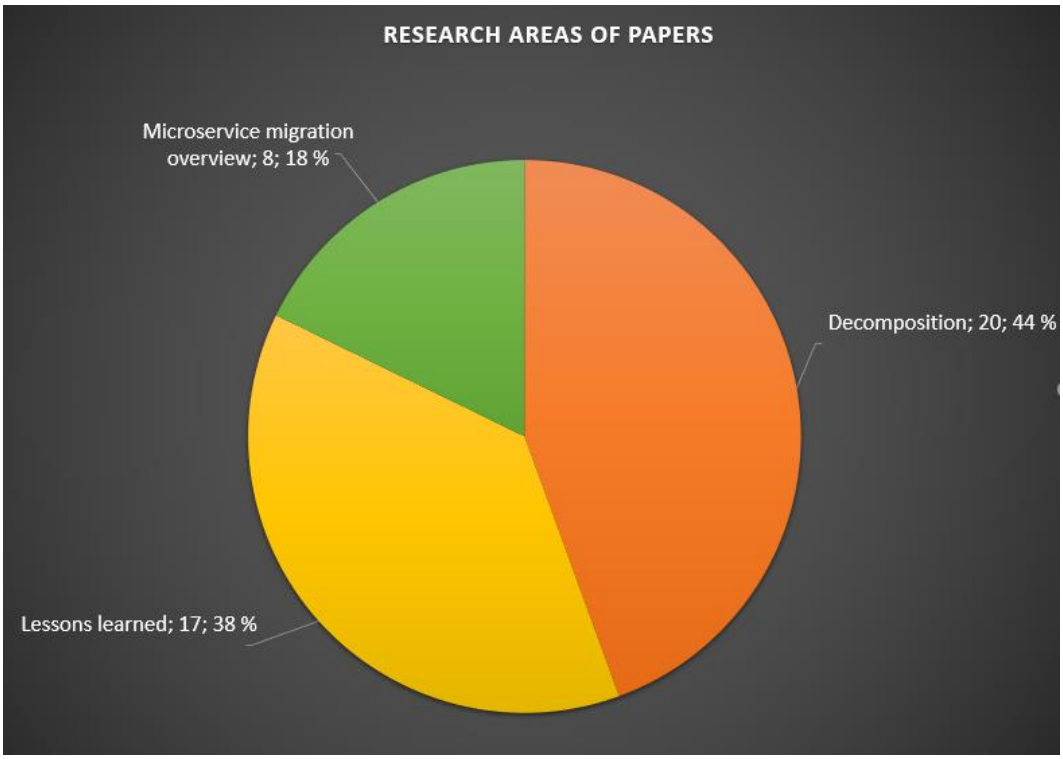

Figure 10. The distribution of research areas.

5.2.1 Decomposition

A large portion of the research papers dealt with decomposition (44 %). Research papers in the decomposition category discuss the best practices, challenges, and experiences of decomposing a legacy system to microservices. Knoche and Hasselbring (2018) present a process to decomposing a large COBOL system consisting of over a million lines of code. Their process establishes platform-independent interfaces that are based on bounded context, improves maintainability and eliminates unnescessary parts of the application. The process first mapped out and implemented the required functionalities to services in the legacy application before migrating them to microservices in Java. This a less risky approach and should only be done in large applications (Knoche, et al., 2018). A common problem in decomposition is recognizing the microservices candidates from legacy application (Escobar, et al., 2016; Kamimura, et al., 2018).

Escobar et al. (2016) created a model based approach for visualizing the structure and dependencies of legacy applications. Their validation of the approach on a mid sized legacy application shows that the visualization greatly improved understandability.

Kamimura et al. (2018) developed a method to identify microservice candidates. The method uses clustering algorithm to recognize relations between programs and data. Additionally, the method also visualizes the candidates to give an overview of the system to developers. This method was validated with two studies.

Balalaie et al. (2015) present a set of migration patterns that they have collected from various microservices migration projects. The idea of these patterns is to support organizations in planning their microservice migrations. The patterns are validated using qualitative empirical research.

5.2.2 Lessons learned

Another large portion of the papers (37 %) reported on lessons learned. Papers categorized as lessons learned report on the positive and negative experiences that researcher have encountered during a microservices migration. Balalaie (et al., 2015) share their experiences on migrating software as a service (Saas) application to microservices. They conclude that microservices are not a silver bullet solution that fits every situation, because of the complexities that are introduced to the system. Additionally, they recommend considering the distributed complexities and other factors related to microservices architecture before implementing it. However, they recommend microservices architecture for situations where flexibility, scalability and availability are required.

Furda et al. (2018) discuss three challenges of microservice migration: multitenancy, statefulness and data consistency. In their article, they explain that the best solutions for these challenges are removing statefulness from migrated legacy code, implementing multitenancy, and paying increased attention to data consistency.

Jamshidi et al. (2018) explain in their research paper the recent history of microservices and future challenges. They conclude that there is a growing number of research papers from the topic of microservice migrations. However, the papers have had a little effect on microservices practices, because researchers don’t have access to large scale projects.

