UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Siim-Morten Ojasalu

Reconstruction of a monolithic application within Maksekeskus AS

Master's Thesis (30 ECTS)

Supervisor: Dietmar Pfahl, PhD
Co-supervisor: Reelyka Läheb, MSc

Tartu 2023

Reconstruction of a monolithic application within Maksekeskus AS

Abstract:
Choosing the correct software architecture can increase the efficiency, performance, security, modifiability, and many more aspects of a system. It can also help save resources due to fast developments or easy deployments for example. Microservices and monolithic architecture are two very relevant and possible variants of software architecture. The goal of this thesis is to find out, which solution would be best suited for payment solutions provider AS Maksekeskus' WooCommerce module. The compared candidate architectures are microservices, monolithic architecture, and a hybrid solution which is a combination of the first two.
For evaluating the architectures, this thesis uses the software architecture analysis method, in short SAAM. It is a scenario-based evaluation that has six analytical steps in order to evaluate and find the best architecture possible. As a result of this analysis, it is concluded that the hybrid architecture consisting of a few microservices and a monolithic part is the best solution for Maksekeskus' WooCommerce module. Due to this evaluation, a tracking link service is also extracted from the original monolithic structure and set up as an independent microservice.
This thesis provides a good example of an architectural solution where both the microservices and a monolith are included. These findings help make the modules that Maksekeskus develops more resource efficient and provide a foundation for further analysis in the field of moving software from on-premise to cloud and conducting architectural analysis.

Keywords:
Microservices, monolithic architecture, software architecture analysis method, cloud computing

CERCS: P170 Computer science, numerical analysis, systems, control

Monoliitse arhitektuuri rekonstrueerimine Maksekeskus AS näitel

Lühikokkuvõte:
Tarkvaraarenduses õige arhitektuuri valimine võib aidata tõsta jõudlust, efektiivsust, turvalisust ja muid olulisi tegureid. Lisaks võib see aidata säästa ressursse tänu parandatud arenduskiirusele ja kiirele juurutamisele. Mikroteenused ja monoliitne arhitektuur on ühed tuntumad arhitektuurid tarkvaraarenduses. Magistritöö eesmärgiks on välja selgitada, milline lahendus sobiks kõige paremini makse- ja tarnemeetodite vahendaja AS Maksekeskuse WooCommerce'i mooduli jaoks. Töös võrreldakse mikroteenuseid, monoliitset arhitektuuri ja nende kahe ühildamisel saadud hübriidarhitektuuri.
Arhitektuuriliste lahenduste hindamiseks kasutatakse töös tarkvaraarhitektuuri analüüsi meetodit (*software architecture analysis method*). Mainitud meetod kasutab analüüsimiseks stsenaariumeid, mille abiga on võimalik hinnata vastavate lahenduste sobivust kuue välja töötatud sammu kaudu. Analüüsi tulemusena jõutakse töös järeldusele, et mikroteenustest ja monoliitsest arhitektuurist koosnev hübriidlahendus on parim Maksekeskuse loodud WooCommerce'i mooduli jaoks. Uue arhitektuuri valimise tulemusena eraldatakse töö käigus ka hetkelisest monoliidist üks teenus ja luuakse sellest eraldiseisev mikroteenus, mis pakub klientidele tarnete jälgimise võimalust.
Käesolev töö on hea näide olukorrast, millal on parimaks lahenduseks nii mikroteenustest kui ka monoliidist koosnev tarkvaraarhitektuur. Töös välja toodud tulemused ja praktiline lahendus aitavad muuta Maksekeskuse loodud mooduli veelgi tõhusamaks ja üleüldise ressursihalduse efektiivsemaks. Lisaks aitavad resultaadid panna aluse edaspidistele töödele, mis analüüsivad arhitektuurilisi lahendusi ning võimalusi liikuda tarkvaraga pilve.

Võtmesõnad:
Mikroteenused, monoliitne arhitektuur, tarkvaraarhitektuuri analüüs, pilvetöötlus

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Acknowledgements

I would like to thank my supervisors Reelyka Läheb and Dietmar Pfahl for their invaluable guidance, encouragement, and support throughout the course of writing this thesis. Their insights and expertise have been instrumental in shaping this thesis, and I am deeply appreciative of that.
I would also like to express my sincere gratitude to Priit Leet, whose initial proposal of the topic and following practical guidance provided the crucial foundation for this work. Without him, this thesis would not have been possible.
Lastly, I would like to thank Maksekeskus for the unique opportunity to lead this project. I am truly grateful for all the assistance and help that I have received while writing this thesis.

Contents

1 Introduction **7**
- 1.1 Motivation . . . 7
- 1.2 Thesis Scope and Objectives . . . 8
- 1.3 Thesis Structure . . . 8

2 Background **9**
- 2.1 Monolithic Architecture . . . 9
  - 2.1.1 Advantages of Monolithic Architecture . . . 9
  - 2.1.2 Disadvantages of Monolithic Architecture . . . 10
- 2.2 Microservices Architecture . . . 11
  - 2.2.1 Advantages of Microservices . . . 11
  - 2.2.2 Disadvantages of Microservices . . . 12
  - 2.2.3 Implementing Microservices . . . 13
- 2.3 Software Architecture Analysis Methods . . . 13
  - 2.3.1 Architecture-Level Modifiability Analysis . . . 14
  - 2.3.2 Software Architecture Analysis Method . . . 15
  - 2.3.3 Architecture Tradeoff Analysis Method . . . 16
- 2.4 Overview of Maksekeskus AS . . . 18
- 2.5 Description of the Current WooCommerce Module Architecture . . . 18
  - 2.5.1 Advantages of the Current WooCommerce Module Architecture 19
  - 2.5.2 Disadvantages of the Current WooCommerce Module Architecture 20
- 2.6 Related Work . . . 23

3 Method **24**
- 3.1 Software Architecture Analysis Method (SAAM) . . . 24
  - 3.1.1 Developing Scenarios . . . 25
  - 3.1.2 Description of Current and Candidate Architectures . . . 25
  - 3.1.3 Classify and Prioritize Scenarios . . . 27
  - 3.1.4 Scenario Evaluation . . . 27
  - 3.1.5 Scenario Interactions . . . 28
  - 3.1.6 Overall Evaluation . . . 29
- 3.2 Extracting Microservices . . . 29

4 Results **31**
- 4.1 Architecture Analysis with SAAM . . . 31
  - 4.1.1 Created Scenarios . . . 31
  - 4.1.2 Description of Current and Candidate Architectures . . . 32
  - 4.1.3 Scenario Classification and Prioritization . . . 39
  - 4.1.4 Scenario Evaluations . . . 40

4.1.5 Scenario Interactions 45
4.1.6 Overall Evaluation 48
4.2 Partial Migration To Microservices 49
4.2.1 Microservice Description 49
4.2.2 Changes in the WooCommerce Module Workflow 52
4.2.3 Service Advantages 58

5 Discussion **60**
5.1 Interpretation of the Results 60
5.2 Shortcomings of the Proposed System 61
5.3 Lessons Learnt and Points of Improvement 62

6 Conclusion **64**

References **67**
I. Licence 68

1 Introduction

Maksekeskus AS is an Estonian company that offers payment solutions for both online and on-site stores. For online stores, the company offers payment and shipping solutions for many different e-commerce platforms. These allow the store owner to easily add many choices of payment to their store. This adds variety and ease of use for the customer, as they can just easily pick whichever payment method is the most suitable for them. However as there are many different e-commerce platforms, each of them requires its own solution for a payment and shipping module. Therefore Maksekeskus currently has multiple modules which all do the same thing but are developed separately due to being on different platforms like WooCommerce, Magento, or PrestaShop.

1.1 Motivation

Switching from monolithic applications to microservices and migrating software from on-premises to the cloud have become recent trends. With big companies such as Netflix, Amazon, and eBay moving from big monolithic architecture to a fine-grained microservice-based architecture, many other companies have followed suit [CLL17]. This provides a motive for all companies to consider changing their architecture. If the new architectures could provide better performance, simplify the deployments, and further developments then these migrations could offer companies a lot of benefits. As the modules developed by Maksekeskus are somewhat inefficient and at times difficult to manage, then moving on to another more beneficial architecture could be of great help. It could speed up future developments, allow for much quicker deployments in order to meet the customers' needs, and set the base for efficiently written and maintained code. What is more, as all the modules that Maksekeskus develops are aimed to solve the same problems, then creating microservices that could serve each of these modules would be highly beneficial. That would eliminate the need to write duplicate code in order to integrate a new feature or modification into all of the modules. Overall, there is a lot to gain from an architectural rework and this is the motivation behind this thesis.
Similar papers and theses have already been written and are covered in section 2.6 but the vast majority have concluded that switching from a monolithic structure to microservices is beneficial. However, very few papers seem to arrive at the conclusion that the original monolithic architecture is better. Therefore this thesis could provide another vital example where microservices are not favorable or confirm the trend and prove yet again why microservices are very highly valued. In the context of theoretical knowledge and examples in the real world, this thesis could also provide the necessary steps in order to successfully integrate a new architecture, based on the analysis of the old one.

1.2 Thesis Scope and Objectives

The goal and contribution of this thesis are to research and analyze the current software architecture for Maksekeskus AS and find out, which architectural solution would fit the company best. However, as Maksekeskus provides payment solutions to different platforms then analyzing them all would be simply too much work and out of scope for this thesis. Therefore the main focus is on the Maksekeskus' WooCommerce module, which is an add-on plugin to the popular e-commerce software WooCommerce on WordPress. However, as mentioned in section 1.1, all the modules in principle solve the same exact problem. Because of that, providing a new architecture is not only beneficial for the WooCommerce module but could help all the other developments as well.
In addition to restricting the scope of this thesis to only cover the WooCommerce module, the new architectures that are analyzed are also restricted to microservices and monolithic architectures. There are other options as well, like serverless architecture for example. However, these are the two main types of software architectures, and including more would just widen the scope too much without being very beneficial. This was also agreed upon with the company itself.
Should microservices prove to be beneficial, then another goal is to partially integrate the new architecture and set up one microservice. This can provide an example and showcase, that the architectural changes are being made and put into use. Reworking the whole system would take a lot more time and is not possible within the time frame of this thesis.

