LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT
School of Engineering Science
Software Engineering and Digital Transformation

# DESIGN SCIENCE IN SOFTWARE ENGINEERING: SOFTWARE ARTEFACT DESIGN FOR A SAAS APPLICATION

Examiners: Associate Professor Sami Hyrynsalmi
Assistant Professor Antti Knutas

# TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT
School of Engineering Science
Software Engineering and Digital Transformation
Jussi Moilanen

**Ohjelmistosuunnittelu: Ohjelmisto artefaktin luominen SaaS-ohjelmistoon**
Diplomityö 2022

69 sivua, 9 kuva, 8 taulukko

| | |
|---|---|
| Työn tarkastajat: | Professori Sami Hyrynsalmi |
| | Apulaisprofessori Antti Knutas |
| Hakusanat: | ohjelmisto artefakti, mikropalvelut, monoliitti |
| Keywords: | design artefact, microservices, monolith |

Perinteiset monoliittiset ohjelmistot voivat olla toiminnallisesti riittäviä, mutta ne eivät sovellu laadukkaiden digitaalisten palveluiden kehittämiseen. Jatkaessaan digitaalista muutosta organisaatioiden tulisi omaksua tehokkaampia sovellusarkkitehtuurisia tyylejä, jotka tukevat uusien pilvipohjaisten palveluiden nopeaa toimitusta asiakkaille. Tässä pyrkimyksessä mikropalveluarkkitehtuuria pidetään tärkeänä edellyttäjänä. Mikropalveluissa ydinajatuksena on, että sovelluksia voidaan rakentaa itsenäisesti julkaistaviin kokonaisuuksiin, mikä mahdollistaa kehittämisen joustavuuden, sovelluksen paremman ymmärtämisen ja järjestelmän skaalautuvuuden. Tämä myös mahdollistaa järjestelmän hajautetun hallinnan, jonka avulla pienet itsenäiset tiimit voivat innovoida nopeammin ja parantaa uusien digitaalisten palveluiden käyttöönottoa. Organisaatiot kohtaavat haasteita, kuinka suunnitella uusia sovelluksia käyttämällä mikropalveluarkkitehtuuria ja kuinka siirtyä monoliitista pilvipohjaiseen mikropalveluarkkitehtuuriin.

Tässä tutkimuksessa tarkastellaan mikropalveluarkkitehtuurin lisäksi ohjelmistosuunnittelun eri vaiheita, haasteita ja kuinka liiketoiminnan vaatimukset voidaan

muuttaa tekniseksi suunnitelmaksi. Työn tarkoituksena oli luoda artefakti mallintamaan järjestelmää, minkä avulla voidaan automatisoida uusien integraatiokumppaneiden palveluiden lisääminen taloushallinto-ohjelmiston sisäänrakennettuun verkkokauppa-alustaan. Ongelman ratkaisemiseksi ja liiketoimintatarpeiden mahdollistamiseksi ehdotetaan, että nykyisen sisäänrakennetun kauppa-alustan kehitystä ei tulisi jatkaa perinteisellä tavalla, vaan uudelleen toteuttaa alusta mikropalveluarkkitehtuuria hyödyntäen.

# ABSTRACT

Lappeenranta-Lahti University of Technology LUT
School of Engineering Science
Software Engineering and Digital Transformation
Jussi Moilanen

**Design science in software engineering: Software artefact design for a SaaS-application**

Master's Thesis 2022

69 pages, 9 figure, 8 table
Examiners: Associate Professor Sami Hyrynsalmi
Assistant Professor Antti Knutas
Keywords: design artefact, microservices, monolith

Traditional monolithic systems may be functionally adequate but are unsuitable for the development of high-quality digital services. As businesses continue to invest in digital transformation initiatives, a more agile style of approach in application design that enables rapid delivery of new cloud services to clients should be explored. In this endeavour, microservices architecture is considered as a major facilitator. The primary principle behind this design pattern is that software may be produced more efficiently by constructing it as a collection of individual services. The upside of this method is that it allows for decentralized governance, which in turn allows small, autonomous teams to innovate more quickly, resulting in faster implementation of new digital services. Organizations are currently facing challenges in developing new applications with as microservices and transitioning from a monolithic to a cloud-based microservice architecture.

Aside from the microservice architecture, this research looks at the various stages of design science and how business objectives can be translated into a technical strategy. The objective of this project was to create a design artifact that can be used to build a system to automate the addition of new integration partner services to the built-in store platform of the financial

management software. To address the issue and meet business requirements, it is proposed that the current monolithic store system should be re-implemented utilizing the micro-service architecture rather than continuing in the traditional manner.

# TABLE OF CONTENTS

# LIST OF SYMBOLS AND ABBREVIATIONS

| | |
|---|---|
| API | Application Programming Interface |
| BPMN | Business Process Model and Notation |
| CI/CD | Continuous integration and delivery |
| DSR | The Design Science Research |
| DSRM | The Design Science Research Methodology |
| HTTP | Hypertext Transfer Protocol |
| IS | Information system |
| UI | User Interface |

# 1 INTRODUCTION

Software systems are becoming increasingly more complex, and the business needs for the software are becoming more complicated (Sommerville, 2016). There is an increasing demand for flexibility and quick adaptation, and at the same time expenditures of running, maintaining, and evolving software are increasingly expensive. In the age of digital transformation, the modernization of software systems is a critical technique for businesses to gain a competitive advantage over other (Khadka et al., 2014).

Software design can be defined in many ways. According to Kapor (1990), software design is an activity that defines the user experience with a piece of software, and it has nothing to do with how the code works, or how big or small the implementation is. Instead, it is up to the software designer to unambiguously define the overall user experience, and deal with the trade-offs. Also, it has been said that the designer stands on one foot in technology, and on one foot in the domain of human requirements, and these two worlds are not easily comparable.

Sommerville (2016), on the other hand, defines that software design is a description of the structure of the program that will be implemented, the data models and structures that will be used by the system, the interfaces between system components, and, in some situations, the algorithms that will be used. Designers do not create a final design all at once but develop it in stages. They add detail as they progress through the design process, with continual backtracking to revise previous concepts. In general, design science is a process where a software product specification is created with the intention of achieving a specific goal. A design challenge is one that involves redesigning an artefact so that it can better contribute to the attainment of a goal. An instrument design goal is the problem of developing an instrument to help answer a knowledge question, whereas an artefact design goal is the problem of making an item to improve a problem setting (Wieringa, 2014).

Hevner and Chatterjee (2010) emphasize that revolutionary advancements in software hardware, information, networking, and human interface technologies will necessitate completely novel approaches to the design, development, and evaluation of software-intensive systems. In design science, human creativity is emphasized in the design and

creation of artefacts in the context of solving real-world business challenges. The core premise is that when creating an artefact, understanding is gained of the design challenge and its solution, which is also vital for organizations in order to minimize knowledge evaporation (Kleebaum, 2019).

Businesses with monolithic codebases that continue to create new functionality will produce unstructured source code, making maintenance more challenging (Lapuz et al., 2021). At the same time, a lot of software expenses are spent on maintenance, and many issues may be traced back to a lack of understanding of the current system, which can cause issues in later development cycles (Reussner et al., 2019). Legacy monolithic systems may be functionally adequate, but they are not well adapted to the development of high-quality digital services. Simultaneously, as businesses invest in digital transformation, they should look at better architectures and design techniques that allow for the rapid delivery of digital services to the cloud. In this endeavour, microservices is considered as a major facilitator. The primary principle behind this design pattern is that software may be produced more rapidly by constructing it as a collection of self-governing and modular services. The primary advantage is that microservices allows for decentralized governance, which enables small, autonomous teams to innovate more quickly, resulting in faster implementation of new digital services.

Businesses are today confronted with two issues: how to develop new software with microservice design pattern and how to shift from a monolithic software to the cloud-hosted microservices (Megargel et al., 2020). In this thesis microservices are studied further to determine whether they can assist with the design challenge and overall architectural approach that the case company's issue has brought up. This thesis will also look into different stages of software design and the challenges of translating business requirements into a design product.

The premise for the thesis is the case company’s financial management software, and the problem with the built-in online store system. The purpose of this internal marketplace is to integrate other partner services that complement the core product. The problem has arisen as a result of the increasing demand for new partner services to be added to the internal store platform, which the current system is unable to handle at the required speed due to the

monolithic state of the system, and that the original design was not to address these business needs.

As a result, with the existing design, adding a new partner service necessitates a significant amount of development effort in order to meet the requirements. It also takes a long time to understand the system owing to its size, complexity, and monolithic nature, and it takes a lot of time for the development team to make changes to the present system. Furthermore, due to the monolith, modifications to the system are frequently massive and time intensive, and updates to the monolith occur only weekly, preventing rapid changes from taking place. Solutions to the problem have been sought in the literature, and according to the design principles, an artefact is created to support the thesis' solution. The main goal of the work is to produce an answer to the case company problem, as well as to increase the knowledge of the existing system by working as a documentation for the reasoning of the proposed solution.

## 1.1 Scope and delimitations of the thesis

The purpose and scope of this thesis is to leverage the theory from recent scientific literature and methodologies to create a design model i.e., a design artefact, for adding a new partner service to the internal store more efficiently. In this context, the efficiency goal refers to an automated system that permits services to be created and transformed at a faster rate than the current system. Adding a new service in the current system requires two to four weeks of development time from the start phase to production. The new system should allow the addition of a new partner service within one day. Enabling better design of the new store system, the current process needs to be thoroughly explored and its weaknesses and limitations identified. The new design model must also consider the possible limitations of current architecture and strive to find a solution that satisfies the requirements from both a technical and a business point of view.

In this work, the software implementation itself is not produced, but the resulting design artefact can subsequently be utilized to develop software of the new store system. Also, the constraints of the current system are drawn up, the common parameters of the individual store service are sought, and a preliminary higher-level design model plan is created. As

there is a lot of complexity in the system, a conceptual attempt has been made to atomize the problem, so that it is easier to understand the entire system.

The purpose is also to identify the constraints of the current code base and which parts should be refactored or replaced entirely. A strategy for the scope of work, as well as an estimate of the amount of development time and alternative possibilities for a new store system, are also considered, although they are not implicitly included in the work objectives.

## 1.2 Design goals of the thesis

Design goal and the sub-design requirements of the thesis are tabulated in **Table 1**, where the objective of each requirement is opened in short. The main objective for the thesis and thus the main design goal is to provide a solution to the case company's challenge with the increasing demand to add new integration partner services to the built-in store system. The sub-design requirements have been developed with the intention of focusing on discovering a theory that will help to support the main goal and create coherence with the practice of design study.

The first sub-design requirement is to find out how the current system works and what are the challenges and limitations of the current implementation related to the rigidity of adding a new partner service store system. The second sub-design requirement aims to identify the main functionalities for the design solution that takes current measures and incorporates the new technical and business requirements. The objective of this question is to gain a better understanding of the system's key parameters and functions, and it also indicates what should be prioritized and which components are necessary for the improved system. This would also aid in understanding the new system's scope and workload. The third sub-design requirement’s purpose is to bring up and provide information on the benefits and drawbacks of microservices in comparison to monolithic architecture. This question justifies the necessity to alter the entire system's architectural structure.

**Table 1.** Design goals and objectives

| **Design Goal** | **Objectives** |
|---|---|
| The system should be able to process the delivery of the new partner's integration product to the internal store platform within one day. | Create or improve the current process and software system so that the addition of a new integration partner product can be completed within one day. |
| **Sub-design requirements** | |
| Find the design weaknesses of the current system | To produce relevant information on existing system and justify actions to create a new system that would generate more value. |
| Define the main features for the design artefact that would meet the technical and business requirements. | To have a better understanding of the system's key parameters and functionalities. It establishes what should be prioritized and which pieces are essential for the design artefact to function properly. |
| Examine the benefits and drawbacks of microservice architecture design. | Provides information on the benefits and possible challenges of micro-services over the monolithic solutions. |

## 1.3 Structure of the thesis

The first section of this master's thesis emphasizes the necessity of the thesis, illustrates the goals and limitations of the design study, and briefly explains the case company's problem. The second section of the thesis discusses the microservice architecture and compares monolithic and microservice application architectures. The third section focuses on the research methodologies that will be used during the study. It goes over design science research framework, different software modelling processes, and the transition from monolithic to cloud-based microservices. The fourth section explores the background of the

case company and delves deeper into the case company's problem. The design artefact created during the thesis is validated with literature and described how it was formed during the work. Sections five and six of the work discuss the results that answers to the design goals, evaluates the design artefact, and considers other relevant subjects that have emerged during the thesis based on the topic. The final chapter concludes and highlights the most important observations of what was presented in the work and summarizes the results of the work.

# 2 LITERATURE REVIEW

This chapter describes microservice architecture and its benefits and drawbacks. The purpose of this chapter is to lay the groundwork for comparing monolithic versus microservice application architecture approaches. The goal is to provide support for the proposed design artifact, which was constructed to provide a solution to the problem of the case company.

## 2.1 What are microservices

The high expense of maintaining outdated systems motivates organizations to transition them to using more agile design patterns. Modular architectural styles like microservices have become popular for modernizing monolithic legacy systems. (Assunção et al., 2022) The basic principle of this design is to create software applications as collections of independent, tiny, modular services in order to develop them more rapidly. This design has the major benefit of empowering decentralized governance, which allows small, independent teams to develop more quickly, reducing time to deliver new application to the customers. (Rajasekharaiah, 2021; Megargel et al., 2020) Microservices design enables agility, shorter development and deployment cycles, greater scalability to the defined functionality, and the possibility to create solutions using a mix of different technologies. Often the purpose of using microservices architecture is to deconstruct a monolithic software into a collection of autonomous services that communicate with one another using scalable messaging services or APIs. (Ramachandran et al., 2020; Megargel et al., 2020).