Lenarduzzi et al. (2020) share the results of their empirical study that monitored the technical debt of migrating from legacy application to microservices. They found out that after the migration technical debt grows 90 % slower. When the code normalizes, technical debt decreases and starts growing linearly, with a growing trend much lower than the monolithic legacy system. Additionally, they give three lessons learned to help future

development. First, to create a microservice template that can easily be dublicated for future microservices to decrease development time. Second, to create a microservices strategy before hand, to reduce rework. Third, not to delay important architectural decision making, as this will cause more effort in the future.

5.2.3 Overview

The third category is microservices migration overview (18 %). Papers, in this category, describe the overall process of migrating to microservices. Larrucea et al. (2018) present a concise overview of migration to microservices. They share a general process for migrating to microservices. The process has five steps: preparing for organizational changes, getting to know the legacy system while noticing there are some special tools that can help with this, creating system architecture that best fits the legacy system in question, and include DevOps, enabling features in it, testing different migration methods to learn which one is best suited for the project, and lastly, creating design that supports testing, developing and integration. The design should use DevOps tools such as Git and Fuge (Larrucea, et al., 2018).

Ramachandran et al. (2020) introduces a methodology for migrating monolithic applications to cloud based microservices. This methodology can be split into two approaches: blank-slate approach and migration approach. In the blank-slate approach, the microservices application is developed as cloud based microservices from day one. Whereas in the migration approach, the existing legacy application is migrated one function at a time. The blank-slate approach is a better option for new applications and the migration approach for legacy applications.

A clear trend in microservices related papers are the benefits gained from them. Mazzara et al. (2018) give an overview migrating a large scale legacy system to microservices architecture. In their paper, they highlight the benefits gained from the migration. The migration led to higher cohesion, lower coupling and reduced complexity. Furthermore, they report that compared to monolithic architecture, microservices have better scalability.

5.2.4 Implications

Identifying bounded contexts from legacy system seems to be a core issue. It requires a thorough understanding of the system (Escobar, et al., 2016; Kamimura, et al., 2018). Many of the papers present processes and methods for identifying parts of a legacy system that can be transferred to microservices. To understand legacy systems better, some practitioners like Escobar et al. (2016) have developed visualization tools. However, Balalaie et al. (2015) points out, it might be that there is no definite process for migrating to microservices. Instead, organizations need to pick the best approach for their projects.

5.3 Results to the third research question

In this section, the third research question: "(RQ3) what are the contributions to modernization from monolithic legacy systems to MSA?", is answered. In total, six different contribution categories were indentified from the research papers: tool, metric, model, method, process and data. The distribution of research contributions can be seen in Figure 11. Majority of the contributions are either processes (33 %) or data (43 %). A process describes series of actions that can or should be taken in order to accomplish a predetermined goal. Papers with data as their contribution, share information, survey study results and personal experiences related to microservices migration. After data and process, the next most popular contribution is method, with 11 % of the total.