1.3 Thesis Structure

For a quick overview and ease of reading, the overall structure of the thesis is briefly described in this section. Section 2 provides a theoretical base for the different solutions that are considered, including monoliths and microservices. The section also describes the current system's advantages and disadvantages from a developer's viewpoint and includes an overview of the used literature and similar works.
Section 3 covers the analysis methods of the current WooCommerce module for the WooCommerce e-store solution. It also includes choosing a software architecture analysis method in order to evaluate the possible architectures and it also provides the necessary steps that need to be taken in order to integrate a new architecture and set up microservices. That will be necessary if the architecture analysis concludes that a microservices architecture is better than the current monolithic one.
Section 4 presents the results of the software analysis and if a new architecture is chosen, section 4 will also present the steps that were taken in order to set up a microservice. All the steps that are presented in section 3 for the software analysis will have the results presented in section 4.
The concluding discussion is presented in section 5. It covers the possible shortcomings

of the implemented solution, provides valuable lessons learned that could prevent mistakes for others, interprets the results and compares them to the theoretical background analysis, and overall compares the completed work to the goals set.

2 Background

This section covers the used literature and gives an overview of all the technologies that are considered. Furthermore, it compares the differences in architecture and overall software building and its maintenance between monolithic applications and applications built using microservices. Of course, there are more possibilities, and monoliths themselves could be defined at lower levels as there are many ways to build a monolith. However, in order to avoid vagueness and stay on topic, only microservices and monoliths are covered as mentioned in the scope of the thesis. Furthermore, it also describes the current software architecture of Maksekeskus' WooCommerce module for the WordPress platform and gives an overview of the problems and advantages. Lastly, section 2.6 covers the related work and read literature that is used in this thesis.

2.1 Monolithic Architecture

Monolithic architecture for software applications could be considered as a single system, which contains everything it needs within itself. It could also be seen as the starting standard for new software projects, as concluded by Kalaske et al. [KMM17]. The architecture itself contains many different layers like the UI, business logic, and data access. Newman considers monolithic applications to be units of deployment [New19]. Whenever all of the functionality of the system has to be deployed as a whole single unit together, then that application can be considered a monolith. Furthermore, he describes that there are multiple types of monoliths and brings out some examples like a single-process monolith and a distributed monolith. The latter might be a bit confusing as distribution is one key factor in microservices. However, the distributed monolith is described as consisting of multiple services, but they all have to be deployed together due to some reason and therefore are defined as a monolith. The single-process monolith has a very simple structure with one database that can be seen in Figure 1. When splitting the single process into multiple different modules which are still deployed together, the resulting structure is the modular monolith on Figure 2.

2.1.1 Advantages of Monolithic Architecture

Monolithic applications are usually beneficial when the complexity and size of the application do not grow too big. Gos et al. determined two main advantages of monolithic applications: ease of development and simple deployment [GZ20]. L. D. Lauretis has

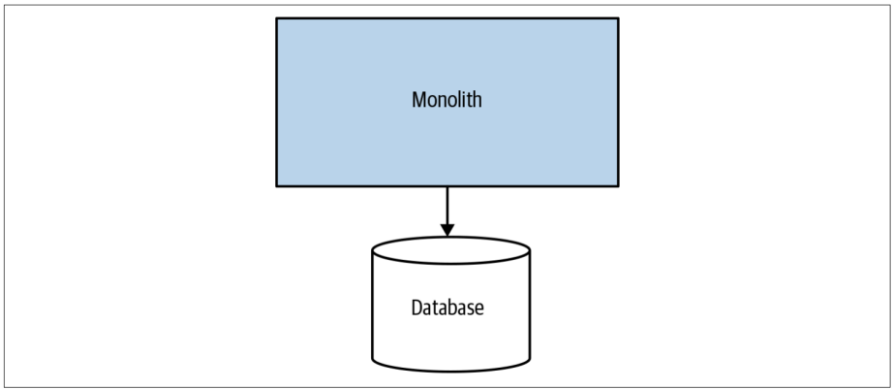

Figure 1. A single process monolith [New19]

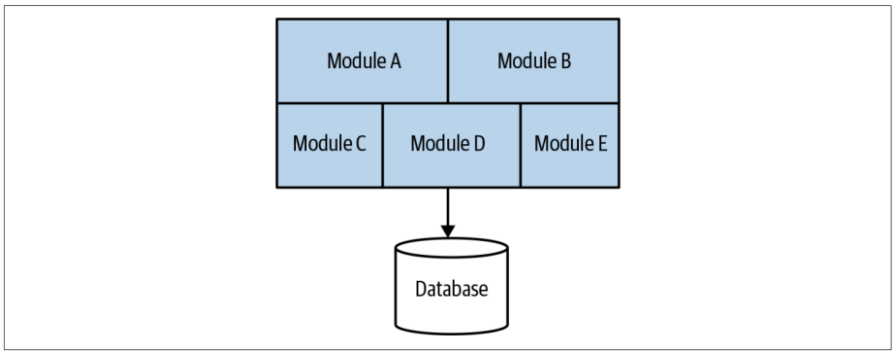

Figure 2. A single process monolith divided into multiple modules [New19]

concluded, that the ease of scaling and testing are also benefits within a limited-size monolithic architecture [DL19]. These two main advantages make it look like the monolithic architecture does not have much to offer, but regarding small applications, it should be a clear choice in most cases. G. Blinowski has determined, that the simplicity of a system might be the strongest advantage of a monolithic application [BOP22]. G. Blinowski describes that testing, deploying, or monitoring a monolithic architecture is easier and another benefit is that all the data could be kept in a single database, which removes the need to synchronize it between different services [BOP22].

2.1.2 Disadvantages of Monolithic Architecture

Such benefits can however disappear quite quickly when the size of the system gets too big. The initial analysis of the Maksekeskus' WooCommerce module in section 2.5 describes that, when the system gets bigger, it becomes increasingly more difficult

to manage. Development can become unorganized due to mistakes or just a lack of understanding. This in turn accelerates the overall disorder. Mistakes could bring down the whole system instead of just parts of it as the monolithic application is deployed as a whole and works as one complete system.
G. Blinowski similarly points out that when an application grows and complex code is added, it makes the overall system more difficult to understand and complexity can cause unwanted or unexpected behavior [BOP22]. G. Blinowski also concludes that a monolithic application and its growth can cause modularity to drop and changes to certain parts of the system to start affecting other parts as well [BOP22]. What is more, G. Blinowski determined that a bigger application requires more developers within a monolithic system and this can lead to ineffective and unequal distribution of work among the developers, which in turn causes the overall productivity to drop [BOP22]. Furthermore, while updating an application with a monolithic architecture, the whole system must be redeployed, and consequently, its availability may be compromised. Lastly, it is harder to scale a monolithic application. Even if some small part of the system should be allocated more resources, it must be given to the whole application and therefore the cost of resources is much higher.

2.2 Microservices Architecture

Software built on microservices is divided, into multiple small services which are distributed and isolated from each other. Kalaske et al. defined microservices as smaller parts of a system that adhere to the single responsibility principle [KMM17]. Kalaske et al. describe these services as components that serve only one function in a bigger application and therefore create more clarity on where new developments should be made [KMM17]. These structural boundaries allow the system to be loosely coupled and consequently, independent changes and deployments to certain systems do not cause downtime or the need for new deployments in other services. Al-Debagy et al. opted for a definition that describes microservices as an approach to development [ADM18]. These multiple smaller services which make up a larger application, communicate with each other through HTTP API for example and each of them runs on its own process. Newman provides an important addition to these definitions as he states, that the boundaries of microservices could be opinionated and not unambiguously understood [New19]. However, the importance of independent deployability remains in his definition.

2.2.1 Advantages of Microservices

There are numerous reasons why organizations choose microservices over monolithic architecture. Taibi et al. determined multiple advantages, which include for example the free choice of technology and programming language for each service, the possibility to scale each independent service on its own, and as they usually are quite small in size, they

are very simple to maintain and develop [TLPJ17]. L. D. Lauretis has concluded that breaking a monolithic application into independent and self-deployable services allows for easier development changes and this in turn increases the overall code knowledge and maintainability of it [DL19].
Moreover, Taibi et al. established that fault tolerance within microservices architecture is better, as only a single service out of a bigger system is affected, unlike in a monolithic architecture, where faults can cause the whole application to break. Lastly, Taibi et al. concluded that microservices support the IDEAL properties due to being native to the cloud [TLPJ17]. These properties are isolation, distribution, elasticity, automation of management, and loosely coupled parts [TLPJ17].

2.2.2 Disadvantages of Microservices

One of the main disadvantages of microservices is the complexity that comes with the distributed architecture and individual services. Kalaske et al. explained that the distribution of multiple individual services creates additional points of failure [KMM17]. Therefore it is important to consider how such failures are dealt with. Kalaske et al. conclude, that whenever an application with a monolithic architecture stops working, nothing works, but if any individual services fail, the others will still work and may try using the failed one [KMM17]. This kind of complex distribution may also create difficulties in trying to grasp and understand how the microservices are related to each other [atl23b].
Furthermore, perfecting the communication between isolated services requires planning and can prove to be difficult. Kalaske et al. explain, that services that are too fine-grained may cause issues and a better practice would be to include any complexity and data modification in the service and refrain from modifying the data in the message pipes [KMM17].
What is more, another big challenge in adopting microservices is the cultural and technical changes that it requires inside the organization. Kalaske et al. determined that organizational changes are needed in order to adapt - developers should take ownership of the microservices that they develop and DevOps would be useful for continuous integration, continuous delivery, and continuous monitoring [KMM17]. Communication between DevOps teams and developer teams becomes more important in order to avoid such confusion and make locating the correct people and information simpler [atl23b].
Lastly, a different kind of disadvantage compared to additional complexity is the cost of microservices. Currently, Maksekeskus' WooCommerce module is an on-premises software that is located in the merchant's e-store server. However, migrating certain parts to the cloud immediately adds an additional expense that needs to be paid and maintained. These services require their own testing and production environments, monitoring tools and any other hosting infrastructure [atl23b].