Microservices are a cloud-native design solution, that attempts to decompose software applications as a set of small modular services. Each application may be deployed individually possibly on a different cloud platform and with a different technical stack. It can execute in its own process while interacting via means of lightweight mechanisms such as Representational State Transfer (REST). In this context, an application is a business capability that may be developed by a small team and can use a different programming languages and ways of storing data. (Balalaie et al., 2016).

Containerization is also possible with microservices due to the small size of the deployment, which may be performed with the help of Docker or similar technologies. A single

microservice can then be deployed into its own virtual machine, where it can run as self-contained lightweight containers that can be elastically scaled out across a variety of host systems. (Megargel et al., 2020). The fundamental idea of an microservice-based architecture is illustrated in the **Figure 1**.

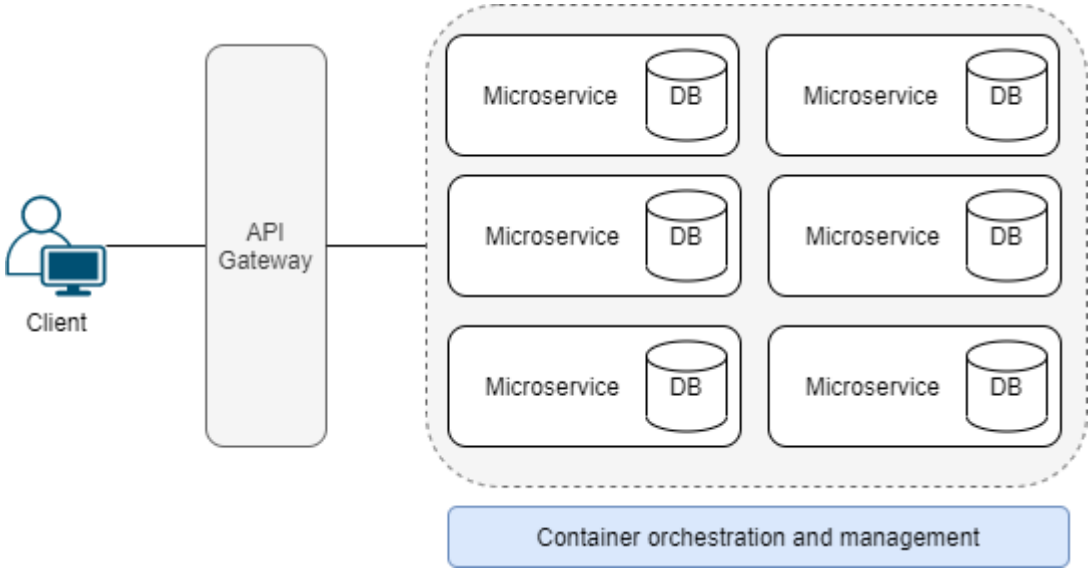


**Figure 1**. Fundamental idea of microservice architecture.

## 2.2 Monolithic Applications

Legacy monolithic systems may be operationally acceptable in traditional software systems, but they are unsuitable for developing new innovative software applications. Traditional monolithic architectures and development methodologies continue to be a stumbling barrier in achieving digital transformation. A single monolith is generally made up of multiple business functions that are all delivered in a single release. (Ramachandran et al., 2020) As monolithic programs can become enormous and complex, they can over time result as a technological debt which leads to unmaintainable and obscure architecture. A single developer or architect requires an ever-increasing quantity of information and time to fully comprehend all components and their interfaces. As a result, maintaining the monolith and adapting to newer and better technology is tough. (Fritsch et al., 2019)

Usually there are three different architecture levels that is shared by all types of monolithic applications, the UI layer, the layer for business logic, and database. A monolith's functions

are typically built in the same programming language, and they must interact with one another with native method calls and are thus tightly coupled (Megargel et al., 2020). In the **Figure 2** is depicted a simplified representation of these three different layers.

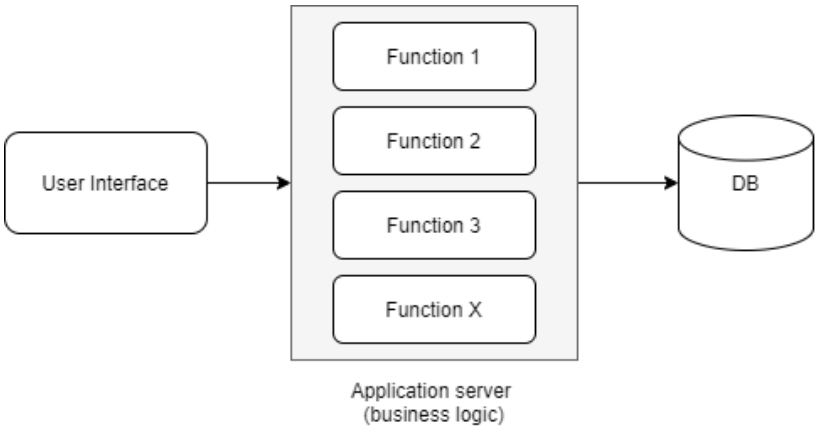


**Figure 2.** Layers of a monolithic application. (Megargel et al., 2020)

A monolithic application is a type of traditional application architecture in which modularity is not used as a design idea. In contrast, microservices often encapsulate a single business function that can be scaled and delivered independently. (Megargel et al., 2020).

## 2.3 Comparison Between Monolith and Microservices

As briefly mentioned earlier, the microservice architecture supports maintainability, scalability, and availability better over monolith. By adapting microservices event-driven designs could also be introduced. The event-based architectures has been used to reduce coherence between existing architecture components. This style alters a component's reliance on another's service to the events. As a result, components in the target architecture may be independently updated, deployed, and recovered. Because the planned architecture follows the loosely coupled concept, each component in the container may be self-contained, allowing it to be deployed and operated separately. (Chondamrongkul et al., 2020) On the other hand, since legacy applications integrate several business activities or components into a single deployable unit, developers can easily violate the modularity principle while still producing fully functional software. As a result, as these legacy monolithic systems age, they become more difficult to comprehend and manage due to the increased complexity in comprehending the application code and projecting the impact of modification. (Mathai et al., 2021)

Agile development teams have also had success with microservices since they can work on small items that can be supplied without having to compile the monolith. One of the most frequently claimed advantages of microservices is reduced team complexity, size, and overall improved team structure. Smaller teams translates to lower communication overhead, resulting in increased productivity and focus with the team's actual domain challenge and service for which it is liable. (Mazlami et al., 2017) In monolithic applications software is built usually with the same programming language, and because software coding practices are changing fast, the whole codebase could become deprecated in a few years. Microservices, on the other hand, may be built using a variety of technologies, thus this would not be an issue.

In addition, there are popular tools like Docker to handle containerization of microservices, and container orchestration systems like Kubernetes for automating and scaling containers, which are supported by mainstream cloud providers. As microservices could be built in a decentralized manner, data access could be improved, as during the failure of a particular node the data could still be accessed by other available nodes. In short, microservices provide a more agile solution for producing higher quality software at a faster pace. **Table 2** further outlines the main differences between monolithic and microservices architectures.

**Table 2.** Comparison of features between Microservices and Monolith systems. (Megargel et al., 2020)

| **Feature** | **Monolith** | **Microservice** |
|---|---|---|
| Technology stack | Commitment to the original technological stack and foundation.<br><br>All functions are written in a single programming language. | Can be built different technology stack, depending on its intended use or developer choice. |
| Change Management | Any minor change necessitates retesting of the entire monolith | Small and easy to test, and they have their own release cadences. |

| | | |
|---|---|---|
| | and testing processes might be complex and time-consuming. | |
| Deployment | The deployment file is big, it takes a long time to boot up, and it may cause downtime. | Can be deployed independently, and deployment processes could be automated. |
| Scalability | A monolith's functions cannot be scaled individually.<br><br>The entire monolith must be horizontally scaled. | Containers allow each microservice to be scaled elastically.<br><br>Monitoring and management tools are required for containers. |

As businesses begin to integrate cloud computing with their systems, it is critical that they use suitable architectures and development processes to get the full benefits of the cloud services. The needs of digital services are not met by a simple transfer strategy, but the transfer of monolithic services poses several challenges. While monolithic apps are easy to create and manage, they are inflexible when it comes to adapting to cloud practices. (Ramachandran et al., 2020)

## 2.4 Challenges of Serverless Approach

The intrinsic structure of microservices architecture, which consists of small, autonomous, and components that are segregated from one another and communicate using normal web protocols, allows for scalability, and flexibility, as well as quick system evolution. Each of these qualities, however, introduces obstacles that are not present in monoliths. The creation of numerous tiny services disperses the core business logic across multiple application and teams, necessitating greater coordination and monitoring. (Rajasekharaiah, 2021) The main topics of challenges with microservices are summarized in **Table 3**.

When it comes to cloud-based architecture, understanding distributed systems becomes critical. The basic premise is that a distributed and partitioned datastore cannot be both highly accessible and consistent at the same time. Another issue is the shift in dependent microservice response datatypes when one microservice relies on another

microservice that returns specified resources. In general, consistency becomes more challenging with data that is geographically spread within the same domain. As software systems continue to evolve parallel, issues arise, such as making it impossible to provide newer versions if users are maintained tightly connected with service specifications. Vice versa, if the users are kept too loose, systems are burdened with being backward compatible and slowing releases. (Preston-Werner, 2021) Versioning is the solution to this problem in microservices, but it can lead to code bloat and system complexity when it comes to maintaining and testing against all definitions (Rajasekharaiah, 2021).

The division of a system into separate portions that communicate via Hypertext Transfer Protocol (HTTP) protocols in the absence of distributed transactions necessitates the planning and mitigation of partial failures in operations. The consistent state of the system is undetectable due to the distribution of domain data, which is separated amongst subdomains and further partitioned for scalability. It also makes obtaining composite data for reporting and analytics more difficult, necessitating the development of new methodologies and solutions. Rapid evolution, while far preferable to monoliths, is not without its difficulties. (Rajasekharaiah, 2021; Megargel et al., 2020).

**Table 3.** Challenges with microservices (Rajasekharaiah, 2021).

| **Challenge** | **Explanation** |
|---|---|
| Dispersed business logic | Decoupling a monolith into numerous microservices improves development, team organization, and deployment agility, but it also disperses the operational logic and execution of complicated features over multiple apps. |
| Lack of distributed transactions | Attempting to keep data transactional integrity causes scalability problems. It is difficult to ensure consistency across several microservices engaged while avoiding monolithic patterns in a business transaction. |

| | |
|---|---|
| Overall dynamic state is inconsistent | There are not any dispersed transactions, and the total state is not immediately consistent. It is impossible to achieve extremely accessible and consistent results. |
| Compiling composite data is difficult | In a microservices design, combining data for total system analytics is difficult. |
| Debugging failures and faults is difficult | Determining the underlying cause of the problem can be challenging, owing to deep microservice hierarchies, scattered business logic, and the difficulty to ascertain the actual state of the system. |
| The difficulty of evolving | Service definitions grow stale as autonomous systems with independent life cycles hinder agility. |

As the number of microservices deployed grows, so does the complexity of a microservices-based architecture, resulting to issues in managing them. Monitoring and management tools must be developed and incorporated into the system, and they are required to track the microservice status and restart any that have stopped working. (Rajasekharaiah, 2021; Megargel et al., 2020).

There are also problems as a result of the intrinsic characteristics of microservices architecture. When a breakdown occurs, discovering the fundamental cause of the problem can be difficult due to deep hierarchies of microservices, distributed business logic, and the difficulties in determining the system's actual state. It also can be extremely hard for a single engineer to have a complete picture of the overall operation of all the microservices in a large system. The lack of a comprehensive perspective makes it difficult for engineers to pinpoint the source of the problem, which is exacerbated by the constant evolution of microservices across all domains in the system. (Rajasekharaiah, 2021)

Although the serverless deployment approach has gained popularity in recent years, it does not allow the development team to customize the deployment architecture. Also, because serverless functions can only be implemented in one of the serverless platform's supported

languages, some cloud service functions, such as AWS lambda, might cause vendor locking concerns. Despite the disadvantages, we can see that the current infrastructure is continuously being improved. (Karabey Aksakalli et al., 2021)

# 3 RESEARCH METHODS

This section examines all research methods and practices that have contributed to the thesis design solution. Firstly, theory of the design science is introduced, which has been used to conduct the research. After that, modelling practices and standards have been introduced, and lastly the phases on how migration from monolith to microservices could be done is elaborated on.

## 3.1 Design Science Research Framework

Gregor and Jones (2007) argue that good anatomy of information systems (IS) design theory could be built on six core and two additional components, described in **Table 4**, and that design theory should include as a minimum the core components. These six main elements can be used to achieve an idea of how an artefact should be constructed.