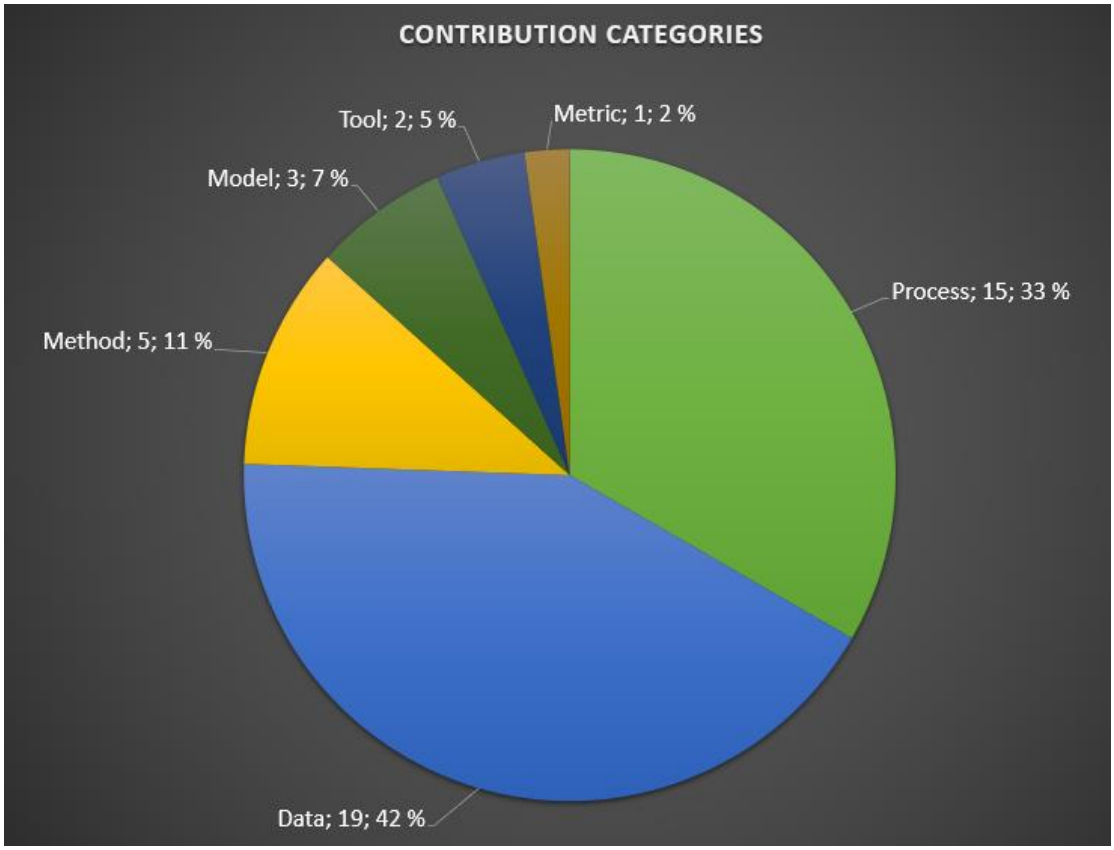

Figure 11. Distribution of research paper contributions.

To better understand the contributions, they were further classified according to the contribution type. For this classification, two categories were used: managerial contribution and technical contribution. Agilar et al. (2016) used this classification method in their research for similar purpose. Managerial contribution describes process, method or approach that manages the migration process. Technical contribution might be a tool, metric, or piece of software to support migration effort. The classification by contribution type can be seen in Figure 12.

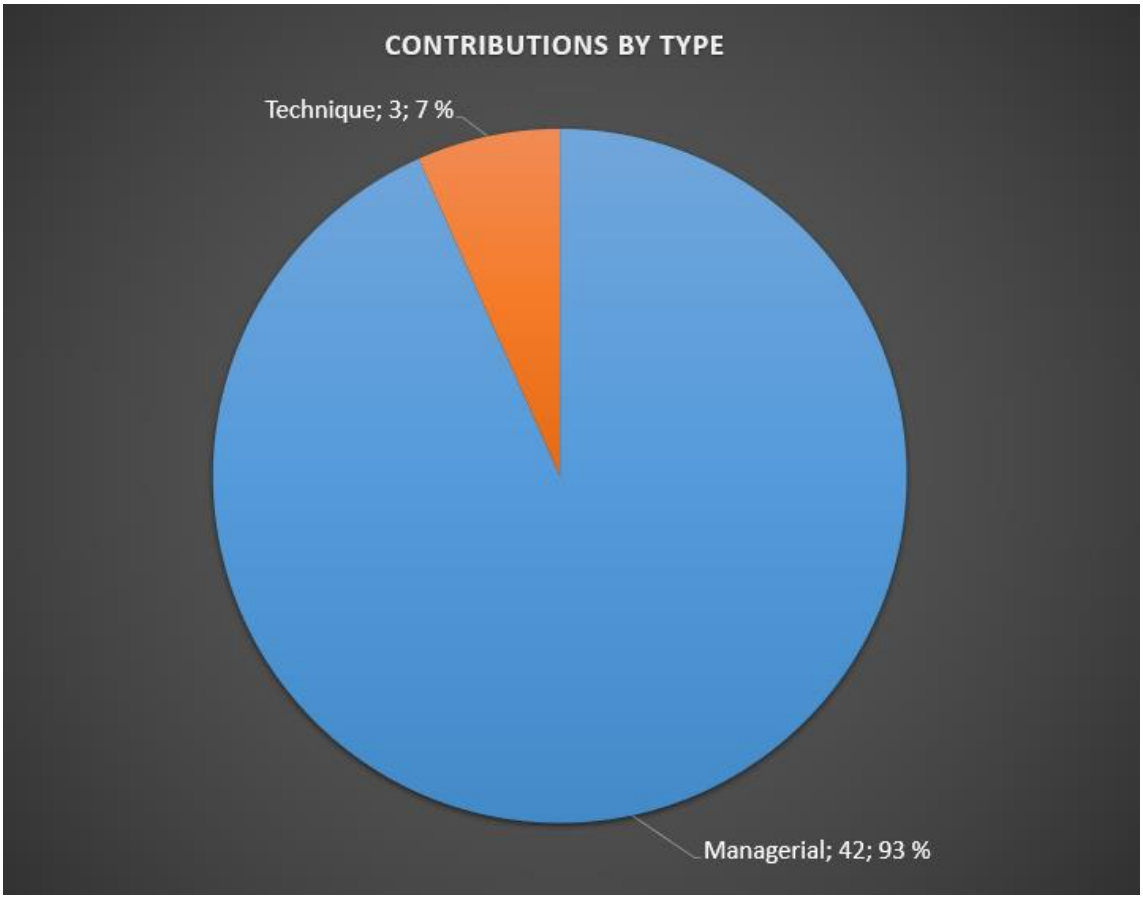

Figure 12. Research contribution types, technical and managerial.

5.3.1 Implications

The large difference between technical and managerial contributions might be that managerial contributions are more relevant in the migration process. This is probably why a large portion of the papers are suggesting migration processes and strategies. Tools for migration are not nearly as relevant as the data and strategy of conducting migration to microservices (Agilar, et al., 2016).

6. Discussion

This research study answers to three research questions related legacy system migrations to MSA.

- RQ1: What are the research approaches?
- RQ2: What areas of migrating to MSA are addressed?
- RQ3: What are the contributions to modernization from monolithic legacy systems to MSA?