2.2.3 Implementing Microservices

Implementing a new architecture instead of an old one requires planning and careful consideration. This section briefly covers some of the most important steps while trying to create separate services and deploy them to the cloud.
Smaller applications are easier to develop as monoliths, which was covered as an advantage for the monolithic architecture in section 2.1.1. However, as the software codebase gets bigger, things get more complicated and the software architects turn to microservices. But these steps are actually encouraged, as it may be easier to create new microservices while starting from a monolithic architecture [atl23a].
When starting with a new microservice, it has to be considered as a separate product that should be developed by a certain team and shipped independently [atl23a]. Butzin et al. defined the importance of self-containment, as each microservice should contain everything that it needs to fulfill its task independently [BGT16]. Butzin et al. also mention that having no dependencies allows the services to be maintained and scaled individually, without having to make changes elsewhere [BGT16].
Moreover, another important aspect to consider when creating microservices is to identify the parts of the monolithic architecture, that will be converted, how the communication will work, and how the data will be stored [atl23a]. Usually, a monolithic architecture has one single database but with microservices, this is not recommended. That is because certain services may need different schemas or overall make more database queries, which could make the same process much slower for the other components of the system [atl23a]. Butzin et al. recommend that with best practices, microservices should have their own databases [BGT16]. The communication between the microservices should be kept as simple as possible as well in order to prevent complex data transfers and additional points of failure [atl23a].
Lastly, when developing microservices it is important to focus on monitoring their status as well [atl23a]. Butzin et al. determine that each service should have its own health status, that other services can then check [BGT16]. Butzin et al. explain, that this kind of health check can prevent services from trying to access broken ones and making calls that end in a failure [BGT16]. These health checks can also be used for monitoring by the administrators to quickly be alerted whenever there is an issue with any of the microservices.

2.3 Software Architecture Analysis Methods

In order to analyze the existing architecture and new candidates, there are different possible analysis methods. This section introduces three analysis methods and gives a quick insight into their methodologies. Of course, there are many more methodologies for software architecture analysis. The chosen analysis methods seemed to be most relevant for this thesis, provided similar results theoretically and it would be impossible

to compare every possible analysis method within the scope of this thesis. Out of these three, a single method is chosen for the analysis and is described in section 3, and the results of the analysis are presented in section 4.

2.3.1 Architecture-Level Modifiability Analysis

The architecture-level modifiability analysis (ALMA) is used in order to predict maintenance costs, assess risks and select the best architecture out of multiple proposed ones. It helps distinguish certain analysis goals, has very clear assumptions, and brings approaches that can be repeated for performing the included steps. Bengtsson et al. described ALMA as a five-step process [BLBvV04]. These stages are as follows:

1. Set Goal(s)
2. Describe Software Architecture
3. Elicit Scenarios
4. Evaluate Scenarios
5. Interpret Results

The first step in ALMA requires setting the goals for the overall evaluation and analysis. Bengtsson et al. describe that ALMA can be used to predict the costs of changes that are required to modify a system in order to include future developments [BLBvV04]. Furthermore, the goals can include the identification of developments for which the candidate architecture may be inflexible, and overall provide guidance for selecting the optimal architecture [BLBvV04].
The second step in ALMA gives a description of the architecture that is analyzed in the upcoming steps. Lassing et al. concluded that the description itself should be as thorough as possible in order to provide the basis for architecture-level impact analysis and therefore assess the effect of all the scenarios that are created in the following steps on the architecture [LBVVB02].
The third step includes creating a set of scenarios, on which to base the analysis. Bengtsson et al. described that this process requires conducting interviews with certain shareholders and that all these people that are interviewed, should have different kinds of responsibilities in order to get as wide of an understanding of the systems changes as possible [BLBvV04].
The next step requires analyzing the architecture in parallel with the created scenarios. Lassing et al. explain, that this analysis determines the need for changes among the components of the system in order to implement the changes needed for all the scenarios to work [LBVVB02].
Lastly, the final step of ALMA involves making conclusions about the architecture that

was analyzed. Bengtsson et al. describe the step as an interpretation for the results [BLBvV04]. Bengtsson et al. explain, that the goal set in the first step determines how the results of the analysis are understood [BLBvV04]. The results could differ if the goals were for example to assess risks or predict future maintenance requirements.

2.3.2 Software Architecture Analysis Method

Software Architecture Analysis Method or SAAM for short is a scenario-based analysis approach. It is used to evaluate the quality of different aspects like maintainability, performance, and reliability within an application. Babar et al. concluded that the main goal of SAAM is to evaluate and compare the software architecture to certain quality attributes. It can also help identify potential issues in the current architecture that should be removed from the system or at the very least reworked. The methodology consists of multiple steps and is defined a bit differently by various authors. Ionita et al. described six different steps for SAAM [IHO02]. These steps are as follows:

1. Develop Scenarios
2. Describe Architecture(s)
3. Classify and Prioritize Scenarios
4. Individually Evaluate Indirect Scenarios
5. Assess Scenario Interaction
6. Create an Overall Evaluation

Ionita et al. describe the first step as a challenge for developing scenarios that would capture the major uses of the system [IHO02]. The last part is very important because without having scenarios for certain important use cases, the evaluation of the system will be inadequate.
The second step is meant for describing the candidate system. Existing architecture may also be considered a candidate in order to compare it to new architecture solutions. Ionita et al. explain that the architecture should be understandable by all parties and represent the important components of the system [IHO02].
The third step includes the classification of the developed scenarios and ordering them by their priority. Ionita et al. bring out the two scenario classifications that are used: direct scenarios and indirect scenarios [IHO02]. The first ones are supported by the candidate architecture without any changes needed and provide a metric for the evaluation of the different aspects like performance or maintainability. Indirect scenarios require modifications to the architecture in order to be realized. The developed scenarios have to be prioritized in this step as well in order to specify, which of them is more important.

For evaluating indirect scenarios in the fourth step, Dobrica et al. explain that for each scenario the number of changes the system requires has to be counted, and based on that, the cost estimation can be evaluated [DN02]. This provides a basis for analyzing whether certain indirect scenarios should be included in the architecture or if they require too many changes then they can be left out.
Assessing scenario interactions in the fifth step requires working through each scenario individually. Ionita et al. describe that if two scenarios require changes to the same component or multiple components then the scenarios interact with each other [IHO02]. This means that the affected components should be changed or split into smaller components in order to avoid the overlap between the scenarios.
For the last step, Ionita et al. explain that each scenario must be weighted by its relevance and importance to the system. Dobrica et al. describe that this kind of weighting can be used to evaluate the overall score of an architecture [DN02]. If there are multiple candidate architectures then this score can determine which one of them would best suit the business requirements and provide the best possible performance.

2.3.3 Architecture Tradeoff Analysis Method

The Architecture Tradeoff Analysis Method (ATAM) is a software analysis method that is focused on the tradeoffs by improving or modifying certain quality attributes. These may include attributes like performance, modifiability, security, and so forth. Kazman et al. determined that often when one of these attributes is modified or more often improved, it has an effect on the other ones as well [KKB+98]. This could mean that improvements in one area can cause issues in another or even multiple areas. ATAM aims to analyze the risks, mitigate any shortcomings, and overall help conclude, whether a planned architecture will meet all the demands and set requirements. Kazman et al. determined that ATAM has evolved from the Software Architecture Analysis Method, which was briefly described in the subsection 2.3.2 [KKB+98]. Nord et al. explained that while ATAM provides the possibility of understanding the technical tradeoffs of certain design elements in the architecture, it does not provide guidelines for perceiving the economic tradeoffs [NBC+03].
Since ATAM has evolved out of SAAM, it has a similar structure and the whole process can be described in multiple steps. Nord et al. described these steps as follows [NBC+03]:

1. Present the ATAM
2. Present Business Drivers
3. Present Architecture
4. Identify Architectural Approaches
5. Generate Quality Attribute Utility Tree

6. Analyze Architectural Approaches
7. Brainstorm and Prioritize Scenarios
8. Analyze Architectural Approaches
9. Present Results

While ATAM may have emerged from SAAM, it contains very few steps that remind the steps in SAAM. The first step of ATAM is quite straightforward and simple. Nord et al. explained, that during this step, the process of ATAM is introduced and defined to different shareholders who are related to the project [NBC+03]. During this step, the expected outputs are also described.
The second step in ATAM includes a presentation that gives an overview of the system from the aspect of the business. Nord et al. described the presentation as a way to describe the business environment and its requirements, set the business constraints and explain the technical constraints, and lastly introduce business needs and determine the quality attributes based on these [NBC+03].
The third and fourth steps are similar and related so they can be described together. Kazman et al. explained that in order to analyze with ATAM, different architectures need to be presented [KKB+98]. This gives the possibility of analyzing the tradeoffs of the proposed solutions in future steps. When the architecture options are ready and thought through, they are introduced to the different shareholders via a presentation like the business overview in the second step.
The fifth step requires an evaluation team to gather the most important quality attributes and their requirements. Nord et al. explain, that these requirements have to be prioritized and defined as scenarios [NBC+03]. Just like in SAAM, these scenarios should cover different aspects of the system in order to be more effective.
Analyzing the architectural approaches is duplicated for steps six and eight. According to Nord et al., these steps include the examination and processing of scenarios according to the studied architecture [NBC+03]. A lead architect will then explain how the scenarios are supported by each of the proposed architectures and what could be the deficits. The changes required to fulfill a scenario are documented and analyzed in order to determine their scale with regard to the currently processed architecture. The sixth and eighth steps are very similar and contain basically the same processes.
The seventh step between the analyzing architectural approaches steps has the task of modifying the scenario list. Nord et al. describe that during this step, the evaluation team picks out the most important scenarios and carries them forward to the next step, leaving some unimportant ones out [NBC+03]. This means that steps six and eight differ by the number of scenarios.
The last step concludes all the steps of ATAM and presents the information that was collected. Nord et al. also bring out, that this step may include any alternative architectures that could have been generated within the analysis [NBC+03].

2.4 Overview of Maksekeskus AS

Maksekeskus AS provides payment solutions and shipping options for stores and e-commerce platforms in the form of modules and is one of the leading service providers in the Baltic states. The company provides payment solutions like bank links, credit card payments, pay later options, and even a custom-made electronic gift card that can be used in all stores that use the company's payment solutions. These are available in all Baltic countries, including Estonia, Latvia, and Lithuania. At the time of writing, the supported platforms are WooCommerce on the WordPress hosting solution, Magento, PrestaShop, OpenCart, Shopify, Voog, and custom solutions. All these require separate developments, as different platforms have different requirements. Overall their purpose is the same though. This means that essentially there is a lot of duplicate code. Most parts of the system could be made universal and therefore provide a solution for all the existing platforms and a base for integrating new ones.
From a technological point of view, the modules like Maksekeskus' WooCommerce module are developed in PHP and Javascript. HTML and CSS are also in use. However, all the modules use an API that is developed in Java and outsourced to a partner company. Therefore in-house development only includes working on the plugins and the team size for doing that is four developers at the time of writing. This means that the developers may have to maintain multiple platforms and are not able to specialize in one specific plugin. As the company is over 10 years old then there is quite a bit of legacy code and parts of software, that are outdated and poorly configured. What is more, when there are multiple issues in different modules, then certain ones have to get priority due to limited resources.