**Table 4**. Design theory components (Gregor and Jones, 2007).

| **Design theory components** | **Description** |
|---|---|
| 1. Purpose and scope | Goal, purpose, and the set of meta-requirements for artefact to which the theory applies. |
| 2. Constructs | Representations of the entities of interest in the theory. |
| 3. Principle of form and function | The abstract system design or architecture that forms a description of an IT artefact. The design product is seen in the properties, functions, or attributes that the product has when constructed. |
| 4. Artefact mutability | Expected changes in the artefact theory; what is the degree of change that theory covers. |
| 5. Testable propositions | Experiments and predictions on the outcomes of the to-be-built system. There is a need to assess whether the meta-design principles satisfy the meta criteria. |

| | |
|---|---|
| 6. Justificatory knowledge | The underlying theory from scientific, social, or design sciences that serves as the design's foundation and explanation. Also known as kernel theories, this principle asserts that design ideas should not be produced in the absence of theoretical backing. |
| 7. Principles of implementation | Description of the processes of implementing the theory in different contexts. |
| 8. Expository instantiation | An implementation of the designed artefact that can help to represent the theory as an expository device. |

In information systems, design expertise is very crucial for both research and practice. To date, little attention has been dedicated to the challenge of articulating design theory in order for it to be communicated, justified, and evolved. The underlining of design principles, however, helps the construction of mutable artefacts where complexity arises with the increase of human interactions. (Gregor and Jones, 2007)

Design science research (DSR) studies relevant problems in the real-world examples of application domains. Solutions to research problems need to be empirically explored with the people of organizations and other stakeholders to fit them with a particular technology. Often, analyzing the business environment and managing special needs will determine the starting point for a DSR project. However, there are also situations where needs have already been explored and can be exploited from a previous study. (Hevner and Chatterjee, 2010)

DSR analyzes the academic knowledge base in terms of the extent to which design information is available to solve a problem that is already of interest. Such knowledge may be in the form of theories, frameworks, instruments, or design products, such as software architectural structures, models, methods, or instances. In **Figure 3,** a framework to describe DSR is presented. (Hevner et al., 2004)

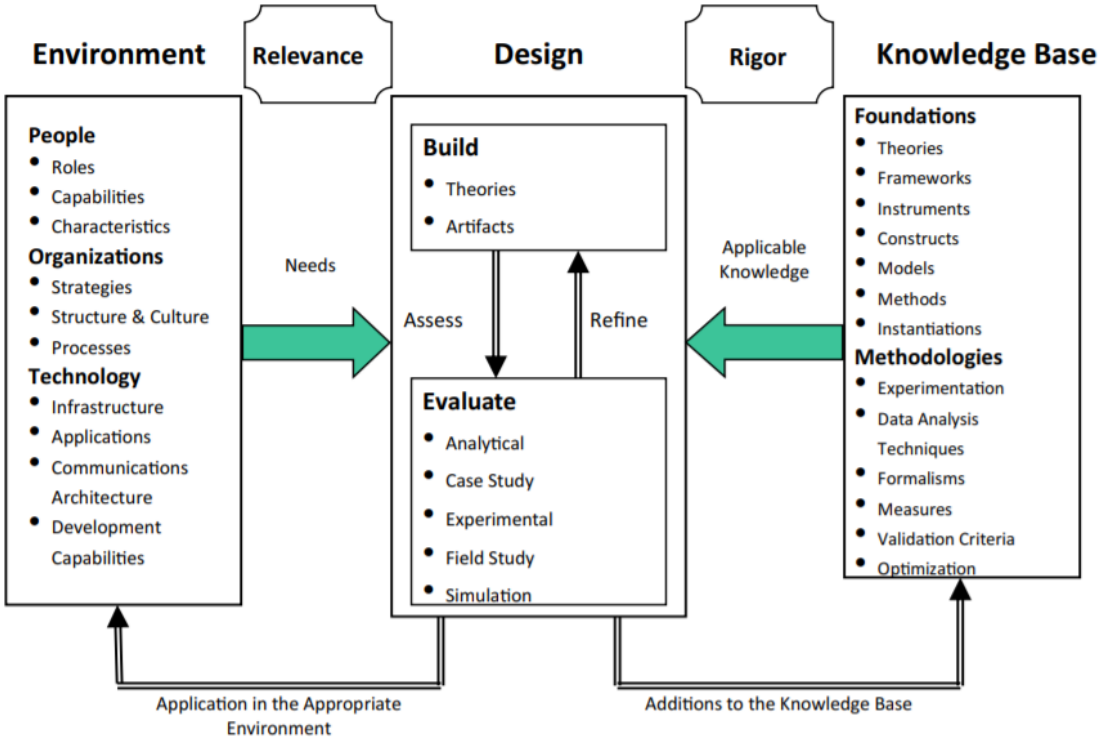


**Figure 3.** DSR framework (Hevner et al., 2004).

The environment defines the problem to which it relates, which stakeholders are interested in the topic, and who is affected by that problem (Brocke et al., 2020). It comprises the objectives, responsibilities, difficulties, and opportunities that constitute needs as viewed by the stakeholders of the organization. Organizational strategy, structure, culture, and existing work methods are used to investigate and evaluate needs. They are positioned in relation to the current technical infrastructure, applications, communication structures, and development capabilities. These factors, taken together, define the researcher's opinion of the study task. The relevance of research is ensured by designing research operations to respond to genuine stakeholder demands. (Hevner et al., 2004)

The Design Science Research Methodology (DSRM) is commonly proposed and accepted framework for successfully carrying DS research. It also helps in recognizing DS research and its objectives and should help researchers present research with reference to commonly understood research rather than justifying research on a case-by-case basis with each new paper. Peffers et al (2007) proposes six steps that researchers can use as a mental model to conduct a design study efficiently.

*Activity 1*. Problem identification and motivation.

This action defines the research problem and justifies the value of the solution. With justification of the value of the solution, it motivates the researcher and its audience for pursuing to find a solution, and it also helps the audience to accept the results and understand the researcher's reasoning associated with the certain problem.

*Activity 2.* Define the objectives for a solution.

From the problem definition and knowledge of what is achievable and doable, infer the goals of a solution. The objectives can be quantitative, for instance terms in which a desirable solution would be better than existing ones or qualitative in which a desirable solution would be better than current ones. This necessitates knowledge of the existing situation as well as contemporary solutions. (Hevner and Chatterjee, 2010; Brocke et al., 2020).

*Activity 3.* Planning and development.

The creation of the artefact. A design research artefact is any created object that incorporates a research contribution into the design. The artefact's desired functionality and architecture are determined throughout design and development. Knowledge of theory that can be applied to a solution is one of the resources required to move from objectives to design and development. (Hevner and Chatterjee, 2010).

*Activity 4.* Demonstration.

Demonstrating how the artefact can be used to solve one or more problems. This could include using it in experiments, simulations, case studies, proof, or any other relevant activities. Effective understanding of how to use the artefact to address the problem is one of the resources required for the demonstration. (Hevner and Chatterjee, 2010).

*Activity 5.* Evaluation.

The evaluation determines how effectively the artifact helps to a solution to a problem. This activity comprises comparing the aims of a solution to the actual observable consequences of the artefact's use in the context. Depending on the nature of the problem and the artifact, evaluation can take various forms. At the end of this activity, the researchers can choose whether to return to step three to improve the efficacy of the artifact, or to go to the last activity communication and leave further improvement to future efforts. (Hevner and Chatterjee, 2010).

*Activity 6.* Communication.

When applicable, informing researchers and other important audiences, like practicing professionals, about the problem and its importance, the artefact, its utility and originality, and its effectiveness. A common structure for empirical research papers is the nominal structure of an empirical research process, which includes problem definition, literature review, hypothesis development, data collection, analysis, results, discussion, and conclusion. Researchers may use the structure of this process to structure the paper in scholarly research publications. Communication necessitates familiarity with the disciplinary culture. (Hevner and Chatterjee, 2010)

## 3.2 IS Artefact

The DSR is centered on the creation of a new artefact. It is especially well-suited for research on the process assessment discipline, as it is practice-based, since DSR should not only attempt to understand how the world is, but also how to change it. An artefact in IS refers to a system consisting of subsystems that are, a technology artefact, an information artefact, and a social artefact. The IS artefact is more than a sum of its three subsystems, where it does not predominate in considerations of design, and where IS itself can be created by people. (Lee et al., 2015). Algorithms, logic programs, and formal systems are only a few of the artefacts in the IT and information systems field, which also includes software architectures, information models, and design standards, as well as demonstrations, prototypes, and production systems. (Johannesson and Perjons, 2014).

Most early IT artefacts were created for military and business purposes. However, in recent years, some of the most inventive IT artefacts have been built for everyday activities like staying connected with friends, sharing, and organizing images, and playing games. In some manner, people, practices, issues, and artefacts are all interrelated. People participate in behaviors in which they perceive issues that can be solved with artefacts. As a result, artefacts do not live in a vacuum; they are always part of a bigger context. (Johannesson and Perjons, 2014). In general, the idea of an artefact is used to refer to a thing that can be transformed into an instantiation, process, or a software. Many IT artefacts have some degree of abstraction but can later converted to material existence (Gregor et al., 2013).

Several ways to doing design science research have been offered, all of which imply a process that consists of two high-level activities: construct and assess. **Figure 4** shows that evaluation activities take place after an artefact is built, which is also a common assumption in other DSR processes (Sonnenberg, et al., 2012).

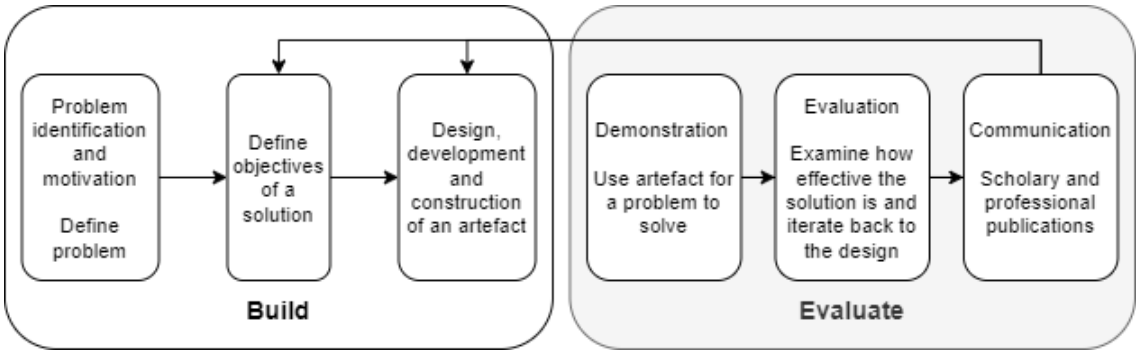


**Figure 4.** Creation and evaluation process of an artefact (Sonnenberg et al., 2012).

As a response to focusing more on the build activities over evaluation activities, a DSR approach proposed by Sein et al. (2011) has been developed that recommends building and evaluation processes concurrently to instantly reflect progress and to drive artefact modifications early in the design phase. The simultaneous assessment takes into consideration the fact that artefacts arise as a result of interaction with the organizational context as well as design interventions through reflection and learning activities.

## 3.3 IS Research Disciplines

IS research approaches can be categorized according to research levels. These levels define how research should be done and what are the goals that should be obtained. In the **Figure 5** these three different levels of IS research disciplines are presented. Prescriptive level is defined as simply prescriptions based on the practical consequences of descriptive research, and complex artefacts are not recognized as research outputs. The conceptual level is interested in what things are out there, descriptive research in how things are out there, and prescriptive research in how things may be out there and how one might effectively fulfil specified objectives. (Hevner and Chatterjee, 2010; Gregor and Jones, 2007)

As DSR outputs, the prescriptive level comprises both suggestions and artefacts. At the conceptual level, theories for analyzing and describing exist. Theories for explaining and forecasting are empirical regularities, whereas theories for explaining and predicting are descriptive theories. The prescriptive level is represented by theories for design and conduct. (Hevner and Chatterjee, 2010)

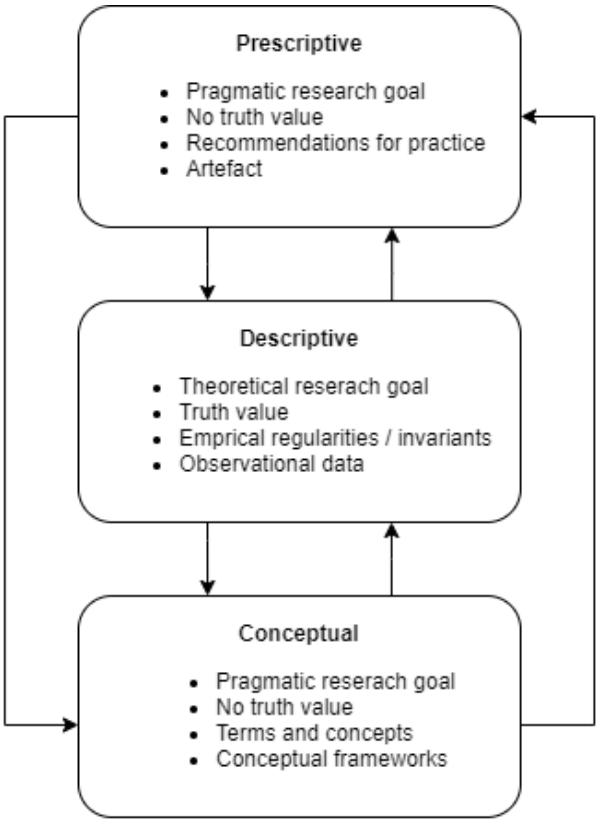


**Figure 5**. IS research levels. (Hevner and Chatterjee, 2010)

The different levels are linked, and each level supplies what will be investigated at a higher degree of research. As objects, the conceptual level provides a model of the research territory and its vocabulary, as well as conceptual frameworks. Descriptive level provides conceptual analysis of theories for conceptual level and theories and observations as foundations for prescriptive level items. On the other side, the prescriptive level supplies artefacts for prescriptive research and conceptual analysis. In summary, the conceptual level is interested in what is out there, descriptive research is interested in how things are out there, and prescriptive research is interested in how things may be out there and how to successfully achieve stated goals. As DSR results, the prescriptive level comprises both suggestions and artefacts. These have no truth or truth-like value in and of themselves, but they do contain claims about their efficiency and efficacy. (Hevner and Chatterjee, 2010)

## 3.4 DSR Evaluation

The requirement for DS research artefacts to be evaluated is highly supported by the literature. Studies show why DS research efforts should be published in high-quality IS research channels if the artefacts are thoroughly appraised in addition to other criteria. Even while everyone agrees that evaluation is necessary, there is no agreement on what constitutes desirable, acceptable, or conventional evaluation. Artefacts should be evaluated based on criteria that consider the needs of the context in which they are used, such as functionality, completeness, consistency, accuracy, performance, dependability, usability, fit with the organization, and other relevant quality attributes. Case studies, field studies, experiments, simulations, and architectural analysis, are all examples of evaluation methods, as is observing how well the artefact supports a solution to the problem. (Hevner and Chatterjee, 2010; Velichety and Srikar, 2012)