The results to the first research question suggest that there is a lack of empirical papers. This may be due to a limited access researcher have on industry-scale MSA migration projects (Jamshidi, et al., 2018). The results to the second research question show that there are three dominant focus areas: decomposition, overview and lessons learned. Furthermore, it is suggested that there is no silver bullet approach to MSA migration, but rather organizations need to find best solution for their project (Balalaie, et al., 2015). Related to third research question, it is found that most common research contributions were data and process. Further analysis shows that over 90 % of the research contributions are managerial opposed to technical. This suggests that MSA migration is predominantly a managerial issue (Agilar, et al., 2016).

Agilar et al. (2016) conducted a systematic mapping study about legacy system modernization. In the study, they classified research papers by research type and contribution. For research type, they used the classification schema by Wieringa (2006) as is used in this paper. Furthermore, they also report a lack of empirical studies in their results. Similarly, their results on research contributions match what is reported on this paper: most of the contributions are related to managerial aspects of the modernization rather than the technical tools (Agilar, et al., 2016). As suggested by Pahl et al. (2016) research related to microservices is still in an early stage. More experimental and empirical work needs to be done.

There has not been a research paper looking specifically into the trends of MSA migration related research. However, there are similar trends seen in other software modernization related research. The research, in this paper, is limited to the research discussing the migration of legacy systems to MSA. Possible threat to validity is bias in the selection of research papers. The research papers were chosen manually by one person. The bias was

mitigated by following strict criteria explained in section 2.4. The starting point for this research was 450 search result from Google Scholars. After the exclusion and inclusion analysis 44 research papers were chosen. The criteria stated that a paper should be at least 4 pages long with English as a primary language and published between 2015 and 2020. It was also demanded that a paper was publicly available, and it was neither a master's nor bachelor's thesis. Furthermore, it needed to relate explicitly to migration of legacy systems to MSA. This study should be possible to reproduce by other researchers.

7. Conclusion

The aim of this paper is to explain what legacy systems are, how to modernize them and describe what microservice architecture (MSA) is. Additionally, a systematic mapping study was performed to map out the trends in MSA migration research. Legacy system modernization is no trivial task and requires considerable effort from any organization attempting it. In the past decade, a new modernization trend has surfaced.

Modernizing legacy systems to MSA is seen as the best option for large distributed systems. MSA provides scalability, maintainability, and higher efficiency, while lowering costs. Furthermore, for this paper a SMS was conducted to understand the research related to microservice migrations better. Based on the results of the study, it can be concluded that in the future, there is a need for more empirical studies in this subject. There is no silver bullet approach to microservice migration, as MSA migration is primarily a managerial process where technical tools have less of an effect.

8. Appendix

Index	Research paper	Link
1	Using Microservices for Legacy Software Modernization	https://ieeexplore.ieee.org/document/8354422
2	On the Modernization of ExplorViz towards a Microservice Architecture	http://ceur-ws.org/Vol-2066/emls2018paper01.pdf
3	Towards the Understanding and Evolution of Monolithic Applications as Microservices	https://ieeexplore.ieee.org/document/7833410
4	Analysis of Legacy Monolithic Software Decomposition into Microservices	http://ceur-ws.org/Vol-2620/paper4.pdf
5	Migrating to cloud-native architectures using microservices: an experience report	https://arxiv.org/pdf/1507.08217.pdf
6	Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices	http://eprints.uni-kiel.de/31830/1/ICPE2016Knoche.pdf
7	Migrating Legacy Software to Microservices Architecture	https://ieeexplore.ieee.org/document/8732170
8	Drivers and Barriers for Microservice Adoption – A Survey among Professionals in Germany	http://eprints.uni-kiel.de/45188/1/EMISA2019.pdf
9	Migrating enterprise legacy source code to microservices: on multitenancy, statefulness, and data consistency	https://ieeexplore.ieee.org/document/8186442
10	Microservices: The Journey So Far and Challenges Ahead	https://ieeexplore.ieee.org/document/8354433
11	Microservices	http://www.rcolomo.com/papers/319.pdf
12	Microservice Decomposition via Static and Dynamic Analysis of the Monolith	https://ieeexplore.ieee.org/document/9095638
13	PRINCIPLES OF THE NEWDIMENSIONS SOFTWARE CREATION FOR A CONTROL CENTRE OF THE FUTURE: CLOUD COMPUTING AND SOFTWARE ARCHITECTURE	http://library.witpress.com/viewpaper.asp?pcode=CR20-032-1
14	Extracting Candidates of Microservices from Monolithic Application Code	https://ieeexplore.ieee.org/abstract/document/8719439
15	Microservices migration patterns	https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2608
16	Migrating to Microservices	https://link.springer.com/chapter/10.1007/978-3-030-31646-4_3