2.5 Description of the Current WooCommerce Module Architecture

Currently, all the modules Maksekeskus provides for different platforms aim to resolve one problem. And that is the complexity of adding different payment solutions to e-commerce stores. Of course, there are other benefits as well, for example, shipping options like Omniva, DPD, Smartpost, or LP Express services. But mainly Maksekeskus wants to help eliminate the need to sign multiple different contracts with banks in order to be able to use bank links for payment options. This is where the developed modules come in. From the possible selection of e-commerce platforms, a customer can set up their store, sign a contract with Maksekeskus, and provide their own customers with a wide variety of payment options, starting with bank links and ending with credit card payments. Subsections 2.5.1 and 2.5.2 are based on the developer-side analysis of the WooCommerce module. As there currently lacks any kind of overview of the module's architecture due to loss of data, then the advantages and disadvantages are written on the basis of experience developing the module itself. The software architecture analysis is introduced in section 3.

2.5.1 Advantages of the Current WooCommerce Module Architecture

The current architecture possesses advantages that must be considered when evaluating the possibilities of integrating a new software architecture for the WooCommerce module. While it is important to fix issues with a new architectural solution, it also has an impact on all the positive sides as well. For the best possible future solution, it is necessary to distinguish the current pros of the system and they are as follows:

1. Common Technology Throughout the Application
2. Easier Debugging
3. Simple Communication

All these positive sides should be considered when designing a new system. Of course, new solutions may provide even more advantages than the existing solution, but the current advantages are a good basis for a new system. The main advantages are brought out and described in more detail in the following paragraphs.

Common Technology Throughout the Application
The usage of the same technologies throughout the whole application is an advantage due to Maksekeskus' small development team. This is unlikely to change, even if an architectural rework is to come. The development team has been put together exactly due to this reason. Changes would be required among developers or some intensive training needs to be provided in order to switch or add additional technologies like a different programming language. For the company, this does not make much sense unless it will be profitable in the near future. Currently having a certain set of technologies and developing languages is an advantage due to the small team of developers.

Easy Debugging
Whenever there are errors or bugs in the code, they have to be found and fixed as quickly as possible. Although having a monolithic application might make finding these errors more difficult, it is still quite easy to debug thanks to different debugging tools. As the whole application is together, knowing the flow and adding breakpoints helps locate problems and issues quickly and without too much trouble. Also being familiar with this kind of system is a benefit in itself. Learning how to debug and find problems quickly will take time with a new architecture. While the current system is getting complex and more difficult to manage, it has provided a simple way for debugging, which makes writing and refactoring code easier.

Simple Communication
In the WooCommerce module currently, all the code is contained within a single application. This means that communication between different parts of the application is simple

and efficient because all components are running in the same environment. Developers can make direct function calls between separate classes of the system without worrying about network latency, communication protocols, or message serialization. This also makes it easier to coordinate between different parts of the application and simplifies sharing data between components, as they are all running within the same environment. This makes it simpler to develop and maintain the application, as there is less overhead and complexity involved in managing all of the components and figuring out the correct communication methods.

2.5.2 Disadvantages of the Current WooCommerce Module Architecture

In order to propose a new architecture or solution for the system, the current WooCommerce module and its flaws should be described. This helps create a basis for comparison and provides the opportunity to fix these issues in the future. A new architecture would not be very useful if it incorporates the same issues as the current system. Therefore this section covers the flaws of the current WooCommerce module architecture and sets the ground for solutions and better systemic structure. The main disadvantages of the current system are as follows:

1. Code Readability
2. Duplicate Code
3. Hard Coding
4. Deploying Changes
5. Scaling
6. Difficult to Adopt New Technologies

All these disadvantages are explained in more detail in the following paragraphs in this section.

Code Readability
As with nearly every application, in order to improve and add new functionalities, new developments are required. That means adding blocks of code or improving the currently existing ones. While editing existing code and expanding the functionalities of already present components, it is necessary to pinpoint the part of the code which needs to be replaced or altered. Currently, while not a big problem, it is still tedious to find certain parts of the application because the code base has become unorganized. Although it gets easier when a person is more familiar with the structure. However, ensuring that the whole system works after refactoring certain functionalities is more complicated. In

many cases, functions from different areas of the system are connected and communicate information with each other. Therefore editing certain functionality is not as simple as making changes in one area of the code. This might mean that refactoring one certain part in the code requires a rework of another function as well. This has caused situations where certain functions have boolean inputs just to work properly while being called from multiple other parts, which in turn complicates the whole system even further.
Bigger problems occur when trying to implement new features and doing that by adding completely new functions or blocks of code. That means that the initial requirement is to analyze whether no other component already does something similar. And if that is confirmed then the new parts have to be situated into the fitting parts of the application in order to keep any kind of systematic organization. However, currently, that is very hard to do. A big part of the code is already quite old and in the past, new functionalities have just been added on top of the existing system and put wherever they would fit best. This has caused the ease of organization to drop and cause further confusion on where certain developments should be located. For example, when adding new methods for payments, how to differentiate between adding the functionality to the Payment class or Gateway class. The former has a function called check_payment() while the latter has a function called check_payment_status(). At first glance, it looks like they should both be in the Payment class however they are not. So if someone, who is not yet very familiar with the system, has to implement a new functionality regarding payments, it can turn out to be a very confusing choice in order to avoid disorganizing and complicating the existing application.
Lastly, due to the complexity of the code, changes and refactoring may cause other parts of the system to become unused and redundant. However, this is not always noticed and so there are parts of the system that are not actually used. Some functions are not used anywhere in the application and without removing them, the code base gets larger and even harder to navigate as new functions are added but deprecated ones are left in.

Duplicate Code

Duplicate code is an issue on a larger scale and does not strictly apply to the WooCommerce module itself. Although there are instances of duplicate code on the module level as well. Certain classes use the same kind of functionalities and they are implemented in both classes, rather than a single function that could be used by both. Such examples can be found in the Label and Shipping class. But on a larger scale, most of the duplicate code is between the different modules that Maksekeskus develops. This thesis scope has been set to cover the WooCommerce module but overall, every module essentially does the same thing. They provide payment and shipping possibilities to different platforms for e-commerce solutions. Therefore there are parts of the code that do the exact same thing but are implemented multiple times just because of the module amount. These parts could be gathered and made accessible for each solution. This would help

eliminate the need to make changes multiple times and will save resources in the long run.

Hard Coding
Certain parts of the application have to be hard-coded because some of the data can not be acquired through API or function calls. For example, providing the customer with a tracking link for their order's shipment. Either every shipping company does not have the possibility to create tracking links through API calls or this kind of feature can not be implemented due to excessive work or incompatibilities. Therefore currently, tracking links are hard-coded into the application. If a new shipment is created, the shipment identification number is added to the tracking link and returned back to the customer as a whole. However, when anything changes within these links, for example, the shipping company changes its URL, the feature stops working. And in order to get it working again, the code must be altered and refactored. Afterward, a whole new deployment is needed to incorporate these changes into the application and every customer has to update their module. This means that a lot of work is necessary for just small changes.

Deploying Changes
WordPress plugins are updated once developers upload a new version of it. Then there are possibilities to have the plugin either update automatically, which means that when a new version comes out then the system automatically downloads and installs it on the server. But there are also manual installations. This means that the owner of the system has to enter the plugins part of their website and manually update outdated plugins. Maksekeskus uses the latter feature. Automatic updates are dangerous for payment solutions as many customers have created their own custom solutions and new updates could break their systems. What is more, some customers have not updated any other plugins, and automatically updating the WooCommerce module could cause integration issues with, for example, the main WooCommerce plugin. That is why manual updates are the only option for Maksekeskus' provided WooCommerce module. Such flow for updating plugins is quite inconvenient for the customer because of the manual work that they have to do. Therefore every time a new update is provided for the module, every customer, who wants to update and get the newest bug fixes, features, and additions, has to upgrade their system themselves.
Furthermore, with every new release or update, the current monolith has to be deployed to its full extent. Even if changes are small and only in one class for example to fix a crucial mistake, the whole system needs to be deployed. That creates a lot of extra necessary testing requirements. When the whole application is deployed, it needs to be tested fully. That is because of the fact that within monolithic applications, fatal errors can cause the whole system to break down. Therefore the risk of failure is much higher in the current architecture and a lot of time is spent on getting new deployments ready and deployed. The latter is currently done semi-automatically and is another process

that could be automated and improved with a new architecture and systemic approach in order to save time and developer resources.

Scaling
Although not as big of an issue as the previous ones, scaling could become a more serious one in the future. With a monolithic application, scaling the system means scaling it as a whole. Currently, there is no possible way to scale one part of the system which might be under a heavier load than others. Therefore if one part of the application requires more resources, it needs to be given to the whole system and that could mean that these resources are wasted in other areas because they might not receive as many requests as the most used part. Similarly, scaling a shared database in a monolithic application is difficult. For example, when one part of the data needs to be scaled to a NoSQL solution instead of a relational database, this would not be possible with a shared database. Either a new database has to be created just for one specific use case or future business requirements and developments might be obstructed.

Difficult to Adopt New Technologies
As monolithic applications are usually developed with the same technologies throughout the system, it is challenging to integrate new, more efficient ones. That would require reworking a big part of the code or creating a new architecture by implementing microservices for example. Therefore it is harder for the developers to respond to the new business needs because the technological solutions are limited and innovating with new solutions is not possible without a lot of unnecessary rework.

2.6 Related Work

When comparing microservices and monolithic solutions, then most of the examples provide reasons, why the distributed architecture is more beneficial. Very few papers actually conclude that a monolithic approach was the more beneficial way to go. Such an example is provided when Isto first decided to migrate from a monolithic architecture to microservices but later concluded that it was the wrong choice and reverted these changes [MBMR21]. Mendonça et al. give a very informative insight into what could go wrong when trying to migrate to microservices and how the benefits do not outweigh the burdens [MBMR21].
A more common result is that microservices prove to be more beneficial and recommended. Therefore there are many papers that provide an example of how to migrate to a microservices architecture. Newman, S. provides patterns for migrating from a monolithic architecture to microservices [New19], Kalske et al. bring out the challenges with such solutions [KMM17], and Chen et al. came up with a decomposition algorithm, which helps develop microservice-based systems [CLL17]. These papers help provide an overview and the theoretical base that is needed in order to successfully analyze whether

microservices would be a good fit and how to approach the migration stage.
Cloud solutions that are implemented with microservices and keeping software on-premise is also already quite a thoroughly researched topic. Many papers cover the different advantages and disadvantages of one or the other solution. Pahl et al. created a comparison in which they analyze existing case studies of migrating to the cloud and provide different processes that are related to such migration and the problems that may occur [PXW13]. This sort of comparison allows others to plan ahead and prepare for possible issues while trying to move from on-premise to the cloud. Similarly, Boillat et al. cover the different business model components that are affected by moving to cloud computing [BL13]. Such research provides the examples and guidance needed, to successfully analyze the advantages of moving to the cloud. It is much simpler to work based on already existing examples and notes rather than trying to research and prepare for everything unknowingly.