The artefacts produced from the result of DS study are varied. Are there any qualities of these artefacts that lead to specific forms of evaluation as a result of this? These are important concerns because analyzing and presenting the results of DS research is crucial to the efficacy of DSR research and the professional success of researchers. DS research reports are unlikely to be published in significant places unless authors can make convincing statements that artefacts were accurately evaluated. (Velichety and Srikar, 2012)

The usability of the artefact determines its validity. To establish its validity, the artefact should be reviewed both before and after applying meta-artefact design to the artefact design cycle. In design science study, there are two forms of evaluation: artificial and naturalistic. Artificial evaluation includes simulations, field experiments, and laboratory studies that are all constructed or non-real in some way. Naturalistic evaluation refers to the entire examination of the situational artefact in its intended environment, the application domain. Naturalistic evaluation methodologies include case studies and action research. (Knutas et al., 2019)

Right from the outset of a DSR process, evaluations should address the validity of incremental design decisions. Sonnenberg et al., (2012) suggests that evaluations in DSR should be conducted according to three principles described in **Table 5.**

**Table 5.** Principles of DSR evaluation (Sonnenberg et al., 2012).

| **Principle** | **Description** |
|---|---|
| Distinguishing between inner and outer modalities of DSR investigation | The artefact's constituents and design decisions, as well as an assessment of the artefact's usefulness |
| Prescriptive knowledge documentation in the form of design theories | Documentation of prescriptive knowledge in an organized manner. This would make it easier to communicate and disseminate the prescriptive knowledge generated by a DSR process. Furthermore, such documentation would already have a truth-like value that would merit inclusion in a DSR knowledge base. |
| Ex ante and ex post evaluations are used to continuously assess the DSR development. | This guideline encourages the design researcher to conduct several assessment episodes throughout a single DSR process iteration. |

Hevner and Chatterjee (2010) further state that the reasons why information systems are reviewed in the first place include promotional, academic, and practical objectives. To increase IS adoption and use within companies, IS systems must be evaluated and proved to be not just functional but also safe, dependable, and cost-effective. Continuous investigation of the structure, function, and effect of information resources must be a primary strategy for exposing its principles as a significant scientific subject, and most reviewers place a premium on how well the system or proposed technology has been tested and compared to other similar systems when reviewing new research. If designers do not analyze new systems, they will never know which techniques or procedures are more successful, or why some approaches fail, and designers can only learn about the nuances of their designs and contribute to the corpus of information for future designers to learn from by evaluating them.

In the design of software systems and digital artefacts, performance is frequently a critical factor. Designers, scientists, analysts, and consumers all want to achieve the best performance for the least amount of money. Jain (1991) states that any attempt without a goal is doomed to fail, and although the requirement for a goal may seem self-evident, many performance endeavors begin without one. One typical argument is that the model will be adaptable enough to solve a variety of challenges. There is no such thing as a general model and each model should be created with a specific aim in mind, even though setting objectives is not a simple task.

As mentioned, there is not universal way of conducting evaluation in IS. There are various evaluation strategies available. If the focus is on technical performance, analytical modeling, simulation, or actual measurements can be used. On the other hand, if the focus is on organizational influence, quantitative surveys or qualitative interviews can be used to do research. Ultimately, it is up to the designer to determine how to evaluate the system. (Hevner and Chatterjee, 2010)

## 3.5 Design Science Challenges

The difficulty that designers encounter daily is dealing with trade-offs. It is the deliberate choice among many alternatives, where each of selection imposes limits on usefulness and resources. As previously said, a designer has one foot in the realm of technology and one

foot in the area of human concerns, and these two worlds are not comparable. (Hevner and Chatterjee, 2010). Design decisions can have a significant impact in the real world, it is not always evident how to put design ideas into practice, and design features that are not related to design principles might hinder efforts to evaluate artefacts (Brocke et al., 2020).

Software design has nothing to do with the internal workings of the code or the size of the code. The designer's job is to describe the user's entire experience in detail and without ambiguity. When designing, the most important aspect to consider is the user's conceptual model. Everything else should take a back seat in order to make that model as clear and obvious as possible. Furthermore, rather than developing better expertise over software design processes, designers are continually challenged by new software technologies, higher quality requirements, and increased complexity of integrated systems. As a result, software design remains a fundamentally difficult problem that is frequently tailored to each software-intensive system built. (Hevner and Chatterjee, 2010; Velichety and Srikar, 2012)

## 3.6 Software Process Modelling

Software process models encapsulate the structural and behavioural features of software development activities, allowing for the elicitation, analysis, simulation, and optimization of software development processes (Kneuper, 2018). With increased expectations for the flexibility and adaptability of development processes, the continual evolution of development methodologies and tools, and the trend toward continuous product deployment, process engineers want enhanced support in terms of widely applicable modelling notations (Dumas, et al., 2016). This section addresses these generally applicable methodologies and approaches for modelling various stages of software development. The aim of the section is to highlight and bring into focus some of the methods found in the literature, and to utilize the best practices to support constructing the thesis design artefact.

### 3.6.1 UML Modelling

The Unified Modeling Language (UML) has emerged as the industry standard for modeling software systems. The UML is widely utilized in all aspects of software development and is well-known throughout the industry. Modeling is a popular strategy in software development and will grow more popular as model-driven technology becomes more widely employed.

(Chaudron, 2012). UML clarifies aspects of design, it can depict the objects or classes in a system, and the associations and relationships that exist between these entities. These models serve as the link between system requirements and system implementation. The level of detail required is determined by the design process used. Abstract models may be all that is required when requirements engineers, designers, and programmers collaborate closely. As the system is built, specific design decisions may be made, and problems may be handled through informal discussions. (Sommerville, 2016)

Modeling with UML is thought to help to common knowledge of the software system and better communication, particularly in bigger and dispersed projects. In turn it prevents costly rework, that happens in projects that do not include modeling. Another significant advantage is that early modeling comes with early validation, verifying that the correct system is being created, and development work is not wasted on the irrelevant tasks. Chaudron et al. (2012) also argues that the modeling into a tool is not the time-consuming part, but the real work is more likely to be spent on creating, discussing, and reaching an agreement on a design.

### 3.6.2 Artefact-Centric Business Process Model

Today's business process modeling and design confront some challenges when it comes to dealing with modifications and expansions of processes to capture new business needs in a systematic manner. Organizations require a more efficient and systematic method to reuse pre-designed process models. Several advantages have been demonstrated to incrementally develop and support complex yet loosely linked software components, as found in object-oriented software reuse. A reuse mechanism is viewed as very appealing and vital since it delivers various economic benefits, including qualitative improvements, cost, and time savings. Similarly, business process reuse should strive to facilitate the flexible extension and customization of current process models. (Yongchareon, 2020).

The concept of artefact-centric business process modeling has been proposed to shift how we model and design a process from an activity-oriented viewpoint to understanding significant business-relevant entities and how they develop throughout the course of the process's operations. Based on the theoretical research conducted in a study (Yongchareon, 2020), an examination is conducted into the necessary and sufficient behavioral-preserving

consistency criteria for specializing artefact lifecycles and associated synchronization dependencies. This technique may be more capable of aggregating artefact and process lifecycles and seeing them at a more abstract level. This feature makes it easier to monitor and report on corporate data and procedures on a continuous basis. (Yongchareon, 2020).

There are several ways to represent business artefacts. Many prefer to use a database schema, while others think of artefacts as a collection of properties or variables. Another option is to layer a relational database with an ontology represented by description logics. Although several of these alternatives include a formal description of the artefacts, none of them depict the items graphically. This has certain drawbacks. The models are more difficult to comprehend, for example, it is more difficult to perceive how the artefacts interact with one another and with other objects. Estañol (2018) argues that none of the approaches evaluated possess all the following characteristics. 1) Approach can represent all the dimensions in the BALSA framework, and it employs a well-known formal language. 2) Business modelers and developers will be able to understand this approach. 3) It connects the artefact-centric and process-centric approaches. 4) There are tools available to model the business process in accordance with the strategy.

### 3.6.3 Business Process Modeling and Notation

Unlike software process modeling, business process modeling has gained a greater level of consensus and standardization, culminating in the Business Process Model and Notation (BPMN). Because of BPMN's success as a standard business process modeling language, researchers have speculated that it might be used to represent software development processes as well (Dumas, et al., 2016). This chapter delves into the subject by extracting assumptions regarding the characteristics of business process models, which ultimately determine which components of the process are included and which are removed from the model.

A business process is a collection of one or more connected processes or actions that are carried out in a specified order to achieve a business aim or policy purpose, often within the framework of an organizational structure that defines functional roles or connections. A process might be completely confined inside a single organizational unit, or it can span many

organizations. Business process cooperation across organizational borders is a difficult undertaking owing to the lack of a unified semantics for their BP model terminology and the usage of disparate standards in BP modeling and execution. BPM on the other hand is the practice of modelling an enterprise's processes for the existing process to be examined and improved in the future. The fundamental objective of BPMN is to offer a notation that is easily understood by business users, who range from business analysts who make early drafts of processes through technical developers who construct them, and lastly to business employees who implement and maintain such processes. (Chinosi et al., 2012)

BPMN is based on three key assumptions that influence its application. A process is made up of distinct process instances that communicate with one another in extremely limited ways. A case is a series of activities that change the state of things associated with the case. An atomic activity is the atomic unit of labor for a single actor. BPMN process model is a graph which consist of events, activities, and gateways. Events are things that happen at a specific point in time. Activities signify tasks that must be completed. Gateways direct the flow of control via the process model's branches. Nodes are linked together via directed edges known as sequence flows. (Dumas et al., 2016). BPMN has three basic types of gateways: (XOR, AND, and OR) gateways. These gateways represent logical operators when there are multiple ongoing or incoming flows.

BPMN is a great tool for modeling a business process and gaining a better understanding of how the process works. By modeling the system's operation, approaches for improving the process could be sought by looking at the entire flow of the process and understanding the technological restrictions and opportunities.

### 3.6.4 Model Selection

The thesis objective is to produce a design artefact that is as relevant and accessible to the development team as possible, and as there is no consensus in the literature on the best method to express an information-centric business model due to the multiplicity of use cases, no novel ways to modelling the system were used. As a result, UML was employed to illustrate the sequence flow of the design solution and BPMN modelling is utilized in the

work to model a higher level of the system since it is easily understood by business users and developers who construct the system.

## 3.7 Migration Phases from Monolith to Microservices

Even in the current microservices era, migrating from traditional monoliths to a services-oriented design is a multidimensional and hence nontrivial task. It is difficult to find thorough instructions for refactoring legacy apps. The importance of the problem is backed by the high costs and effort of the refactoring process, which has a variety of other repercussions, such as overall processes like DevOps and team composition. (Fritzsch et al., 2019) A key challenge in the context of migration, however, is the extraction of microservices from existing monolithic software's. (Mazlami et al., 2017; Assunção et al., 2022) The difficulty of identifying microservices in legacy code is viewed as a software re-modularization task that is known to be an NP-hard problem, which means that there is not an algorithmic way to solve this task. Furthermore, there are desired attributes to be accomplished or restrictions to be considered when doing the re-modularization. (Assunção et al., 2022; Mathai et al., 2021)

Methodologies for identifying microservices can be categorized in three groups. First, model-driven, which rely on design elements, such as business objects, use cases and domain entities. Second, static analysis, approaches based on the source code for identifying the microservices. Finally, dynamic analysis techniques, which focus on the examination of system functionality at runtime and use execution traces as a source of information to organize source code entities into microservices. These existing methods use various criteria such as coupling, cohesion and clustering strategies to generate recommendations for prospective microservice candidates to identify microservices. (Mazlami et al., 2017; Assunção et al., 2022)

Without a well-thought-out technique, migration might turn into a trial-and-error activity that not only wastes time but also leads to the wrong results. Furthermore, because aspects such as business needs, present circumstances, and the skills and experience of team members may differ between firms and scenarios, a single and inflexible technique would

not be sufficient. Before microservices, the ideal outcome of a migration was to add a layer of abstraction in front of the legacy monolith, allowing for a more flexible architecture while prolonging the lifespan of the old system. The ultimate goal for established firms in the microservices age is to replace their on-premises legacy monoliths with a functionally similar set of cloud-based microservices that can be designed, deployed, and created independently. (Frey et al., 2015; Mazzara, 2021)

As a result, rather than a one-size-fits-all methodology, it is better to opt for a situational approach (Balalaie et al., 2016). However, it is important to consider how the process can be carried out. Megargel et al. (2020) offers a phased approach for migrating from a monolith to a cloud-based microservice. The following six phases were used in an actual banking system migration project, where monolith was migrated to microservices.

Phase 1. *Decouple Monolith*

To prepare for the inevitable move away from the monolith, a facade layer between the user interface and the monolith is commonly used to decouple the front-end layer from business logic layer. The facade layer implements logic that mimics the underlying monolith interface, allowing any current user interfaces to be physically separated from the monolith without requiring code modifications. Then, above the facade layers, a service mediation layer is added to offer run-time control over the channel-to-service mapping. This allows the complete collection of channels to be reassigned to consume microservices without changing a single line of code in any of the channels.

Phase 2. *Develop Local Microservices*

Disassemble the monolith into individual microservices. This might entail reverse engineering the monolith to find potential microservices. The most time-consuming and crucial phase in the entire migration process is microservice identification. For any microservice, it is critical to achieve the optimal amount of cohesiveness and loose coupling. Create a library of microservice interface definitions that can be loaded into a variety of microservices development and testing tools that follow industry standards. Microservices should be developed, and unit tested. The microservice must implement the same business logic and data structure as the monolith's original function.

Phase 3. *Implement Local Microservices*

After the microservices have been developed the following steps are repeated for each microservice. Data should be transferred from the monolith to the microservice. Perform a parallel run in which the channel calls both the monolith and the microservice, and the data is reconciled between the two. Reassign the channel to the microservice instead of the monolith in the channel-to-service mapping. This method can be continued until all the channels are communicating with microservices.