17	Microservices	https://ieeexplore.ieee.org/document/8354423
18	Translating a Legacy Stack to Microservices Using a Modernization Facade with Performance Optimization for Container Deployments	https://link.springer.com/chapter/10.1007/978-3-030-40907-4_14
19	From Monolith to Cloud Architecture Using Semi-automated Microservices Modernization	https://link.springer.com/chapter/10.1007/978-3-030-33509-0_60
20	Using Microservices and Software Product Line Engineering to Support Reuse of Evolving Multi-tenant SaaS	https://dl.acm.org/doi/10.1145/3106195.3106224
21	Cracking the Monolith : Challenges in Data management to Cloud Native Architectures	https://dl.acm.org/doi/abs/10.1145/3241403.3241440?download=true
22	Does Migrate a Monolithic System to Microservices Decreases the Technical Debt?	https://research.tuni.fi/uploads/2019/03/ff7209ff-1902.06282.pdf
23	Microservices: Migration of a Mission Critical System	https://dsg.tuwien.ac.at/team/sd/papers/Zeitschriftenartikel_2019_SD_Microservices.pdf
24	Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example	https://link.springer.com/chapter/10.1007/978-3-030-33624-0_4
25	Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems	https://link.springer.com/chapter/10.1007/978-3-030-03596-9_3
26	A Decoupled Health Software Architecture using Microservices and OpenEHR Archetypes	https://www.researchgate.net/publication/342282120_A_Decoupled_Health_Software_Architecture_using_Microservices_and_OpenEHR_Archetypes
27	From Monolith to Microservices: A Classification of Refactoring Approaches	https://link.springer.com/chapter/10.1007%2F978-3-030-06019-0_10
28	Migrating Towards Microservice Architectures: An Industrial Survey	https://ieeexplore.ieee.org/document/8417114
29	Strategies Reported in the Literature to Migrate to Microservices Based Architecture	https://link.springer.com/chapter/10.1007/978-3-030-14070-0_81
30	Microservices Migration in Industry: Intentions, Strategies, and Challenges	https://ieeexplore.ieee.org/document/8919170
31	A Model-Driven Approach Towards Automatic Migration to Microservices	https://link.springer.com/chapter/10.1007/978-3-030-39306-9_2
32	An Experience Report on the Adoption of Microservices in Three Brazilian Government Institutions	https://dl.acm.org/doi/10.1145/3266237.3266262
33	Migrating towards microservices: migration and architecture smells	https://dl.acm.org/doi/abs/10.1145/3242163.3242164

34	Discovering Microservices in Enterprise Systems Using a Business Object Containment Heuristic	https://link.springer.com/chapter/10.1007/978-3-030-02671-4_4
35	Practical Use of Microservices in Moving Workloads to the Cloud	https://ieeexplore.ieee.org/document/7742277
36	Migrating Web Applications from Monolithic Structure to Microservices Architecture	https://dl.acm.org/doi/abs/10.1145/3275219.3275230
37	Towards Identifying Microservice Candidates from Business Rules Implemented in Stored Procedures	https://ieeexplore.ieee.org/document/9095626
38	The pains and gains of microservices: A Systematic grey literature review	https://www.sciencedirect.com/science/article/abs/pii/S0164121218302139
39	State of the Practice in Service Identification for SOA Migration in Industry	https://link.springer.com/chapter/10.1007/978-3-030-03596-9_46
40	Migrating from monolithic architecture to microservices: A Rapid Review	https://ieeexplore.ieee.org/document/8966423
41	Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems	https://arxiv.org/abs/1605.03175
42	Migration to Microservices: Barriers and Solutions	https://link.springer.com/chapter/10.1007/978-3-030-32475-9_20
43	From a Monolithic Big Data System to a Microservices Event-Driven Architecture	https://www.researchgate.net/publication/341822188_From_a_Monolithic_Big_Data_System_to_a_Microservices_Event-Driven_Architecture
44	Automatic Microservices Identification from a Set of Business Processes	https://link.springer.com/chapter/10.1007/978-3-030-45183-7_23

9. References

Agilar, E., Almeida, R., & Canedo, E. (2016). *A Systematic Mapping Study on Legacy System Modernization.* SEKE. doi:10.18293/SEKE2016-059

Allen Wang, S. T. (2013, 1 28). Announcing Ribbon: Tying the Netflix Mid-Tier Services Together. Retrieved from https://netflixtechblog.com/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62

Bailey, J., Budgen, D., Turner, M., Kitchenham, B., Brereton, P., & Linkman, S. (2007). *Evidence relating to Object-Oriented software design: A survey.* IEEE. Retrieved from https://ieeexplore-ieee-org.ezproxy.cc.lut.fi/document/4343786

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2015). Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. Retrieved from http://arxiv.org/abs/1507.08217

Bass, L., Clements, P., & Kazman, R. (2013). Software architecture in practice. Upper Saddle River, NJ. Retrieved from http://jz81.github.io/course/sa/Software%20Architecture%20in%20Practice%20(3rd).pdf

Bennett, K. (1994). *Legacy Systems Copint with Success.* IEEE Software. Retrieved from https://ieeexplore-ieee-org.ezproxy.cc.lut.fi/stamp/stamp.jsp?tp=&arnumber=363157

Brooks, F. P. (1987). *No Silver Bullet: Essence and Accidents of Software Engineering.* IEEE Computer. Retrieved from http://worrydream.com/refs/Brooks-NoSilverBullet.pdf

Brooks, F. P. (1995). The mythical man-month: essays on software engineering. (p. 322). Addison-Wesley Pub. Co.