3 Method

This section covers the methodology used in order to determine the best architectural solution for Maksekeskus' WooCommerce module. Subsection 3.1 covers the analysis method which is used for the proposed architectures and an evaluation for finding the most suitable architectural solution. Subsection 3.2 describes the processes and steps needed to take in order to modify an existing architectural solution and how to create microservices out of parts of a monolithic system.

3.1 Software Architecture Analysis Method (SAAM)

This section introduces the analysis methodology of the architectures with the Software Architecture Analysis Method (SAAM). Section 2.3 covered the essentials of three different methods for analyzing software architectures. This thesis uses Software Architecture Analysis Method due to it being the best fit for the scope of this thesis. SAAM is a compact analysis method that provides a scenario-based evaluation that is appropriate and does not become too complicated or out of the scope of this thesis in terms of the volume of work. ATAM benefits from a larger team for conducting the analysis which takes a longer time to conduct and that is not available for this thesis. ALMA is mostly focused on the modifiability of an architecture but this thesis aims to analyze the architectures in more ways and therefore SAAM was chosen as the optimal analysis method.
The principles of SAAM were briefly described in section 2.3.2. SAAM has six main steps that need to be completed in order to analyze software architecture. Figure 3 provides the visualization of the steps that are taken in this thesis in order to choose the optimal architecture for Maksekeskus' WooCommerce module.

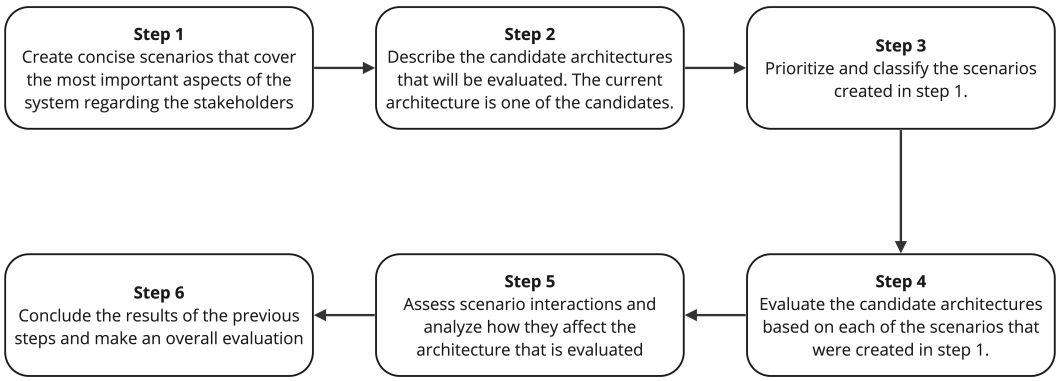

Figure 3. Steps in the Software Architecture Analysis Method

All the displayed steps in figure 3 and their descriptions are covered in the following subsections, starting with section 3.1.1 and ending with section 3.1.6.

3.1.1 Developing Scenarios

The first step of SAAM is creating scenarios from which to evaluate the candidate systems. Scenarios should cover the usage of the most critical parts of the application. Therefore the number of scenarios should be limited and they must be as concise as possible. In order to develop scenarios for specific stakeholders of the system, these stakeholders must be identified. There are four main classes that would have interest or concerns regarding the Maksekeskus' WooCommerce module. These are customers and merchants from the e-commerce side. The first ones buy products online using Maksekeskus' payment options. The merchants are the ones that set up stores and make a contract with Maksekeskus to share a small part of the revenue for the payment collection services.
From the company side, there are also two main stakeholders: developers and system administrators. Developers are the ones who write code and expand the application. They add new functionalities, fix bugs and improve the efficiency and performance of the overall system. Administrators make sure that the different environments meant for testing and demoing for new clients are working well. Since the current system is not cloud-based, there are no services to maintain regarding the module's functionality.

3.1.2 Description of Current and Candidate Architectures

The second step of the analysis covers the creation of design class diagrams for the current architecture and the candidate architectures. As the current documentation of

the WooCommerce module's architecture has gone missing then these diagrams could provide a good basis for analysis and description. The diagrams are created by using the concepts of Unified Modeling Language (UML). Evans et al. described UML as a quickly growing de-facto standard for different system modeling jobs [EFLR14]. UML provides universal notions and concepts for modeling object-oriented architectures. The notations that are used in this thesis are as follows:

- Multiplicity notation is not used due to all the connections being one-to-one.
- Public function or variable - "+"
- Private function or variable - "-"
- Protected function or variable - "#"
- Static function or variable - underlined
- Abstract function or variable - italics
- Final function - upper case throughout
- Arrow with a solid line - association connection between classes
- Arrow with a dashed line - dependency relationship between two classes

The covered architecture models are based on the background work of this thesis and a conclusion of a company side meeting. As this thesis is focused on the differences between monolithic architecture and microservices architecture, then these two will be the edge case candidate architectures. And a combination of the two architectures will be analyzed as a third possible candidate. Therefore this thesis covers three different architecture models and their respective design class diagrams. These architecture models contain the current Maksekeskus' monolithic WooCommerce module and two potential theoretical solutions that could replace the current system - a hybrid solution containing a part of the already existing monolith with newly added microservices and a mostly microservices-based system that includes a small monolithic part of the old system. The whole system can not really be converted into microservices and brought into the cloud due to the way that WordPress and its plugins work. A part of the plugin has to be downloaded and installed locally and therefore turning a plugin into only microservices would be very difficult and overly complicated.
In order to create all the design class diagrams and provide descriptions of the candidate architectures, the current monolithic architecture has to be documented. As it is one of the candidates, then the current code can be analyzed in order to map out the functionalities and create the diagram. After the current architecture is described and modeled on a diagram, the theoretical hybrid and microservices solution diagrams can be created out

of the existing ones. The monolithic architecture is analyzed in order to find the most critical sections that should be converted into microservices for the hybrid architecture. For the microservices architecture, the monolithic system is analyzed, and everything that can and should be a microservice, is converted into one. Of course, there are parts of the system that can not be turned into microservices or are very WooCommerce module specific that they should not be migrated as separate services. These will stay as a small monolithic part of the microservices architecture.

3.1.3 Classify and Prioritize Scenarios

Ionita et al. concluded that during the software analysis with SAAM, the scenarios should be classified as indirect and direct [IHO02]. Direct scenarios are ones that the architecture supports and does not need any changes to complete them. Indirect scenarios need certain changes in the architecture in order to support their completion. Since the new proposed architectures in this thesis are theoretical and require implementations and changes with every scenario, the classification of direct and indirect scenarios does not play a role. Therefore this analysis will include all the scenarios in the evaluation process. Otherwise, the indirect scenarios would be different with each architecture and the overall evaluation would have to be specifically modified to match the amounts of indirect scenarios and their effects on the assessment. This modification of the SAAM methodology was agreed upon with the supervisors and it does not change the outcome of the evaluation, which is one of the main goals of this thesis.
For prioritizing the scenarios, the stakeholders and their importance to the system must be considered. While not a definite deciding factor, higher priority scenarios could help determine the best architecture if the overall evaluation concluded as a tie. Ionita et al. determined that prioritization helps specify which scenarios are more important than others [IHO02]. In this thesis, the prioritization will be done on the stakeholder level and therefore each scenario for that particular stakeholder will have the same priority. Weights according to the stakeholder priority will be added to each scenario in the overall evaluation.

3.1.4 Scenario Evaluation

Scenario evaluation covers each scenario independently and evaluates how each possible architecture performs regarding multiple different non-functional requirements. The evaluations are displayed in the form of a table and each architecture gets a rating from 1 to 5 for each requirement and the architecture with the highest possible overall points is theoretically the best choice with the considered scenarios.
Non-functional requirements are defined and described in many multiple ways. M. Glinz has brought out 13 different variations in his paper [Gli07]. Many of the definitions that are included by M. Glinz in his paper are quite specific and could be hard to understand

[Gli07]. However, a few of them are more compact and M. Glinz has brought out one that states that non-functional requirements "describe the non-behavioral aspects of a system, capturing the properties and constraints under which a system must operate" [Gli07]. Therefore they characterize the system not in a functional way, but in a way that describes the system's attributes like security, availability, portability, and any other similar kinds of behavioral properties.
There exist many different non-functional requirements and analyzing each scenario against every possible requirement is not only unnecessary but also very time-consuming. Therefore this thesis focuses on the five main requirements, which are the most important for this project and overall most important for all the modules that Maksekeskus provides for different platforms. In no particular order, as they are all equally important, these non-functional requirements include the following:

1. Security
2. Performance
3. Scalability
4. Modifiability
5. Reliability

Every scenario will then be evaluated in all candidate architectures with each of the non-functional requirements. As the ratings used range from 1 (very bad) to 5 (very good), the total amount of points that any architecture can collect from one single scenario is 25 due to the five non-functional requirements. And the total amount of points will be 25 times the amount of scenarios created.

3.1.5 Scenario Interactions

Ionita et al. deduced that when two scenarios require changes to the same part of the architecture then they are interacting [IHO02]. Therefore as the fifth step in SAAM, scenario interactions can showcase the downsides of the candidate architectures and highlight the architectural changes that are needed. As explained in section 3.1.3, this thesis considers each scenario as indirect for consistent evaluation regarding every candidate architecture. Therefore scenario interactions will analyze each two scenarios and whether they can be performed simultaneously. If two scenarios somehow affect each other then the impacts will be described. For ease of reading, all scenario interactions will be conducted within a table that provides an informative overview of each of the interactions.

3.1.6 Overall Evaluation

Dobrica et al. concluded that the overall evaluation of SAAM should take into consideration the scenario evaluations and their prioritization [DN02]. Therefore the final outcome of the analysis will be the outcome of the scenario evaluations, which will then be weighed accordingly based on the scenario prioritization. The architecture that scores the highest points is theoretically the best-suited candidate. However, other factors have to be considered as well. For example, the development time and overall resources that have to be put into realising each of the candidate architectures. The overall evaluation should propose a clear decision that should be taken regarding the choice of a future architecture for Maksekeskus' WooCommerece module.