Phase 4. *Deployment of Microservices to the Cloud*

Set up an API Gateway in the target cloud environment to act as a single point of entry and control for microservices invocation. Transfer the on-premises microservices to the cloud. Any necessary microservices administration and monitoring tools should be put in the cloud. End-to-end testing should be performed to ensure that each microservice can be accessed from the outside via the API Gateway.

Phase 5. *Implementation of Microservices*

This stage is similar to the third, but the operations are carried out in the cloud. Data is migrated to the cloud-based microservice. Parallelization is run, where the channel invokes both the on-premises microservice and the microservice in the cloud, and the resulting data is reconciled between the two. Finally, reconfigure the channel-to-service mapping and use the cloud-based microservice instead of the on-premises microservice. In the event of a software bug, channel to service -mapping can be temporarily delegated to the on-premises microservice.

Phase 6. *Decommission Monolith*

In this phase or even after the phase 3 the monolith is no longer in use and can be deactivated. The on-premises infrastructure then serves as a development and testing environment for microservices. Existing channels can be refactored to access the API Gateway directly rather than through the service mediation layer.

As mentioned earlier, this approach is not the only correct way to handle the migration, but it does provide a good perspective on how it could be performed. In reality, each application, aim and needs for the system are unique, thus a uniform guideline is pointless.

# 4 CASE SOFTWARE AND DESIGN ARTEFACT

This chapter examines the case company and its financial management software, as well as the problem with the built-in online store platform in the core software. The current systems architecture and software solution has been investigated, and flaws in the design flow have been sought. Based on the current weaknesses in the system, an iterative approach has been conducted for the creation of design artefact as guided in chapter 3.2 to answer to the thesis design goal.

The research in this project was guided by DSRM principles, and six steps proposed by Hevner and Chatterjee (2010) in chapter 3.1. These steps included first identifying the problem and justifying the value of the solution. Following that, the objective of the solution is defined in chapter 4.3. The planning and development of a design artefact is next, followed by the clarification of the solution. Later in chapter 5.2, the design artefact is evaluated, and a review on how well the design artefact contributes to the thesis design problem is made.

## 4.1 Background

The case company is a software company that provides SaaS services to other companies. One of its largest and primary products is a financial management software, which combines financial management and corporate business processes as a cloud service. It can be used to manage sales, purchases, human resources, payroll, and accounting, among other things.

### 4.1.1 Case Software Introduction

The topic of the thesis was specifically focus on the core software's built-in online store platform, which can be used for purchasing, deploying, and terminating value-added partner services that complement the core product. Purchases can be targeted in the form of bundles, extra services offered or facilitated by the product development team, such as electronic service channels, payment transactions, and so on. The integration partner store services are designed to be integrated software applications from partners that provide benefits to both the core software and the customer. The purchase and activation flow of an individual store service can be done by a company administrator or a user with sufficient privileges. For example, an accounting office user can buy services for one or many accounting office

customers. From the standpoint of a partner, they outsources their client sales to the core software.

### 4.1.2 Software Development Environment

The case software is a monolithic software solution that has been in development for two decades, implying that a wide range of technical solutions have evolved into a monolith over time. The largest entity in the system is the monolith, but during the writing of this thesis, the transition from monolithic to microservice architecture and newer technologies was well underway, and a few new implementations have already been implemented as separate microservices integrated into the core software. Software development is based on agile development principles, and the number of personnel in the development team has continuously increased, with the software presently having more than 50 people in multiple different Scrum teams. The internal store platform, a small entity constituted of monolithic, is the focus of this thesis. Its size corresponds to the size of one Scrum team development area of responsibility.

### 4.1.3 Software Components and Delivery Process

The software delivery procedure for the current monolith is carried out on a weekly basis. The distribution of new updates into production is done in such a way that the changes are integrated into a single larger weekly release branch that is published weekly. The reasoning behind this is that modifying a monolithic application requires for the publisher to compile the entire system to ensure everything works, and it is easier and more secure to implement a publication this way. There are also no complete continuous integration and delivery (CI/CD) tools for the monolith, and the release is currently manual, requiring time and labor resources. The database is updated weekly in accordance with the weekly release. The part of this system's software is written with .NET Framework. A relational database is used to store and associate data across multiple tables, and there are around 20 table’s relevant to the domain of this internal store system. However, the store system is dependent on other monolithic components, resulting in a larger total number of tables.

## 4.2 Design problem and motivation

This section defines the thesis design problem and justifies the value of the solution. With justification of the value of the solution, it motivates for pursuing to find a solution, and it also helps to accept the results and understand the reasoning associated with the problem (Hevner and Chatterjee, 2010).

Underlying to the system enhancement is the business goal to increase the number of integration partners and satisfy existing demand. The problem has emerged as a consequence of the growing demand for new partner services to be added to the internal store, which the present system is unable to manage on a desired speed due to its current architectural design. The present implementation necessitates a considerable amount of development effort to add a new partner service, and owing to the system's size, complexity, and monolithic state, it takes a long time to understand the system as well as time for the development team to make modifications to the current system. Furthermore, due to the difficulties of the monolith, system updates are frequently large and time expensive, and version releases occur only once a week, prohibiting agile change of delivery.

The motivation for the thesis is to provide a technical plan; an artefact that can be used to describe the current state of the system and provide a solution to the aforementioned problem, thus achieving business goals. In general, an artefact is something that can be turned into an instantiation, process, or program (Gregor et al., 2013). The artefact itself is an abstract implementation that can subsequently be realized into existence at later stages. The solution and proposals for system enhancements are based on findings in the literature, as well as expert opinions and architectural practices that support the current system. In this thesis the purpose of the artefact is to describe the solution to the design problems posed, thereby assisting the case company with the business problem with the internal store system and generating information about the system domain. By examining what the microservice architecture enables can justify the importance of change as in literature microservices architecture is considered as a major facilitator endeavouring digital change over monolithic architecture (Megargel et al., 2020).

## 4.3 Objectives

From the problem definition and knowledge of what is achievable the objectives of the solution are elaborated on in this section. The objective of this thesis is to provide solution to answer how to enhance the current system to add new partner services to the store more efficiently. In particular, the main objective is to create a system that allows a new service to be exported within one day to the store. As there is a growing demand for adding new partner integration services and the current approach takes up two to four weeks for development time, it is better to design process to automate this work. The central question in assessing the artefact is how well it can be used to create a system that solves the case company's problem of automating the addition of new store products to the store.

There have been several approaches to conducting design science research, all of which indicate a process that consists of two high-level activities: construct and assess. Evaluation activities occur after the creation of an item, which is also a frequent assumption in other DSR processes. (Sonnenberg et al., 2012). In the thesis the first goal is to construct the design artefact and when the artefact is completed, the second goal is to evaluate the artefact profoundly. The process of construct and evaluation is done in an iterative manner following Sein et al. (2011) recommendations building and evaluation processes to be done concurrently to instantly reflect progress and to drive artefact modifications early in the design phase. The process of construction and assessment is described in **Figure 4** in more detail.

The quantitative objective is to speed up and automate the process so that the new store product addition may be completed within one day. There are several challenges for this to happen. Enabling better design of the new store design system, the current process needs to be thoroughly explored and its weaknesses and limitations identified. The new design model must also consider the possible limitations of the case software’s current architecture and strive to find a solution that serves the requirements from a technical and business point of view.

The desire to expand system automation also creates new problems for the technology and architecture solution used. To avoid incurring additional technical debt, it is preferable to

begin by enhancing the system by establishing new architecture patterns that allow automation rather than further develop what has already been done. The microservice architecture seems like a promising practice to enable goals, which is why the other qualitative objective is to what benefits microservice architecture provides over traditional monolithic approach.

## 4.4 Design requirements

This section describes the specifications for a design artefact that supports business objectives. The system improvement needs are not definite, but iterative improvement targets that are technically viable. Design requirements highlighted in **Table 6** emphasize the most significant technological directions in which the solution should be advanced. The process of establishing these design criteria was done iteratively. They are designed to emphasize essential objectives, avert potential future difficulties, and justify the necessity for systemic architecture changes from a monolithic to a microservice.

**Table 6.** Design requirements for the design artefact.

| | |
|---|---|
| 1. | Reduce the amount of development work required to set up a new store service. |
| 2. | Management UI for admin users to add a new store service. |
| 3. | Generalized model for a store product. |
| 4. | Changes to the system cannot be dependent on a weekly publication. |
| 5. | External API for the partner to maintain store product content. |
| 6. | Invoicing procedures and billing model that are consistent. |
| 7. | Service termination that takes into account billing processes and customers. |
| 8. | Processing of expired partner keys during service termination. |
| 9. | Data collection on visits, openings, purchase process, closures, invoiced billings. |
| 10. | Informing the partner about the openings and closures on a monthly basis. |
| 11. | Automatic distribution of metrics to partners. |
| 12. | Enhancements in UI flow compared to the current one. |
| 13. | Displaying a targeted and general recommendation about store services in the core application. |
| 14. | Personalization of the offer based on the company’s industry, general need, or the role of the user. |

The paragraph excludes requirements that have no bearing on the technical solution, such as customer service requirements. Efforts have also been made to plan invoicing-related measures so that no current changes to invoicing logic are required.

## 4.5 Research steps and design challenges

This chapter explains the study's progress in practice, the steps involved, the conclusions, discoveries, and drawbacks discovered about the current system, and the design decisions justified by the current system problems. At the beginning of the work currently achievable technical solutions were discussed. Discussions with software architects and developers were held, and boundary requirements and constraints were discussed. Business requirements were mirrored based on the architecture and the feasibility of practical implementation options and discussions with business side of stakeholders continued iteratively. The current system's implementation was questioned, and it was assessed which areas made sense to start making improvements to the old platform and which should be done entirely from scratch in accordance with better architectural practice.

In the following stage the overall picture of the existing implementation was examined and its reliance on various areas of core software such as billing, pricing models, and user interface were explored in detail. Because of the current size and complexity of the current system, the user flow and its orthodoxy also had to be thoroughly investigated. The clarification began at the code level looking at the source code, and implementations. It was discovered that some of the code had characteristics of code smell. For example, a duplicate, speculative prevalence, was found. Furthermore, the issues raised in past conversations about the code's monolithic state grew more concrete. It was highlighted that the existing system is somewhat tightly coupled with the monolith in many ways, which implies that for example detaching billing from the system must be developed as a separate entity.

As it was easier to find out about the current implementation from people who had originally done it, a conversation with a developer who had done most of the current system was held. It improved understanding and reasoning of the system furthermore. It was also found out that the original intent of the store was to take only a few integration partner products to the store. The business needs at the time did not take into account the need to integrate new store

products on an ever-faster schedule, and the scaling of the system had not been considered in more detail. It was also disclosed that for new store services to be produced at a faster rate by automating the process, individual services must follow the same model, and custom cases must be excluded.

During the thesis there was also a need to add a new partner service into the current store system, and there was no time to wait for a new system to be created. The new product was added following the old technique, but also considering what information an individual store product requires and how the current addition process operates. Implementations of other integration partner services were also examined at the code level. Throughout the addition, several smaller procedures that could be automated, actions that took a long time to implement, and steps that required the time of additional specialists were discovered and noted. Throughout this procedure, the system's drawbacks became apparent, and the scope of the system became clearer. Disadvantages are documented in **Table 7**.

One step also that were noted time taking process during the implementation with the old system was gathering all the relevant information about the needs for a specific partner product. The information flow was not clear, and pieces of information was coming around from different places and people. This small procedure could be made better with general service model and clear understanding of what necessary parameters to know beforehand, and what could be updated later. Ideas like external API for a partner to update information about their built-in store service came up. It was also noted that the store products lacked a clear general model of what should be included in the service, which has led many existing implementations to be tailored to meet the demands of the client. This complicates matters for both the client and the developer because it provides a hazy picture of what the store system permits.

**Table 7.** Disadvantages of current system.

| **Disadvantages** | **Solution** | **Extent of alteration** |
|---|---|---|
| Development dependence | A tool for automation | medium |
| Human dependence in configurating changes in monolith | The solution cannot depend on a monolith | large |
| Rigidity in change of delivery and understanding of the system | Microservice solution over monolith | large |
| Custom cases with store products | Automation needs to take into account more complex logic, and changes need to be made | medium |
| Testing with partner | Automated testing and better support for integration partners testing site | medium |

The weaknesses uncovered throughout the research backed up the preliminary decision to re-implement the system rather than making modifications to the current system, which could have resulted in technical debt. Albeit the scope of re-implementation development work is large and time-consuming, plans are promoted to achieve the best solution, which means introducing microservice architecture for the project.

## 4.6 Creation of design artefact

In this section the proposed design artefact is described. The objective of the design artefact was to provide a solution to the case company's problem of how to improve the current store system, so that the delivery of a new integration partner product can be done in a single day to the store. To accomplish this goal, the design artefact rationalizes the design solution with a BPMN model of the new addition process and a sequence diagram of the function of

internal tool for automation. The benefits and drawbacks of microservices are studied and the solution of rebuilding the system with microservice design and abandoning monolithic implementation is justified and supported by the literature.

The separate generic model for the individual product is not defined in detail, because no fundamental changes are made to the structure of the individual store product. The products in the first iteration of the new system will be similar to the current store, and the addition of new products will prioritize products with basic functionality that do not require custom implementations. In other words, automation is designed for simple store products that only transmits data back and forward via the APIs. The most significant reform is to enable the microservice architecture so that future development and the goal of a more automated process can be realized.

The conversations about the current built-in store system design, and ideas for the possibilities with the new system, and the launch of a new store product based on the current practices highlighted problems in the current process, which was laborious. As a result, an improved design flow in **Figure 6** for the new store platform was formed to enhance the operation of the system. The goal of this new model is to automate the addition of a new store product for the client.