Carrasco, A., Bladel, B., & Demeyer, S. (2018). Migrating towards microservices: migration and architecture smells. (pp. 1–6). New York, NY, USA: Association for Computing Machinery. doi:10.1145/3242163.3242164

Chen, R., Li, S., & Li, Z. (2017). From Monolith to Microservices: A Dataflow-Driven Approach. (pp. 466-475). IEEE Xplore. doi:10.1109/APSEC.2017.53

Chikofsky, E., & Cross ll, J. (1990). *Reverse Engineer and Design Recovery: A Taxonomy.* IEEE Software. Retrieved from https://ieeexplore-ieee-org.ezproxy.cc.lut.fi/stamp/stamp.jsp?tp=&arnumber=43044

Cohen, J., Teleki, S., Brown, E., DuRette, B., Brown, S., & Fuller, B. (2006). Best Kept Secrets of Peer Code Review. Retrieved from https://static1.smartbear.co/smartbear/media/pdfs/best-kept-secrets-of-peer-code-review_redirected.pdf

Comella-Dorda, S., Wallnau, K., Seacord, R., & Robert, J. (2000). *A Survey of Legacy System Modernization Approaches.* Carnegie Mellon University. Retrieved from https://www.researchgate.net/publication/235126722_A_Survey_of_Legacy_System_Modernization_Approaches

Computerworld. (2012, 2 16). Cobol brain drain: Survey results. Retrieved from https://www.computerworld.com/article/2502430/cobol-brain-drain--survey-results.html

Conway, M. E. (1968, 4). How Do Committees Invent? Thompson Publications, Inc. Retrieved from http://www.melconway.com/Home/Conways_Law.html

Dayini-Fard, H. (1999). *Legacy Software Systems: Issues, Progress, and Challenges.* IBM. Retrieved from www.cas.ibm.com/toronto/publications/TR-74.165/k/legacy.html

De Alwis, A., Barros, A., Fidge, C., Polyvyanyy, A., Panetto, H., Debruyne, C., . . . Meersman, R. (2018). Discovering Microservices in Enterprise Systems Using a Business Object Containment Heuristic. (pp. 60-79). Springer International Publishing. doi:10.1007/978-3-030-02671-4_4

Dewayne E. Perry, A. L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes, 17*(4), 40-52. doi:10.1145/141874.141884

Dogru, A., & Bicer, V. (2011). *Modern Software Engineering Concepts and Practices: Advanced Approaches.* Retrieved from https://books.google.fi/books?id=mor9t6v-AAYC&pg=PA98&dq=software+modernization&hl=en&sa=X&ved=2ahUKEwjHlM6oibrtAhWNAxAIHQmUDhUQ6AEwAXoECAIQAg#v=snippet&q=legacy&f=false

Emam, K. E., Benlarbi, S., Goel, N., Melo, W., Lounis, H., & Rai, S. (2002, 5). The optimal class size for object-oriented software. *IEEE Transactions on Software Engineering*, 494-509. doi:10.1109/TSE.2002.1000452

Escobar, D., Daniel, D., Amarillo, R., Castro, E., Garcés, K., Parra, C., & Casallas, R. (2016). Towards the understanding and evolution of monolithic applications as microservices. (pp. 1-11). IEEE Xplore. doi:10.1109/CLEI.2016.7833410

Fowler M., L. J. (2014). Microservices. Retrieved from https://martinfowler.com/articles/microservices.html

Fowler, M. (2014, 1 15). BoundedContext. Retrieved from https://martinfowler.com/bliki/BoundedContext.html

Fowler, M. (2015, 6 1). Microservice Trade-Offs. Retrieved from https://martinfowler.com/articles/microservice-trade-offs.html

Furda, A., Fidge, C., Zimmermann, O., Kelly, W., & Barros, A. (2018). Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency. 63-72. doi:10.1109/MS.2017.440134612

Hatton. (1997). Reexamining the fault density component size connection. 89-97. IEEE Xplore. doi:10.1109/52.582978

Hevner, A., March, S., Park, J., & Ram, S. (2004, 3). *Design science in information systems research.* MIS Quarterly Vol. 28 No. 1 pp 75-106. Retrieved from https://www.researchgate.net/publication/201168946_Design_Science_in_Information_Systems_Research/citations

Jamshidi, P., Pahl, C., Mendonça, N., Lewis, J., & Tilkov, S. (2018). Microservices: The Journey So Far and Challenges Ahead. *IEEE Software*, 24-35. doi:10.1109/MS.2018.2141039

Kamimura,, M., Yano, K., Hatano, T., & Matsuo, A. (2018). Extracting Candidates of Microservices from Monolithic Application Code. (pp. 571-580). IEEE Xplore. doi:10.1109/APSEC.2018.00072

Kazanavičius, J., & Mažeika, D. (2019). Migrating Legacy Software to Microservices Architecture. (pp. 1-5). IEEE Xplore. doi:10.1109/eStream.2019.8732170

KitchenHam, B., & Charters, S. (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering.* Durham: Department of Computer Science University of Durham. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.117.471

Kizior, R. J., Carr, D., & Halpern, P. (2000). Does COBOL Have a Future? Retrieved from https://web.archive.org/web/20160817115437/http://proc.isecon.org/2000/126/ISECON.2000.Kizior.pdf