3.2 Extracting Microservices

This section covers the steps that should be taken in order to deploy a microservice and how one could be extracted from the current WooCommerce module's monolithic architecture. Two of the three candidate architectures include microservices and one of the goals of this thesis is to partially implement a new architecture if the software analysis proves that the hybrid or microservices architectures are better suited than the current monolithic architecture. Therefore the extraction and setting up of a microservice must be planned. Of course, if the evaluation of the analysis concludes that the current monolithic architecture is the best-suited option then this plan will not be put to use as no microservices are extracted.
The current Maksekeskus infrastructure is hosted in Amazon Web Services (AWS). Therefore to keep everything in the same place and allow for easier maintenance and monitoring, the created microservices will also be deployed in AWS. Currently, the monolithic applications for testing and demo purposes are deployed with Kubernetes using Docker containers. This methodology will be used for deploying the microservices as well.
Kubernetes is an open-source software that simplifies the management of containers [kub23]. It allows running different containers on one physical server, where each container uses the shared operating system, however, they all have their own share of the processor, memory, and filesystem for example [kub23]. In order to run these containers in Kubernetes, another open platform software called Docker is used. These containers are isolated applications that can be developed, shipped, and deployed separately on a single host computer and what is more, they simplify the standard continuous integration and continuous deployment pipelines [doc23]. The Docker container contains everything that is needed to run an application. Therefore it allows for the avoidance of problems like applications running on the developer's computer but not in production [doc23].
Using Docker and Kubernetes, the plan for extracting a microservice was created and it consists of five main steps.

1. Define the service and its goals. Describe the importance and necessity of it and what kind of problems it would solve.
2. Create and set up a GitHub repository where all the necessary code is maintained. GitHub is also used for running pipelines and deployments whenever new code is pushed into main/test.
   (a) Add description in the readme for more information.
   (b) Determine a domain.
   (c) Create branches test, main (main should be there already after creating the repository).
   (d) Separate branches should be created per ticket.
   (e) Code reviews must be done unless said/agreed upon differently.
3. Create a local development environment.
   (a) Find an official docker image that is maintained well, contains PHP, and a basic web server. Should be as clean/lightweight as possible.
   (b) Make it work with Fast Reverse Proxy and profile environment variables like other existing environments.
   (c) Should run on Dockerfile and GitHub repository. Might need backups depending on the service.
   (d) If possible, implement the Xdebug plugin for ease of debugging.
   (e) Use a simple, minimal but yet professional folder structure .
   (f) Use the latest version of PHP and Apache/nginx or any other software if possible.
4. Create an initial version of the service.
   (a) Place the code as required in the folder structure.
   (b) Create unit tests.
5. Deploy service to AWS.

If the evaluation concludes that either the hybrid or the microservices architecture is the best possible option for the WooCommerce module, this plan will be used in order to fulfill one of the goals of this thesis, which is to partially integrate the new architecture.

4 Results

This section covers the results of the thesis and conducts the analytical work described in the Methods section. Section 4.1 and its subsections cover the software architecture analysis of the three proposed candidate architectures. The analysis is performed using the SAAM methodology. Section 4.2 takes the result of the software analysis and follows the plan set in section 3.2 in order to partially integrate a new architecture.

4.1 Architecture Analysis with SAAM

This section covers the results of all the steps that were taken while analyzing possible architecture solutions for Maksekeskus' WooCommerce module. Each subsection presents the result of one step in the SAAM methodology. The methods used for achieving these results were described in section 3.2.

4.1.1 Created Scenarios

Scenarios are filtered by stakeholders which were established to be four groups: the merchants, customers, developers, and system administrators. The created scenarios are the following:

Customer Scenarios

- A customer must be able to select from multiple different enabled payment options.
- A customer must be able to select from multiple different enabled shipping options.
- A customer must be able to track their order with methods that allow it.

Merchant Scenarios

- A merchant must be able to offer payment options ranging from credit card payments to bank links and gift cards.
- A merchant must be able to control which shipping options are available for the customers.
- A merchant must be able to refund purchases that do not meet the customer's expectations.

Developer Scenarios

- A developer must be able to quickly make changes to static code

- A developer must be able to add new shipping and payment options quickly in order to meet the market demand.

Administrator Scenarios

- An administrator must be able to scale parts of the system differently, due to the variety in usage load.
- An administrator must be able to monitor the system and identify problems in real-time.

These scenarios will be used in the upcoming SAAM steps in order to evaluate all the candidate architectures, which are introduced in section 4.1.2.

4.1.2 Description of Current and Candidate Architectures

This section covers the basic descriptions of the current architecture of Maksekeskus' WooCommerce plugin and the possible future variants of this architecture. Each class has its main functionality specified but not on the level of individual functions. This would be unnecessary and out of scope for this thesis.

Candidate Architecture - the Current Monolithic Architecture
The first candidate architecture is the current monolithic architecture and its completed design class diagram can be found in figure 4. It includes the main classes of the application and their associations with each other. Although the design of such a class model does not require special placement of classes, the displayed and created diagram has a left-to-right hierarchical structure. Parent classes, classes that create or initiate other classes, or classes which have other classes dependent on them are placed more to the left. This is a monolithic application, where the whole system is deployed and shipped as a new version in its entirety.
The main class that is initiated first is Makecommerce. This class mostly creates other class objects and initiates them. Through it, other classes can create API calls to the current back-end which provides the application with data like enabled payment options, and the ability to create new transactions and register shipments for example.
Gateway is the parent class for the Maksekeskus' payment methods. It extends WooCommerce's own gateway in order to provide an alternative and therefore inherits the functionality and variables of the parent class as well. The Gateway class uses traits Refund and Subscription. Refund provides the possibility for merchants to create refunds for the customers' purchases after their money has already been transferred through Maksekeskus. The latter trait gives gateway support for subscription-based transactions.
The gateway has a child class WooCommerce. It consists of functions and variables that are overwritten compared to the original WooCommerce gateway. This includes

Figure 4. Current architecture of Maksekeskus' WooCommerce module

different settings required for the admin page in WordPress as well as functions that enqueue our Javascript files and initialize all the enabled payment methods for the customers. WooCommerce also uses three different traits, which are Banklink, Creditcard, and Paylater. These provide the gateway with payment method-specific functionalities. For example, the Creditcard trait allows for the system to initialize the iframe for card payments and the Paylater trait provides methods for calculating interest for the provided pay later payment options. The Banklink trait contains functionalities that are more universal for the other methods as well. This includes a method for updating all the possible payment methods in the WordPress database.
WooCommerce's child class Methods has specific functionality for making sure that the correct payment methods are displayed for the store's customer side and that it would be customizable. For example, grouping all the payment methods by certain countries

or displaying said countries in a chosen order. What is more, the Methods class creates notifications for the merchant on the WordPress admin side. For example, it lets the merchant know that they do not have any payment methods enabled even though they are using Maksekeskus' WooCommerce plugin. Payment class checks a customer's payment and makes sure that the it has been correctly processed. If there are any errors or the customer has canceled the purchase then this class handles such occasions. API class provides the application with different functions that are required for the merchant on Wordpress' admin side. However, Maksekeskus' software development kit (SDK), which is included in every one of the developed plugins, is initialized by the main Makecommerce class and provides the ability to communicate with the Java-based back-end. It is the class Maksekeskus. The SDK provides the system the ability to create all kinds of requests and queries to the Java-based back-end application. Therefore the naming scheme between Maksekeskus and API can cause confusion, as the former is created by Makecommerce with a function called get_api(...).

The second class that extends a parent WooCommerce class is the abstract class Method. It is a parent for all the shipping methods available like Omniva, DPD, Smartpost, and LP Express. The class includes all the common functionalities and base values like maximum shipping weight and phone number verification. The abstract classes Parcelmachine and Courier are child classes of Method. They provide more specific functionalities and values for either parcel machines or courier services. The specific child classes of those, such as Omniva and Smartpost for example (the classes under Courier and Parcelmachine differ, even though they share the same class names), provide the methods with even more specifications. For example the names of each method and custom functionalities like the function for obtaining LP Express parcel machine template size called get_lp_express_presets().

The completed design class diagram in figure 4 shows that the overall structure of the system is quite compact and well-managed. There are however a few components that are problematic. The first of which is the shipping methods part that is extended from the Method class. The Courier and ParcelMachine classes are parent classes for all the different shipping methods that the module currently offers. Every time a new method is added, a new class must be created, configured correctly to be the child class of either Courier or ParcelMachine, and then properly called and handled within the system itself. These additions take a bit of time and whenever this has to be done, it is usually done in all the other modules that Maksekeskus offers as well. Moving these new methods out of the monolith and into separate microservices would eliminate the waste of resources that is used on implementing the same feature multiple times on different platforms. Furthermore, the addition of new methods would be much smoother and would not require the merchants to update their respective modules. The cloud solutions would automatically be available after a new deployment.

Another weak point is the existence of hard-coded information like tracking link URLs

or pay-later information. The possibility of these static values changing is quite likely and should therefore be made into separate services in order to add flexibility and speed to updating the values and providing all the customers with the correct information. Currently, whenever the tracking URLs change, for example, older module versions will be broken and the tracking will constantly provide the wrong links to the customer. Having such services in the cloud enables the developers to quickly fix these values and instantly make them available for the end users.