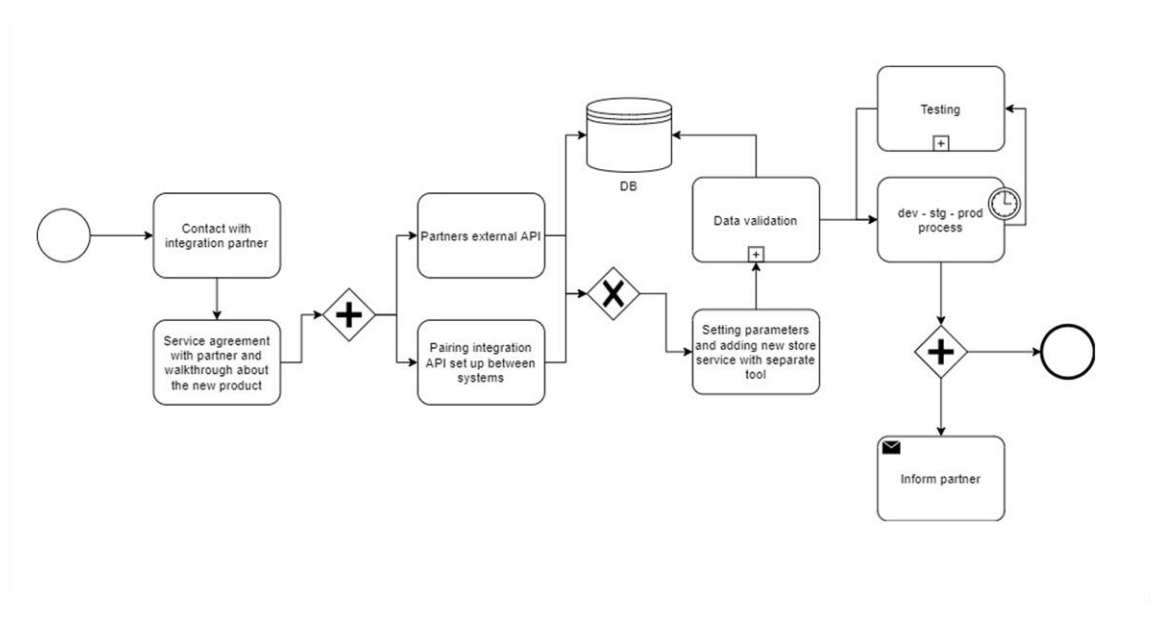


**Figure 6.** BPMN flow solution of the addition of a new partner service to built-in store platform

Adding a new store service, contrary to current practice, would not necessitate any development. An internal tool would be created for the internal operator to use with adequate privileges, that could create a new store product with the tool, which implements an API for the insertion of a new store product to the database. Furthermore, the system's UI might be constructed entirely on a database structure, allowing a new product to be put into production anytime the store automation tool is used. Agreements and setting up integration between the case software and the partner client would continue to be done in the same manner, but in this model, partner would now be able to update the information on the essentials of the product in advance using external API.

**Figure 7** illustrates the operation of the current process through the essential measures. These pieces of time-consuming operations include various configurations phases and for example testing against a partner API, which takes time. Development work itself also takes time, but that is not the most time-consuming part of the process.

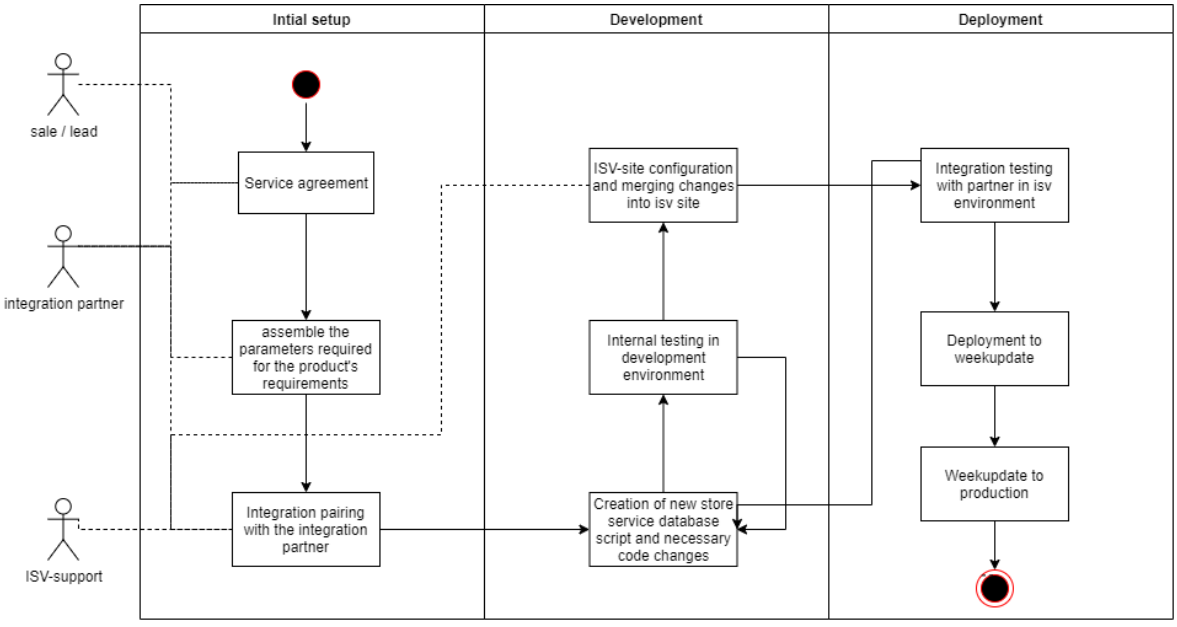


**Figure 7.** Action flow about the addition of a new store service process in the current system.

Main issue is not only the flow of the current process, but the constraints imposed by the monolith architecture because in that the change of delivery to the customer is rigid. As a result, to answer the problems, the system could be made as a microservice, so that the application can be independently deployable, and not be dependent on weekly update cycles.

A microservice design would also speed up the development process and allow rapid modifications, as the microservice would be embedded in the core software. The current architecture enables for such solutions, but in return it requires a re-implementation of whole current software solution. In the current implementation, there is some generation of a UI based on the database structure, but the notion may be developed further. The execution of the database changes is better to perform as an API request, which would speed up the modifications and data alteration on services.

The ISV environment is an environment for ISV partners for testing and other relevant procedures. The disadvantage of the current model is that for testing and quality assurance, the implementation of the ISV-site must be configured separately. Here, the developer also needs internal support, which continues to cause personal dependencies, and takes time away from other things to do. In the new model, no code changes would be required for the ISV-site separately if the administration tool for creation of a new store service were to handle the operation. The most important part of the new model is therefore this internal tool, which would be at the center as an enabler of automation. In **Figure 8** this process has been described as an UML sequence diagram.

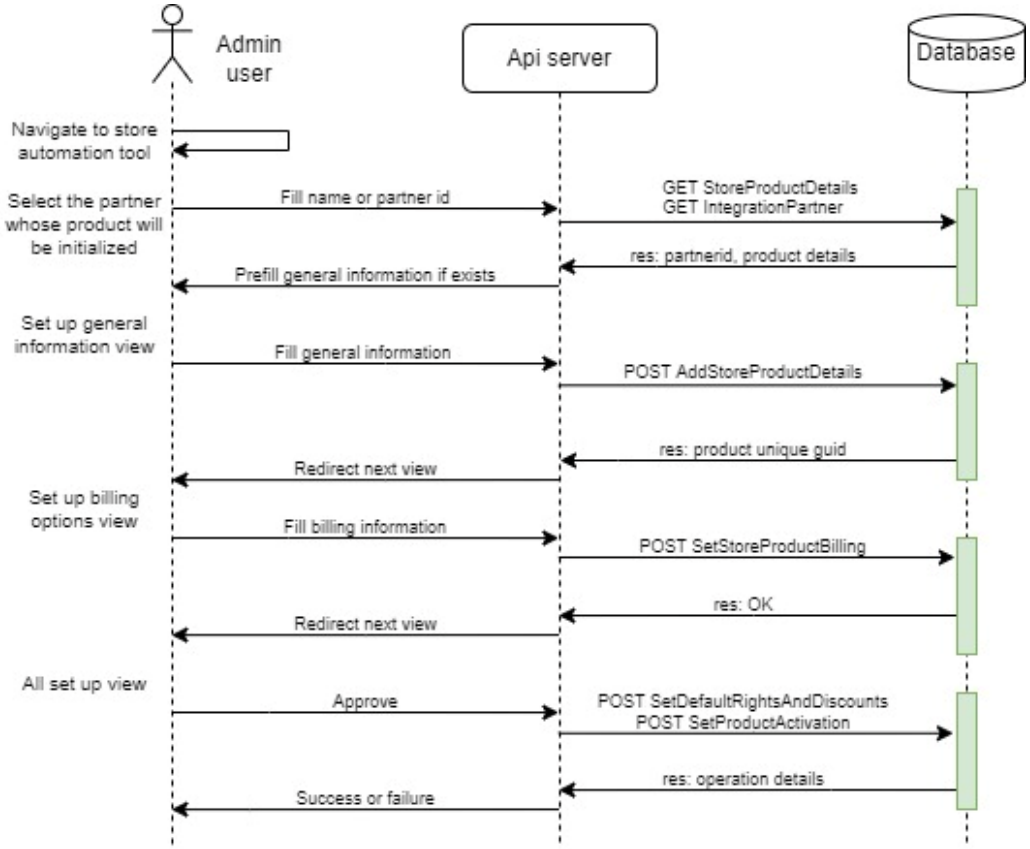


**Figure 8.** UML Sequence diagram for addition of a new partner store service to the built-in store platform

First, an internal admin operator with the necessary permissions navigates to the user interface where the action can be carried out. Based on the name or ID, the admin user can then select the partner whose new product they want to configure to the store. Following that, basic information such as product name, product description, product image, billing approach and so on can be configured. For example, in billing, the service can be designed to be billed to the end of case software, or it can be designated to be handled by a partner. Service billing criteria options for billing initially would be transaction-based invoicing or monthly invoicing by company. The addition could then be finished by approving selected choices and the process continues with HTTP request to the API. The API would be in charge of adding a new product to the database as well as notifying any issues or errors that may arise. Though this administration tool would not take into account difficult custom cases, as its complexity would increase too much. Promoting custom implementations can be a challenge if there are a lot of cases like that. This problem and potential error cases are discussed in more detail in the discussion section.

For everything to work in the new system, the UI must also be re-implemented. The technical requirement is that the UI can build the store product supply and the content of individual products on the basis of the database structure automatically. This requires the implementation of API interfaces to obtain the necessary datasets for creation, as well as possible improvements to the database structure. Also, consideration on how the user interface will receive information about the addition of a new product to the database has to be taken into account. Possible solutions have been considered more in the discussion section.

## 4.7 Phases of Decoupling Monolith to Microservices

A key challenge in decoupling monolith into manageable entities is the identification and extraction of microservices from the existing monolithic code base. (Mazlami et al., 2017; Assunção et al., 2022) Also, according to Fritzsch et al. (2019), the biggest problem throughout the migration is determining the right granularity for the microservices. Approaches for the identification in this occasion relies on model-driven approaches, where description of system is done using design elements, use cases and domain entities. The decomposition of microservices is based on business competencies, also usually referred as bounded contexts.

The proposed solution to create a new internal store system to the case software requires changes to the microservices shown in **Figure 9**. Simultaneously with the thesis, progress has been made in the development of case software in relation to other sections, and some of the internal API containers required in this context are already containerized and operational in cloud environment. For that reason, only the new API routes and domain logic associated with the development of the new store will be required to be developed. The UI for the tool handling automated creation of an individual store service is required to be build, and it will be individual application. In the **Figure 8**, the flow of an administration user who can create the new store service is presented. The new store UI application, which will be the layout of new store will be individual application. More containers for API servers may arise after the development efforts is started, where each resource is determined by the core products boundary contexts, but in this context at this stage it is irrelevant.

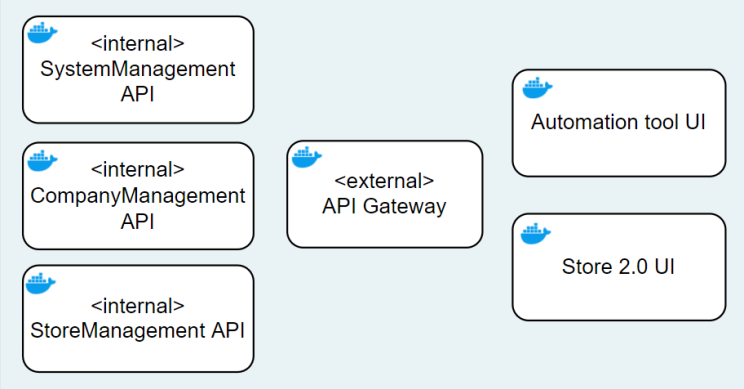


**Figure 9.** Microservices for the first iteration.

As mentioned in literature, utilizing the microservice architecture, different entities can be developed and updated independently. This allows for rapid change and breaks the system into manageable entities that are easier to understand and deployed. Microservices bring their own challenges, such as connections between different containers, but in general, the benefits outweigh the disadvantages. Another issue that remains is the monolith's database dependency, which cannot be addressed right away; the monolithic data is tied to other services, such as the billing service, which is currently out of scope to be refactored as a microservice. Although these microservices would not move the database to the microservice, they would still process data via API, making it easier to detach the data from the monolith eventually.

Megargel et al. (2020) propose a stepwise migration path, detailed in **3.7** from a monolith to a cloud-based microservice. In the solution to the case company's problem the literature is used as a model for how the existing store entity's monolith should be converted into microservices in the cloud. The present domain logic should be implemented as API interfaces in the first step. It is also possible to test that the current functions conform to the API interfaces at this point. Following that, the monolith's components are separated into distinct micro-service entities that can be built, unit-tested and maintained independently.