Knoche, H. (2016). Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices. *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16* (pp. 121-124). ACM Press. doi:10.1145/2851553.2892039

Knoche, H., & Hasselbring, W. (2018). Using Microservices for Legacy Software Modernization. 44-49. doi:10.1109/MS.2018.2141035

Langer, A. (2016). *Guide to software development Designing and Managing the Life Cycle, second edition.* Springer. doi:DOI 10.1007/978-1-4471-6799-0

Larrucea, X., Santamaria, I., Colomo-Palacios, R., & Ebert, C. (2018). Microservices. *IEEE Software, 35*(3), 96-100. doi:10.1109/MS.2018.2141030

Lehman, M., Perry, D., & Rami, J. (1998). *Implications of evolution metrics on software maintenance".* IEEE Computer. Retrieved from http://maveric0.uwaterloo.ca/~migod/846/papers/lehmanPerry-metrics.pdf

Lenarduzzi, V., Lomio, F., Saarimäki, N., & Taibi, D. (2020). Does Migrate a Monolithic System to Microservices Decrease the Technical Debt? *Journal of Systems and Software, 169*. Retrieved from http://arxiv.org/abs/1902.06282

Lithicum, D. S. (2016). *Practical Use of Microservices in Moving Workloads to the Cloud.* IEEE. doi:10.1109/MCC.2016.114

Liu, K. (1998). *Report on the First SEBPC Workshop on Legacy Systems.* Retrieved from www.dur.ac.uk/CSM/SABA/legacy-wksp1/report.html

MacKenzie, M. L. (2006). *Reference model for service oriented architecture 1.0. OASIS Standard.*

Mahanta, P., Chouta, S., Debruyne, C., Panetto, H., Guédria, W., Bollen, P., . . . Meersman, R. (2020). Translating a Legacy Stack to Microservices Using a Modernization Facade with Performance Optimization for Container Deployments. (pp. 143-154). Springer International Publishing. doi:10.1007/978-3-030-40907-4_14

Maisto, S., Di Martino, B., Nacchia, S., Barolli, L., Hellinckx, P., & Natwichai, J. (2020). From Monolith to Cloud Architecture Using Semi-automated Microservices Modernization. *Advances on P2P, Parallel, Grid, Cloud and Internet Computing* (pp. 638-647). Springer International Publishing. doi:10.1007/978-3-030-33509-0_60

Malinova, A. (2010). Approaches and techniques for legacy software modernization. Retrieved from https://www.researchgate.net/profile/Anna_Malinova/publication/267181092_Approaches_and_techniques_for_legacy_software_modernization/links/5739e7c508ae298602e36682.pdf

Markus, M. L., Majchrzak, A., & Gasser, L. (2002). *A design theory for systems that support emergent knowledge processes.* MIS Quarterly, 26(3), 179-212. Retrieved from https://ezproxy.cc.lut.fi/docview/218120404?accountid=27292

Maurizio Gabbrielli, S. G. (2016). *Self-Reconfiguring Microservices.* Cham: Springer International Publishing. doi:DOI: 10.1007/978-3-319-30734-3_14

Mayer, B., & Weinreich, R. (2018). An Approach to Extract the Architecture of Microservice-Based Software Systems. (pp. 21-30). IEEE Xplore. doi:10.1109/SOSE.2018.00012

Mazzara, M. (2006). Towards Abstractions for Web Services Composition. Retrieved from https://www.semanticscholar.org/paper/Towards-Abstractions-for-Web-Services-Composition-Mazzara/fb17f3ad38f2355d05b4b4f560478fb0f5719b2c#citing-papers

Mazzara, M., Dragoni, N., Bucchiarone, A., Giaretta, A., Larsen, S., & Dustdar, S. (2018). Microservices: Migration of a Mission Critical System. *IEEE Transactions on Services Computing*, 1-6. doi:10.1109/TSC.2018.2889087

Mishra, M., Kunde, S., & Nambiar, M. (2018). Cracking the monolith: challenges in data transitioning to cloud native architectures. (pp. 1–4). New York: Association for Computing Machinery. doi:10.1145/3241403.3241440

Mujtaba, S. P. (2008). *Software product line variability: A systematic mapping study.*

Munindar P. Singh, M. N. (2004). *SERVICE-ORIENTED COMPUTING, Semantics, Processes, Agents.* John Wiley & Sons, Ltd. Retrieved from http://onlinelibrary.wiley.com/doi/abs/10.1002/0470091509.fmatter

Nicola Dragoni, S. G. (2017). *Microservices: Yesterday, Today, and Tomorrow.* Spring International Publishing. doi:DOI: 10.1007/978-3-319-67425-4_12

Nunamaker, J., Chen, M., & Purdin, T. D. (1990). *Systems Development in Information Systems.* Retrieved from http://web.a.ebscohost.com.ezproxy.cc.lut.fi/ehost/pdfviewer/pdfviewer?vid=0&sid=4edd8b99-bc20-4717-bd88-747cef55a404%40sdc-v-sessmgr01