Candidate Architecture - Hybrid Solution Containing a Monolith and Microservices

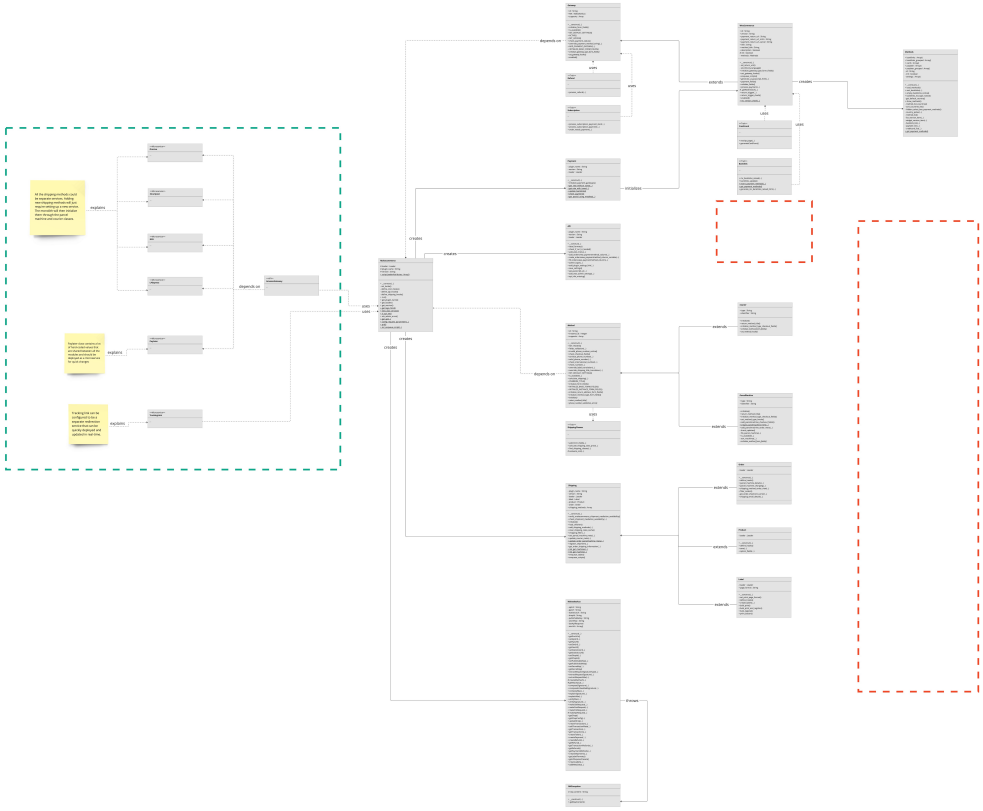

Figure 5. Hybrid architecture of Maksekeskus' WooCommerce module

The second candidate architecture is the hybrid variant which is intended to remove all the negative parts of the current system, that were described in the Candidate Architecture - the Current Monolithic Architecture part in this section. The design class diagram of such a solution can be seen in figure 5. The changes in comparison to the original architecture are marked with either green or red slashed boxes. Green boxes represent the new parts

of the system. In the added diagram, these new parts are the new microservices and the new API that provides the module with the possibility of using these services. The red boxes represent the parts that have been removed from the monolithic architecture.
The original application has parts that contain embedded configuration data inside the code, like shipment tracking link web addresses for example. Whenever a shipping company decides to change their tracking link address or modify the system in some other way that makes the previous URLs invalid, the WooCommerce module then requires a new version update just to have the shipping method working properly again. In order to make this more comfortable for the store owners, the tracking link should be migrated to a microservice. As a microservice in the cloud, real-time updates can be done quickly and the merchants do not have to do anything to take advantage of new tracking possibilities or just regular updates. However, this could be used by third-party users as well, to track shipments that were not created through the WooCommerce module's system as shipment tracking for different companies like Omniva or Itella is still possible for any order made through their system. This creates an opportunity to build the tracking link into a product in itself. Therefore the microservice can act as a redirection at first but can be built into a web page with the possibility of just inserting your tracking number and quickly finding your order status, whether or not the order was placed through Maksekeskus' modules. The tracking link microservice can therefore exist as a separate service that will be publicly available and not access-restricted via the API. That is also represented in the diagram in figure 5, as the module can directly use the service without having to make an API call.
Another part of the system that benefits from being turned into a microservice is the pay-later functionality. Similarly to the tracking link system, it contains hard-coded information that is prone to change. For example the descriptions of different pay later possibilities. Displaying relevant and up-to-date information for Maksekeskus' customers is important and should anything change in the way that the interest is calculated or if a payment service changes its description, these changes can be quickly developed and deployed for the customers. Furthermore, new pay-later options can be added fast. Currently, every change that is made requires a new plugin deployment but the cloud provides the ability to adapt and react to market changes quickly and without having to trouble the merchants themselves.
What is more, all the shipping methods have been moved into the cloud as separate microservices. They will be initialized through the new API and will either be a new courier or parcel machine method based on the parent classes that have been left in the monolith. Whenever a new method is implemented, it can be added as a new microservice and the module can initialize it as a new courier method for example. This would require minimal if not any changes in the monolithic side of the module and therefore provide a much faster integration and removal of duplicate code, as all the different platform modules could use these services instead and there would be no need to add a

new shipping method to every single platform independently. This also applies to all the other microservices as well. Whenever either pay later gets an update or new tracking functionalities are added, they are instantly available for all the merchants for example on the Magento or PrestaShop platforms, not only the ones using the WooCommerce module.
Of course, these proposed theoretical changes require practical work in order to be realized in the actual module itself. Since the amount of new microservices is quite limited, then the new architecture and the migration to the cloud could be completed in a reasonable amount of time. The initial versions of the tracking link and pay-later service should both be done in a month by one developer. Of course, testing will take time to make sure that everything works but effectively, working only on these services should provide the results in the span of a single quarter. However, due to the fact that the module itself needs constant work - new features, bug fixes, logic changes, and modifications, the resources can not fully be devoted to only developing the microservices. Therefore a more real-life estimation for the completion and deployment of the paylater and tracking link microservices would be around two to three months. Implementing the shipping methods as microservices will take more time and resources. This solution requires more changes in the monolith and the logic behind initializing them through the API must be carefully thought through. The time estimation for such a solution would be around three to four months with all the other tasks in parallel.
Lastly, integrating microservices makes the whole system more complex. As was described in section 2.2.2, implementing independent and isolated services makes the coordination and communication of separate parts of the whole application more complicated. It also adds more points of failure as one service could break and become non-functional while the other services are trying to use it. Therefore the overall design of the communication between the services and monitoring has to be thought through before the practical final implementation.

Candidate Architecture - Microservices Solution Containing a Small Monolith
The third candidate architecture contains as many microservices as possible. However, as the plugins used in WordPress are locally installed on the site's server, there does not exist any way to move the whole system into the cloud. A small piece of the plugin has to be downloaded as a .zip file by the customers or installed directly in WordPress. This small part of the plugin could then have all the functionality needed, to call every other part that is extracted as a microservice. However, compared to the hybrid architecture, the microservices architecture would require a lot more work in order to be completed. In addition to all the services that were migrated into the cloud in the hybrid solution, the microservices solution has many additional parts that need to be developed. All these changes can be seen in the figure 6. The red boxes display the changes in the monolith and show which parts have been removed and the green box shows all the additional

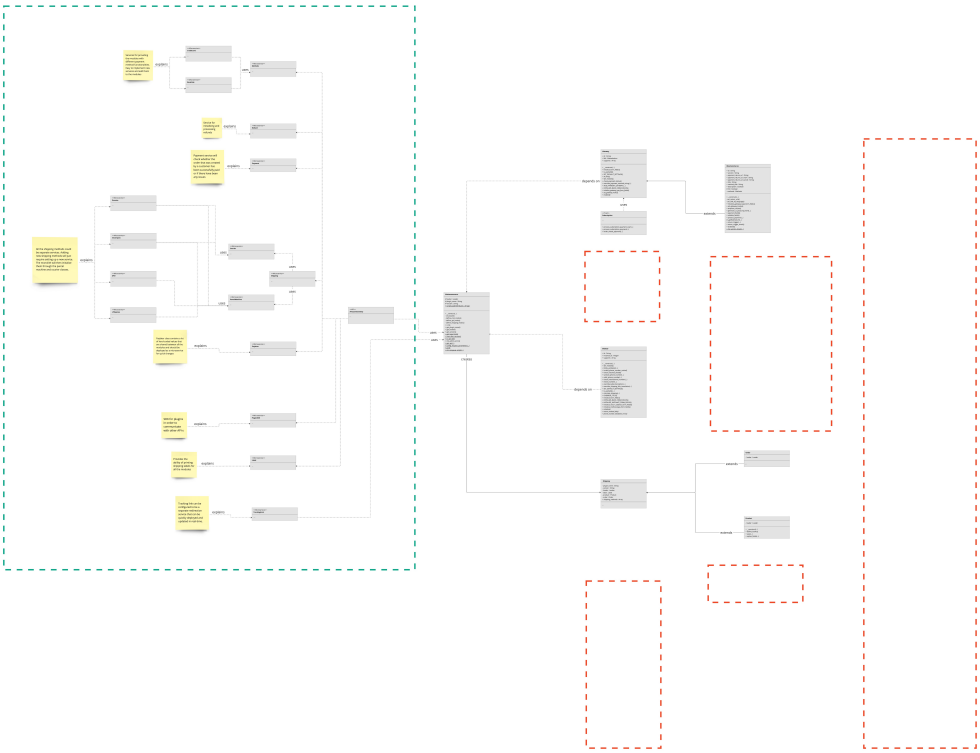

Figure 6. Microservices architecture of Maksekeskus' WooCommerce module

services that are theoretically proposed, similarly as for the hybrid architecture.
The monolithic part retains all the WordPress and WooCommerce-specific functionalities, as these provide very little to no help to other platform modules. Therefore migrating these components into microservices would only create more work and basically waste resources. These are the WooCommerce child classes Method and Gateway. The Subscription trait used by Gateway is also a WooCommerce-specific functionality and does not need to be turned into microservice. The WooCommerce class is a child class of Gateway and provides our module with the necessary functionalities in order to make the payments work for the merchants and their customers. This class has no use outside of our WooCommerce module and should therefore not be turned into a microservice. Similarly, the Shipping class and its child classes Order and Product are very WooCommerce specific, although there are certain parts of the Shipping class that can be utilized elsewhere and therefore this class is split and partially migrated into the cloud and turned into the Shipping microservice. Lastly, the Makecommerce class will still be included in the monolith in order to communicate with the new API and initialize the classes that are left in the monolithic part.

One of the biggest changes is having the whole payment processing part as separate microservices. The structure would be similar to the current one in the monolithic architecture, but each payment method type would be a separate service and provide the WooCommerce module with all the required functionalities. Additional microservices also include more specific shipping solutions. Most of the shipping is moved into the cloud and distributed as microservices and the monolithic application can then call them through the API and Shipping service. Each method can be regularly maintained and initialized separately. And whenever new methods are added, these can be introduced as new services and the store owners can get access to them instantly, without having to update their modules or specifically, the monolithic part that is installed on their server. Lastly, the creation of labels is also moved into the cloud and prepared as a microservice. Since every module offers the ability to create labels for the orders and shipping them with the respective company, it could be unified as a microservice to suit all modules. This will also help integrate label creation into newly developed modules.
By taking into account the advantages and disadvantages of microservices, this architectural solution will suffer the most from the disadvantages and gain the most from the advantages. The biggest issue is the development and deployment time for getting this kind of architecture running. Taking into account the pace of the current development speed and available time to work on such architectural changes, the magnitude of these changes could take around two years to finally complete the migration. Furthermore, the complexity of the communication between all the services will require very specific and thorough mapping and the hosting cost for all the infrastructure will be significantly more than for the hybrid architecture.
On the other hand, the microservices architecture would be distributed and have the highest modularity out of all the three analyzed architectures. Each individual service could be developed with the best possible technology that does not have to match the other services. What is more, each of the services will be smaller in size and focused only on one functionality, which means the modifiability is high and they are easy to maintain and transform. Lastly, the independent services can be deployed individually and also scaled as such as well.