The local micro-services that must be implemented are the new store UI, API implementations and management tool for automating the addition of new store service. In this case, the database migration should not be transferred to the microservices, as the existing data is dependent on the implementations of the other monolith and its transfer should be left to a later date. After this, the local microservices will be deployed to the cloud

and the new store entity can be tested in a test environment. In addition, service monitoring and system scalability can be tested. In the final stage, when the microservices operating against the current monolithic implementation meet the business requirements and the various tests have passed, the switch from the monolithic solution can be made to a the new embedded microservice implementation.

# 5 RESULTS

The findings of the research and solutions to the thesis design problems are discussed in this section. Furthermore, utilizing design science concepts based on DSR theory, the generated design artefact is analysed and appraised.

## 5.1 Results and Analysis

The thesis examined and addressed the problem of the case company using design theory. To characterize the problem and the work's objectives, the main design problem was established using design theory:

Design Goal: *The system should be able to process the delivery of the new partner's integration product to the internal store within one day.*

The design goal's main objective is to provide an answer to the design topic of creating an improved software system design that could handle the addressed issue. Sub-design requirements were created to help to structure the design problem into easier-to-handle entities and to open the issue for further investigation. The first sub-design requirement's goal was to gather information about the current system and find the design weaknesses of the current system. Its objective was to justify measures to build a new system that would provide more value and provide relevant domain knowledge about the current system. The first sub-design requirement is as follows:

Sub DR1: *Find the design weaknesses of the current system*

The current system starts with a commercial agreement with a partner about the product and its details. The partner's service is then paired to the system, and the integration secrets are exchanged so that the data can be exchanged between applications. In the process it is agreed with the partner what specific API resources are opened from the case system's end, what basic information about the integration product has, and how is the billing handled. The product is then exported to a development stage where the necessary code changes and database additions are being implemented. Once the functionality has been proven in the

development environment, the new store product is then migrated to a test site for integration partners, where testing will be repeated with the partner. Following that, the code and database changes can be merged to a weekly publication, after which the product will be put into production for end-customers to use. The process can take anything from two to four weeks, depending on how the different staff dependencies are free, whether there will be problems in the testing phase, and which weekly release the time will be included.

The current monolithic architecture is not built to handle the automated addition of new partner services on a fast and large scale. Initially intended solely for one integration partner product to support the core product, the built-in store platform has since been expanded to include other products. As a result, new integration partner services have been developed to the monolith on top of the original system design, with little opportunity for refactoring the codebase resulting functionally adequate but obscure system, where changes can take a lot of development time.

The analysis of the flaws in the current system is based on discussions with various stakeholders and a closer look at the current process. The current process of integrating a new partner product into the store is depicted in **Figure 6**. Staff dependencies, the rigidity of the development cycle, and the lack of a generic product model for the single store service are the main design weaknesses of the current system. In monolithic systems, the system frequently grows to a significant extent, affecting the comprehensibility of the system codebase and resulting in staff dependencies (Megargel et al., 2020). The rigidity of the development cycle is primarily due to the current monolithic architecture, which limits development flexibility because publications can only be made on a weekly basis. The lack of a general model for store service has resulted in the introduction of customized solutions for specific products, resulting in code smell.

In a monolithic system, exporting the code to a different version takes time as the entire monolith must be compiled. Also with monolithic systems, new developers must spend time understanding the system, which adds to the issues. Small entity automation with CI/CD tools would be possible, and more agile solutions with microservices. It is difficult, if not impossible, for large monoliths to implement modern DevOps methods and technology (Megargel et al., 2020).

The design artefact presented in the thesis for a new store system could solve or at least introduce an improved version with most of the design issues that have arisen. The microservice architecture can break free from the weekly publishing cycle and decentralize the system to managed entities, which aids in the understanding of the specific entity. Furthermore, a management tool designed to export a new product eliminates the need for staff, and at best, only one person is required to complete the entire process of importing a new store product to production.

The second sub-design objective was to identify the system's major functionalities and essential parameters so that they may be considered in the creation of design artefact. It also prioritized what part of the process to focus on, and which parts in the system are necessary for the design artefact to function fulfilling the business and technical requirements. The second sub-design requirement was:

Sub DR2: *Define the main features for the design artefact that would meet the technical and business requirements.*

It was discovered how the existing system operates and what its flaws are based on the thesis first sub-design topic. Based on this knowledge and the needs for the new system, artefact-centric process models could be utilized to create a design artefact to represent the new system. Depending on the needs of the basic requirements, the system must be able to add the product faster using techniques that would enable higher-level automation and easy usage.

Inspecting the current system and migrating the new integration partner service to the existing built-in store, it was found that in order to meet the new requirements, there must be some tool or process to automatically add a new store service by eliminating the coding phase and staff dependency needs. The tool should receive required parameters that the new product requires, as well as support some optional parameters. To accomplish this, in first iteration only general store product will be utilized in the automation tool. Then admin user could use this tool for adding a new store service. Furthermore, the UI should be able to generate only from the basis of the database structure, where each store product, and its information is stored. In addition, the implementation should be detached from the monolith

so that there is no need to rely on weekly publications with changes. This would be made possible by a microservice architecture where the new store could be implemented as multiple microservices embedded to the core application.

The third sub-design requirement's objective was to specify the advantages and disadvantages of microservices in general as well as provide information about the benefits and drawbacks of micro-services compared to the monolithic solutions:

Sub DR3: *Examine the benefits and drawbacks of microservice architecture design.*

Over a monolith, the microservice design provides superior maintainability, scalability, and availability. Components in target architecture designs can be upgraded, deployed, and retrieved separately. Each component in the container may be self-contained, allowing it to be deployed and controlled individually, thanks to the planned architecture's loosely linked approach. Monolithic programs use the same programming language, and because software coding standards change so quickly, the entire codebase might become obsolete in a matter of years. Microservices, on the other hand, may be constructed with a wide range of technologies, therefore this is not an issue.

Microservices have also proven to be successful for agile development teams since they allow them to work on little objects that can be delivered without having to build the monolith. On the other hand, the complexity of a microservices-based architecture grows in lockstep with the number of deployed microservices, resulting in management challenges. Monitoring and management tools are required to track the status of microservices and restart any that have stopped running. (Rajasekharaiah, 2021; Megargel et al., 2020)

Returning to the thesis original design problem and goal, and considerations of sub design requirements, the study suggests decoupling the current system from the monolith and creating the new system as an embedded microservice to the core software. The system should also have an admin tool that enables for the automated addition of new integration services to the store platform. As a result, new containers, APIs, and user interfaces are required to be build.

## 5.2 Evaluation of Design Artefact

This section discusses the design artefact evaluation technique that was implemented. Chapter 5.2.2 further outlines how the evaluation was approached, and it summarizes the evaluation outcomes. Throughout the project, the evaluation was done iteratively.

### 5.2.1 Evaluation Strategy

In the literature, there is no unambiguous solution for carrying out the design evaluation as projects are different. However, one positive commonality is that design artefact evaluation should be an iterative process in the context of system development. The strategy for the selection of DSR evaluation process was utilized in this work as suggested in paper Venable et al (2016). It proposes four steps: (1) explicate the goals of the evaluation, (2) choose the evaluation strategy or strategies, (3) determine the properties to evaluate, and (4) design the individual evaluation episode or episodes.

Step 1: *Explicate the goals of the evaluation*

The aim of the evaluation is to answer whether the design artefact is sufficient to enable development work of the proposed system, and whether the solution meets the business objectives, and can the individual Scrum team be able to implement the system as intended with chosen architectural approach. In other words, can the development of the proposed system begin within the case company with the use of the design artefact.

Step 2: *Choose the evaluation strategy or strategies*

The "Technical Risk and Efficacy strategy" has been chosen as the evaluation strategy. It was chosen because the most significant design risk is technological, and it is impossible to evaluate the system with real users and real systems in a real environment.

Step 3: *Determine the properties to evaluate*

The ISO 9126 quality model specifies six characteristics: Functionality, Reliability, Usability, Efficiency, Maintainability, and Portability, which are further subdivided into 21

sub qualities (Behkamal et al., 2009). The properties used in this evaluation are adapted from this standard as the design artefact proposes a complex piece of software. The following properties functionality, efficiency, usability, portability, and organizational fit are used for the evaluation. Organizational fit is not defined in ISO standard, but it is important as the artefact is done for the case company.

Step 4: *Design the individual evaluation episode or episodes.*

The evaluation process considers the goal of the artefact. The objective is considered according to the properties defined in step three. Evaluation is done analyzing the proposed design artefact and justifying whether the goal is reached.

### 5.2.2 Evaluation

The purpose of the design artefact was to give a design solution for building a software to improve the present procedure for adding new partner services to the built-in store more efficiently. In particular, the fundamental question in evaluating the artefact is how well it can be used to develop a system that addresses the problem. The goal for the improved store system was to shorten the development time in taking a new service to the store from current two to four weeks to one day. The resulted design artefact, that describes how the system could be develop constructs of a BPMN model of the addition process and a sequence diagram of the functionality of internal tool to enable automation in the proposed system. The microservice architecture as a proposed solution for the new system has been reviewed based on the literature and system constraints. In addition, an activity diagram of the current system has been described with the aim of modelling the problem and targeting pain points in the current flow, which further highlights the need of rebuilding the system as a collection of microservices.

The technological solution was reviewed with developers who had prior domain expertise of the current system, and answers about whether this was the best technique to accomplish the job were evaluated. The discussions were labeled as unstructured. The discussion demonstrated that the solution proposed by the design artefact is technically reasonable and supports the broader goals of reforming the system architecture. Functionality, efficiency,

usability, portability, and organizational fit were the design artefact attributes that were examined. In **Table 8**, an evaluation matching to each property has been provided as a summary of the interviews and activities. The advantages of the solution reflected the advantages of the microservice architecture in the application's maintainability and development. API implementation and correctness of specifications will still necessitate design effort during development. However, owing to the system's complexity, it does not make sense to design a comprehensive API specification, so it is best to start by implementing the primary and required APIs.

**Table 8.** Evaluation of design artefact.

| **Properties** | **Evaluation** |
|---|---|
| Functionality | The functionalities of the system proposed by the design artefact meet the requirements of the goal. However, a person with no prior domain knowledge of the current system would face difficulties starting development work based on models and this work alone, so more clarification is required. |
| Efficiency | The time-based efficiency goal is the same as the thesis's main goal. The proposed new system could automate the entire process, achieving the efficiency goal. However, once the first version of the system is obtained, the current process should be re-evaluated. |
| Usability | The proposed solution represented by the design artefact suggests that only one person could handle the process of adding new services to the store. At this point, evaluating usability is difficult, but this method would still be viable in terms of providing a technique to enable the goal. |
| Organizational fit | Models are created in easy-to-understand languages such as UML and BPMN. The microservice architecture has been implemented within the organization, and larger architectural plans are being developed around it. As a result, it makes sense to decouple the current system into microservices. |

| | |
|---|---|
| Portability | The proposed steps to further decouple the current monolithic system from microservices must be promoted further. This is theoretically possible and supported by the cloud platform, but a more detailed plan should be considered to reduce risks. In addition, a more thorough examination of the system's current dependencies is required. |

Continuing with the idea of creating the system from the ground up was the only logical alternative. The initial step in the development of the proposed solution should be to create API specifications that correspond to the new store platform system. Furthermore, these specifications could be used for the creation of new UI. Specifications makes it easier to identify relevant domain contexts, and which APIs are still needed, and which are missing. Also, potential changes to the API specifications are easier to implement in the early phase of development. Finally, the remaining APIs and services should be implemented, containerized, and deployed to the development cloud environment, allowing testing to commence. The automation tool should be created last as it relies on everything else to work effectively.

Because there is no testable version of the design artefact, the theoretical methods for evaluation are difficult to introduce at this point. As a result, it can be only ensured that the current stage contributes to the design goal and that the steps for continuing the development work are clear. The evaluation and validation of the correctness of the project objectives must be continued in conjunction with the development work.

# 6 DISCUSSION

This section delves deeper into the findings and conclusions that were not further elaborated on in the thesis solution. This section also looks at future opportunities and directions for how design artefact can be considered and carried forward in the case company with a development team. Furthermore, flaws and other issues that were purposefully excluded from the scope are discussed and justified.

The goal of the thesis was to improve flow in the current store system such that the addition of a new integration partner product could be integrated in a single day. This goal for the current system resulted in the system requiring re-implementation and a different architectural direction, because of the current system's technical debt. Technical debt is incurred as a result of implementing shortcuts in system architecture, which must eventually be repaid. If this debt is not properly handled, it may accumulate as interest, reducing the overall quality of the created software systems (Yli-Huumo et al., 2016). The current store system and its dependencies are implemented with the .NET framework, which can already be defined as a technical liability as Microsoft has announced that support for this Framework will be discontinued in the near future. This further elaborates on why iterative changes to the current system will become technical debt later. It was proposed that creating an altogether new system with a microservice design rather than making technical modifications to the current system would be better in the long run.

Difficulties in approaching the current monolithic implementation were complicated by the system’s current dependencies. The existing system contains domain dependencies on other projects as well as logical dependencies for example on invoicing, making it too complex task to migrate the database structures of the current implementation under microservices' responsibility. This was also considered throughout the planning of the project’s objectives, and it was concluded that the design solution to handle invoicing is left for the scope of this thesis but bearing in mind its existence.

The technologies chosen for the new system are based on case company's use of modern architecture-supporting frameworks, which in this case includes the use of .NET Core for backend API development and the React framework for UI implementation. Docker

containers will be used to containerize applications, which will be hosted in Kubernetes environment on the cloud. Individual issues with scalability, performance, or content delivery automation for the new system are solvable within the context of the case company's development method of operation, and only a relative amount of attention has been paid to these challenges, as it is known that these problems can be treated and solved.

One aspect was purposefully left out of the task scope: the created design artefact does not examine how to automate unique implementations for individual store services, as the complexity of internal automation tool would increase too much. The system's purpose is only to automate the addition of a generic store product to the internal marketplace. The choice was made not to consider things in greater depth, because demands are changing, and it is more important to focus on system that can be readily updated and well understood rather than a domain logic that can handle every business case. This decision should be kept in mind while constructing the new system and revisited later if necessary.