O'Brien, L., Bass, L., & Merson, P. (2005, 9). Quality Attributes and Service-Oriented Architectures. Retrieved from https://www.researchgate.net/profile/Liam_Obrien5/publication/200086155_Quality_Attributes_and_Service-Oriented_Architectures/links/5513d6e70cf283ee08349291/Quality-Attributes-and-Service-Oriented-Architectures.pdf

OLE LEHRMANN MADSEN, B. M.-P. (1993). *Object-oriented Programming in The Beta Programming Language.*

Pahl, C., & Jamshidi, P. (2016). Microservices: A Systematic Mapping Study:. *6th International Conference on Cloud Computing and Services Science* (pp. 137-146). Rome, Italy: SCITEPRESS - Science and and Technology Publications. doi:10.5220/0005785501370146

Parnas, D. (1994). *Software Aging.* IEEE. Retrieved from https://www.cs.drexel.edu/~yfcai/CS451/RequiredReadings/SoftwareAging.pdf

Peffers, K., Tuunanen, T., Gengler, C. E., Rossi, M., & Hui, W. (2006). *THE DESIGN SCIENCE RESEARCH PROCESS: A MODEL FOR PRODUCING AND PRESENTING INFORMATION SYSTEMS RESEARCH.* University of Jyväskylä.

Retrieved from https://www.researchgate.net/publication/228650671_The_design_science_research_process_A_model_for_producing_and_presenting_information_systems_research

Petersen, K., Feldt, R., Mujtaba, S., & Mattsson, M. (2008). *Systematic Mapping Studies in Software Engineering.* School of Engineering, Blekinge Institute of Technology. Retrieved from https://www.scienceopen.com/hosted-document?doi=10.14236/ewic/EASE2008.8

Pour, G. (1998). Component-based software development approach: new opportunities and challenges. (pp. 376-383). IEEE Xplore. doi:10.1109/TOOLS.1998.711055

Pour, G. (1999a). Software Component Technologies: Javal3eans and ActiveX. *Proceedings of Technology of Object-Oriented Languages and systems*, 398 - 398.

Pour, G. (1999b). Enterprise JavaBeans, JavaBeans and XML expanding the possibilities for Web-based enterprise application development. (pp. 282-291). IEEE Xplore. doi:10.1109/TOOLS.1999.796495

Pressman, R. (2015). *Software Engineering A Practioners Approach.* New York, NY: McGraw-Hill Education. Retrieved from http://ce.sharif.edu/courses/98-99/2/ce474-2/resources/root/Roger%20S.%20Pressman_%20Bruce%20R.%20Maxin%20-%20Software%20Engineering_%20A%20Practitioner%E2%80%99s%20Approach-McGraw-Hill%20Education%20(2014).pdf

Ramachandran, M., Mahmood, Z., Megargel, A., Shankararaman, V., & Walker, D. (2020). Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example. Springer International Publishing. doi:DOI: 10.1007/978-3-030-33624-0_4

Raymond, E. (2003). The Art of Unix Programming. Retrieved from https://nakamotoinstitute.org/static/docs/taoup.pdf

Seacord, R., Plakosh, D., & Lewis, G. (2003). *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices.* Addison-Wesley Professional. Retrieved from

https://www.researchgate.net/publication/234822137_Modernizing_Legacy_Systems_Software_Technologies_Engineering_Process_and_Business_Practices

Sommerville, I. (2016). *Software Engineering Tenth Edition.* Pearson Education.

Tuli, S. (2018, 5 16). Microservices vs SOA: What's the Difference? - DZone Microservices. Retrieved from https://dzone.com/articles/microservices-vs-soa-whats-the-difference

Vaclac T. Rajlich, K. H. (2000). A Staged Model for the Software Life Cycle. IEEE Computer. doi:10.1109/2.869374

Vogels, W. (2006, 6 30). A Conversation with Werner Vogels. (J. GRAY, Interviewer) Retrieved from https://queue.acm.org/detail.cfm?id=1142065

W3C. (2004). Web Services Architecture. Retrieved from https://www.w3.org/TR/ws-arch/

Walls, J., Widmeyer, G., & El Sawy, O. (1992). *Building an information system design theory for vigilant EIS.* Retrieved from http://web.b.ebscohost.com.ezproxy.cc.lut.fi/ehost/pdfviewer/pdfviewer?vid=0&sid=8612f98c-fe09-4bc3-8088-fdab2fcc47fe%40sessionmgr101

Warren, I. (1998). The renaissance of legacy systems. London, Britain: Springer. Retrieved from https://www.researchgate.net/publication/265287408_The_Renaissance_of_Legacy_Systems

Wieringa, R. M. (2006). *Requirements engineering paper classification and evaluation criteria: a proposal and a discussion.* Computer Science. Retrieved from https://www.semanticscholar.org/paper/Requirements-engineering-paper-classification-and-a-Wieringa-Maiden/fd3fde428c91e084b329196ec69380d40e54e16b

Xia Cai, L. M.-F. (2000). Component-based software engineering: technologies, development frameworks, and quality assurance schemes. (pp. 372-379). IEEE. doi:10.1109/APSEC.2000.896722

YourDictionary. (2020). Legacy Software. Retrieved from https://www.yourdictionary.com/legacy-software