The concept for future development could completely detach the store database structure from the monolith. This could be performed by constructing data structures in another information system service that would correspond to the data of the current store system and converting the current data there. The migration of the database to the responsibility of a microservice has its own set of issues, and the current monolith's dependencies on billing and other parts of the system must be resolved first. However, the longer-term goal for a system should be that it is not reliant on a monolith at all and could be independently developed.

In certain cases, well-designed monoliths have surpassed microservices architecture in terms of speed, accuracy, and simplicity of maintenance. Another school of thinking holds that the success of microservices architecture is due to a change in technical expertise. Engineers who used to build high-quality monolithic applications are increasingly investing in microservice architectures. The argument over whether success is due to more smarter engineers or better design continues. (Rajasekharaiah, 2021) Although microservices are not a perfect comprehensive solution that will address all the system's difficulties, they appear to be a better solution than monolithic applications at this moment.

# 7 SUMMARY

The demand for software systems is increasing, as are the costs of running, maintaining, and improving software. Companies must modernize their software platforms to gain a competitive advantage in the age of digital transformation. Traditional monolithic systems, while functionally sound, are insufficient for the development of high-quality digital services. Businesses should explore a more agile approach to application architecture in their digital transformation activities, allowing for the quick delivery of new cloud-based services to customers. In this endeavour, microservices are considered as a vital solution. The basic idea behind this design pattern is that software is created as a collection of independent and modular services, allowing small, autonomous teams to innovate frequently, resulting in faster deployment of new digital services.

The purpose of this thesis was to address the case company's problem of an increasing need to add new integration partner services to the built-in store platform of the case software. The DSR literature was used to help approach the problem. The design goal was to improve the system and create a design artefact that could automate the process of delivering new integration partner's services in one day, rather than the current two to four weeks of development time. The current system's design flaws were identified, and the main features were defined, establishing what should be prioritized and which pieces are required for the design artefact to function properly and meet the design goals requirements. In addition, the benefits, and drawbacks of microservices were examined and compared to monolithic systems. During the thesis, the created design artefact was designed, presented, and evaluated.

The monolithic architecture is a fundamental deficiency in the existing system, which is why making changes to the existing system requires a significant amount of development effort and time. As a result, the current monolithic solution should not be the focus of improvement, and the thesis proposed design artefact advocates for the development of a new system as a collection of microservices, in which the application is divided into manageable entities that can be hosted in the cloud. By converting and migrating the current solution to a microservice architecture, the application could be delivered in independently operating

containers, allowing for greater development flexibility, increased application comprehension, system scalability, and rapid system growth.

# REFERENCES


1. Sommerville, I. (2016) Software engineering. Tenth edition. Boston: Pearson.
2. Khadka, Ravi, Belfrit V. Batlajery, Amir M. Saeidi, Slinger Jansen, and Jurriaan Hage. "How Do Professionals Perceive Legacy Systems and Software Modernization?" In Proceedings of the 36th International Conference on Software Engineering, 36–47. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014. https://doi.org/10.1145/2568225.2568318.

3. Kapor, Mitchell. "A Software Design Manifesto." In Bringing Design to Software, 1–6. New York, NY, USA: Association for Computing Machinery, 1996. http://doi.org/10.1145/229868.230026.

4. Hevner, Alan, and Samir Chatterjee. Design Research in Information Systems. Vol. 22. Integrated Series in Information Systems. Boston, MA: Springer US, 2010. https://doi.org/10.1007/978-1-4419-5653-8.

5. Kleebaum, Anja, Marco Konersmann, Michael Langhammer, Barbara Paech, Michael Goedicke, and Ralf Reussner. "Continuous Design Decision Support." In Managed Software Evolution, edited by Ralf Reussner, Michael Goedicke, Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, and Lukas Märtin, 107–39. Cham:

6. Lapuz, Neil, Paul Clarke, and Yalemisew Abgaz. "Digital Transformation and the Role of Dynamic Tooling in Extracting Microservices from Existing Software Systems." In Systems, Software and Services Process Improvement, edited by Murat Yilmaz, Paul Clarke, Richard Messnarz, and Michael Reiner, 301–15. Communications in Computer and Information Science. Cham: Springer International Publishing, 2021. https://doi.org/10.1007/978-3-030-85521-5_20.

7. Reussner, Ralf, Michael Goedicke, Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, and Lukas Märtin. “Introducing Managed Software Evolution.” In Managed Software Evolution, edited by Ralf Reussner, Michael Goedicke, Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, and Lukas Märtin, 3–8. Cham: Springer International Publishing, 2019. https://doi.org/10.1007/978-3-030-13499-0_1.

8. Wieringa R.J. (2014) Implementation Evaluation and Problem Investigation. In: Design Science Methodology for Information Systems and Software Engineering. Springer, Berlin, Heidelberg. https://doi-org.ezproxy.cc.lut.fi/10.1007/978-3-662-43839-8_5

9. Megargel, Alan, Venky Shankararaman, and David K. Walker. “Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example.” In Software Engineering in the Era of Cloud Computing, edited by Muthu Ramachandran and Zaigham Mahmood, 85–108. Cham: Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-33624-0_4.

10. Assunção, Wesley K. G., Thelma Elita Colanzi, Luiz Carvalho, Alessandro Garcia, Juliana Alves Pereira, Maria Julia de Lima, and Carlos Lucena. “Analysis of a Many-Objective Optimization Approach for Identifying Microservices from Legacy Systems.” Empirical Software Engineering 27, no. 2 (March 2022): 51. https://doi.org/10.1007/s10664-021-10049-7.

11. Ramachandran, Muthu, and Zaigham Mahmood, eds. Software Engineering in the Era of Cloud Computing. Computer Communications and Networks. Cham: Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-33624-0.

12. Rajasekharaiah, Chandra. Cloud-Based Microservices: Techniques, Challenges, and Solutions. Berkeley, CA: Apress, 2021. https://doi.org/10.1007/978-1-4842-6564-2

13. Balalaie, Armin, Abbas Heydarnoori, and Pooyan Jamshidi. “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture.” IEEE Software 33, no. 3 (May 2016): 42–52. https://doi.org/10.1109/MS.2016.64.

14. Chondamrongkul, Nacha, Jing Sun, and Ian Warren. “Formal Software Architectural Migration Towards Emerging Architectural Styles.” In Software Architecture, edited by Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, and Olaf Zimmermann, 12292:21–38. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-58923-3_2.

15. Mathai, Alex, Sambaran Bandyopadhyay, Utkarsh Desai, and Srikanth Tamilselvam. “Monolith to Microservices: Representing Application Software through Heterogeneous GNN.” ArXiv:2112.01317 [Cs], December 17, 2021. http://arxiv.org/abs/2112.01317.

16. Mazlami, Genc, Jürgen Cito, and Philipp Leitner. “Extraction of Microservices from Monolithic Software Architectures.” In 2017 IEEE International Conference on Web Services (ICWS), 524–31, 2017. https://doi.org/10.1109/ICWS.2017.61.

17. Preston-Werner, Tom. “Semantic Versioning 2.0.0.” Semantic Versioning. Accessed November 20, 2021. https://semver.org/.

18. Karabey Aksakalli, Işıl, Turgay Çelik, Ahmet Burak Can, and Bedir Tekinerdoğan. “Deployment and Communication Patterns in Microservice Architectures: A Systematic Literature Review.” Journal of Systems and Software 180 (October 2021): 111014. https://doi.org/10.1016/j.jss.2021.111014.

19. Jones, David, and Shirley Gregor. “The Anatomy of a Design Theory.” Journal of the Association for Information Systems 8.5 (2007): 312–335. Web.

20. Hevner, Alan R., Salvatore T. March, Jinsoo Park, and Sudha Ram. “Design Science in Information Systems Research.” MIS Quarterly 28, no. 1 (March 2004): 75–105. https://doi.org/10.2307/25148625.

21. Peffers, Ken, Marcus Rothenberger, and Bill Kuechler, eds. Design Science Research in Information Systems. Advances in Theory and Practice. Vol. 7286.

Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. https://doi.org/10.1007/978-3-642-29863-9.

22. Brocke, Jan vom, Alan Hevner, and Alexander Maedche, eds. Design Science Research. Cases. Progress in IS. Cham: Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-46781-4.

23. Lee, Allen S., Manoj Thomas, and Richard L. Baskerville. "Going Back to Basics in Design Science: From the Information Technology Artefact to the Information Systems Artefact." Information Systems Journal 25, no. 1 (January 2015): 5–21. https://doi.org/10.1111/isj.12054.

24. Johannesson, Paul, and Erik Perjons. An Introduction to Design Science. Cham: Springer International Publishing, 2014. https://doi.org/10.1007/978-3-319-10632-8.

25. Gregor, S. & Hevner, A. R. (2013) Positioning and Presenting Design Science Research for Maximum Impact. MIS quarterly. [Online] 37 (2), 337–355.

26. Sonnenberg, Christian, and Jan vom Brocke. Evaluation Patterns for Design Science Research Artefacts. Practical Aspects of Design Science, 2012. https://doi.org/10.1007/978-3-642-33681-2_7.

27. Sein, Maung K., Ola Henfridsson, Sandeep Purao, Matti Rossi, and Rikard Lindgren. Action Design Research. MIS Quarterly 35, no. 1 (March 2011): 37–56. https://doi.org/10.2307/23043488.

28. Velichety, Srikar, and Sudha Ram. Common Citation Analysis and Technology Overlap Factor: An Empirical Investigation of Litigated Patents Using Network Analysis, 2012. https://doi.org/10.1007/978-3-642-29863-9_21.

29. Knutas, Antti, Zohreh Pourzolfaghar, and Markus Helfert. "The Role and Impact of Descriptive Theories in Creating Knowledge in Design Science." In Computer-Human Interaction Research and Applications, edited by Andreas Holzinger, Hugo

Plácido Silva, and Markus Helfert, 654:90–108. Communications in Computer and Information Science. Cham: Springer International Publishing, 2019. https://doi.org/10.1007/978-3-030-32965-5_5.

30. Jain, R. (1991) The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling. New York: Wiley.

31. Kneuper, Ralf. Software Processes and Life Cycle Models: An Introduction to Modelling, Using and Managing Agile, Plan-Driven and Hybrid Processes. Cham: Springer International Publishing, 2018. https://doi.org/10.1007/978-3-319-98845-0.

32. Dumas, Marlon, and Dietmar Pfahl. "Modeling Software Processes Using BPMN: When and When Not?" In Managing Software Process Evolution, edited by Marco Kuhrmann, Jürgen Münch, Ita Richardson, Andreas Rausch, and He Zhang, 165–83. Cham: Springer International Publishing, 2016. https://doi.org/10.1007/978-3-319-31545-4_9.

33. Chaudron, Michel R. V., Werner Heijstek, and Ariadi Nugroho. "How Effective Is UML Modeling?: An Empirical Perspective on Costs and Benefits." Software and Systems Modeling 11, no. 4 (2012): 571–80. https://doi.org/10.1007/s10270-012-0278-4.

34. Yongchareon, Sira, Chengfei Liu, and Xiaohui Zhao. "Reusing Artefact-Centric Business Process Models: A Behavioral Consistent Specialization Approach." Computing 102, no. 8 (August 2020): 1843–79. https://doi.org/10.1007/s00607-020-00798-6.

35. Estañol, Montserrat, Jorge Munoz-Gama, Josep Carmona, and Ernest Teniente. "Conformance Checking in UML Artefact-Centric Business Process Models." Software & Systems Modeling 18, no. 4 (August 2019): 2531–55. https://doi.org/10.1007/s10270-018-0681-6.

36. Chinosi, Michele, and Alberto Trombetta. “BPMN: An Introduction to the Standard.” Computer Standards & Interfaces 34, no. 1 (January 2012): 124–34. https://doi.org/10.1016/j.csi.2011.06.002

37. Frey, Frank J., Carsten Hentrich, and Uwe Zdun. “Capability-Based Service Identification in Service-Oriented Legacy Modernization.” In Proceedings of the 18th European Conference on Pattern Languages of Program - EuroPLoP ’13, 1–12. Irsee, Germany: ACM Press, 2015. https://doi.org/10.1145/2739011.2739021.

38. Fritzsch, Jonas, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. “From Monolith to Microservices: A Classification of Refactoring Approaches.” In Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment, edited by Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, 128–41. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019. https://doi.org/10.1007/978-3-030-06019-0_10.

39. Mazzara, Manuel, Nicola Dragoni, Antonio Bucchiarone, Alberto Giaretta, Stephan T. Larsen, and Schahram Dustdar. “Microservices: Migration of a Mission Critical System.” IEEE Transactions on Services Computing 14, no. 5 (September 2021): 1464–77. https://doi.org/10.1109/TSC.2018.2889087.

40. Venable, John, Jan Pries-Heje, and Richard Baskerville. “FEDS: A Framework for Evaluation in Design Science Research.” European Journal of Information Systems 25, no. 1 (January 1, 2016): 77–89. https://doi.org/10.1057/ejis.2014.36.

41. Behkamal, Behshid, Mohsen Kahani, and Mohammad Kazem Akbari. “Customizing ISO 9126 Quality Model for Evaluation of B2B Applications.” Information and Software Technology 51, no. 3 (March 2009): 599–609. https://doi.org/10.1016/j.infsof.2008.08.001.

42. Yli-Huumo, Jesse, Andrey Maglyas, and Kari Smolander. “The Effects of Software Process Evolution to Technical Debt—Perceptions from Three Large Software Projects.” In Managing Software Process Evolution, edited by Marco Kuhrmann, Jürgen Münch, Ita Richardson, Andreas Rausch, and He Zhang, 305–27. Cham: Springer International Publishing, 2016. https://doi.org/10.1007/978-3-319-31545-4_15.