4.1.3 Scenario Classification and Prioritization

As there are different architectures then the scenario classification is different for each of them. In order to avoid issues with differing classifications, the choice of classifying every scenario as indirect was made in section 3.1.3. This creates an even basis for the evaluation of each architecture type. Therefore each architecture is analyzed with every scenario.
Maksekeskus is focused on offering its customers the highest possible quality service. Therefore the merchant scenarios are the most important and get the highest priority. Similarly, it is very important to offer the best, most comfortable, and secure possible

solutions and experiences to the merchant's customers as well. Because of that, the customer scenarios get the highest priority after the merchant scenarios. Of course, it is important to make sure that the development and deployment of the module are not to be missed but as SAAM requires prioritization, the developer and administrator scenarios come after customer scenarios in priority and share the same priority between the two.

4.1.4 Scenario Evaluations

The first scenario evaluation can be found in table 1. It covers the evaluation of the customer scenario "A customer must be able to select from multiple different enabled payment options" and evaluates all three architectures based on the non-functional requirements that were introduced at the start of this section. Each following scenario evaluation is based on the same requirements.

Table 1. The First Scenario Evaluation

Architecture	Security	Performance	Scalability	Modifiability	Reliability	Overall
Monolithic	5 (very good)	5 (very fast)	2 (bad, no separate scaling)	3 (mediocre, low modularity)	4 (good, no separate failing parts)	19
Hybrid	5 (very good)	5 (very fast)	3 (mediocre, payment options still in the monolithic part)	3 (mediocre, low modularity in the monolithic part)	4 (good, few points of failure)	20
Microservices	5 (very good)	4 (fast, slower communication between services)	5 (very good, separate services can be scaled individually)	5 (very good, each service can be modified and deployed separately)	3 (mediocre, many services that can fail)	22

The second scenario evaluation can be found in table 2. It covers the evaluation of the customer scenario "A customer must be able to select from multiple different enabled shipping options".

Table 2. The Second Scenario Evaluation

Architecture	Security	Performance	Scalability	Modifiability	Reliability	Overall
Monolithic	5 (very good)	5 (very fast)	2 (bad, no separate scaling)	3 (mediocre, low modularity and changes may affect other parts)	5 (very good)	20
Hybrid	5 (very good)	4 (fast, slower api calls)	5 (very good, separate scaling)	5 (very good, high modularity)	4 (good, a few parts of failure)	23
Microservices	5 (very good)	3 (mediocre, a lot of communication between services)	5 (very good, separate scaling)	5 (very good, high modularity)	3 (mediocre, more parts of failure)	21

The third scenario evaluation can be found in table 3. It covers the evaluation of the customer scenario "A customer must be able to track their order with methods that allow it".

Table 3. The Third Scenario Evaluation

Architecture	Security	Performance	Scalability	Modifiability	Reliability	Overall
Monolithic	4 (good, static URL's and slow deployments may pose problems)	5 (very good, fast computing)	2 (bad, no separate scaling)	3 (mediocre, low modularity and may affect other parts)	5 (very good, no multiple parts of failure)	19
Hybrid	5 (very good, separate service with quick deploys)	4 (good, slower redirection)	5 (very good, separate service)	5 (very good, separate service, no other affected parts)	4 (good, separate service is a point of failure)	23
Microservices	5 (very good, separate service with quick deploys)	4 (good, slower redirection)	5 (very good, separate service)	5 (very good, separate service, no other affected parts)	4 (good, separate service is a point of failure)	23

The fourth scenario evaluation can be found in table 4. It covers the evaluation of the merchant scenario "A merchant must be able to offer payment options ranging from credit card payments to bank links and gift cards".

Table 4. The Fourth Scenario Evaluation

Architecture	Security	Performance	Scalability	Modifiability	Reliability	Overall
Monolithic	4 (good, static code may cause inaccurate information)	5 (very fast)	2 (bad, no separate scaling)	3 (mediocre, new developments affect existing ones and are complicated)	4 (good, few fault points, faults are local in the server)	18
Hybrid	5 (very good)	4 (fast, service communication time is longer)	3 (mediocre, no separate scaling but smaller monolith to scale in the server)	4 (good, low modularity so other parts are affected, less than with only a monolith)	4 (good, few fault points, faults are local in the server)	20
Microservices	5 (very good)	4 (fast, service communication time is longer)	5 (very good, can separately scale the payment services)	5 (very good, modify only the required services)	3 (mediocre, if faulty then all merchants have faults due to microservices)	22

The fifth scenario evaluation can be found in table 5. It covers the evaluation of the merchant scenario "A merchant must be able to control which shipping options are available for the customers".

Table 5. The Fifth Scenario Evaluation

Architecture	Security	Performance	Scalability	Modifiability	Reliability	Overall
Monolithic	5 (very good)	5 (very fast)	2 (bad, no separate scaling)	2 (bad, very low modularity, a lot of affected parts)	5 (very good, simple communication)	19
Hybrid	5 (very good)	4 (fast, slower api calls)	5 (very good, separate scaling)	5 (very good, high modularity and separate services)	4 (good, a few parts of failure and more complex)	23
Microservices	5 (very good)	3 (mediocre, a lot of communication between services)	5 (very good, separate scaling)	5 (very good, high modularity and separate services)	3 (mediocre, more parts of failure and much more complex structure)	21

The sixth scenario evaluation can be found in table 6. It covers the evaluation of the merchant scenario "A merchant must be able to refund purchases that do not meet the customer's expectations".

Table 6. The Sixth Scenario Evaluation

Architecture	Security	Performance	Scalability	Modifiability	Reliability	Overall
Monolithic	5 (very good)	5 (very fast)	2 (bad, no separate scaling)	3 (mediocre, other affected parts and low modularity)	5 (very good)	20
Hybrid	5 (very good)	5 (very fast)	3 (mediocre, smaller monolith but still no separate scaling)	4 (good, less affected parts)	5 (very good)	22
Microservices	4 (good, more complex and harder to achieve security)	4 (fast, more communication between services)	5 (very good, separate service to scale)	5 (very good, no other affected parts)	4 (good, more fault points and more complex communication)	22

The seventh scenario evaluation can be found in table 7. It covers the evaluation of the developer scenario "A developer must be able to quickly make changes to static code".

Table 7. The Seventh Scenario Evaluation

Architecture	Security	Performance	Scalability	Modifiability	Reliability	Overall
Monolithic	5 (very good)	5 (very good)	2 (bad, no separate scaling)	4 (good, static parts are simple to modify)	5 (very good)	21
Hybrid	5 (very good)	4 (good)	4 (good, static parts are mostly as separate scalable services)	5 (very good, high modularity)	5 (very good)	23
Microservices	4 (good, complexity adds security risks)	3 (mediocre, much more communication between services)	5 (very good, separate services)	5 (very good, high modularity)	4 (good, complex and more points of failure)	21

The eighth scenario evaluation can be found in table 8. It covers the evaluation of the developer scenario "A developer must be able to add new shipping and payment options quickly in order to meet the market demand".

Table 8. The Eighth Scenario Evaluation

Architecture	Security	Performance	Scalability	Modifiability	Reliability	Overall
Monolithic	5 (very good)	5 (very good)	2 (bad, no separate scaling)	3 (mediocre, complex developments)	5 (very good, few points of failure, simple architecture)	20
Hybrid	4 (good, new services may cause risks)	4 (good, service communication takes time)	5 (very good)	5 (very good, separate service)	4 (good, higer complexity and more points of failure)	22
Microservices	4 (good, new services may cause risks)	4 (good, service communication takes time)	5 (very good)	5 (very good, separate service)	3 (mediocre, very high complexity and even more points of failure)	21

The ninth scenario evaluation can be found in table 9. It covers the evaluation of the administrator scenario "An administrator must be able to scale parts of the system differently, due to the variety in usage load".

Table 9. The Ninth Scenario Evaluation

Architecture	Security	Performance	Scalability	Modifiability	Reliability	Overall
Monolithic	5 (very good)	5 (very good)	1 (very bad, not possible for the administrator to scale)	3 (mediocre)	5 (very good)	18
Hybrid	4 (good, more complex and scaling may produce issues)	5 (very good)	3 (mediocre, some parts are scalable)	4 (good, few separate services)	4 (good, few points of failure)	20
Microservices	4 (good, more complex and scaling may produce issues)	4 (good, slower communication s)	5 (very good, mostly scalable)	5 (very good, separate services)	3 (mediocre, complex and most points of failure)	21

The tenth scenario evaluation can be found in table 10. It covers the evaluation of the administrator scenario "An administrator must be able to monitor the system and identify problems in real-time".

Table 10. The Tenth Scenario Evaluation

Architecture	Security	Performance	Scalability	Modifiability	Reliability	Overall
Monolithic	5 (very good)	5 (very fast)	2 (bad, hard to scale monitoring possibilities)	3 (mediocre, modifications to boost monitoring are complex)	4 (good, reliable monitoring)	19
Hybrid	4 (good, more complex parts to monitor)	5 (very fast)	4 (good, difficult to add additional monitoring to some parts)	4 (good, separate services are easily modified)	5 (very good, more possibilities for monitoring)	22
Microservices	4 (good, more complex parts to monitor)	4 (fast, monitoring many services takes more time)	5 (very good, almost all parts can be monitored separately)	5 (very good, many separate services are easily modified)	5 (very good, many possibilities for reliable monitoring)	23

All the evaluation results and their overall values are added up and the conclusions are made in section 4.1.6.

4.1.5 Scenario Interactions

The scenario interactions section covers the different clashes between the scenarios and analyses, and whether there should be necessary changes made to the architecture. These interactions may have different impacts and have to be taken into account. For example, one scenario might be dependent on another scenario, or two of them can not be completed at the same time. All the interactions are covered in the table 11.

Table 11. Scenario Interactions

First Scenario	Second Scenario	Type of Interaction	Potential Impact
A customer must be able to select from multiple different enabled payment options.	A merchant must be able to offer payment options ranging from credit card payments to bank links and gift cards.	Dependancy	Whenever the merchant changes the available payment methods, the changes are made visible in the store checkout upon reloading the page. However, changing them will not forbid the customer from paying with a method that they had chosen before the changes were made. Interaction is therefore safe.

A customer must be able to select from multiple different enabled shipping options.	A merchant must be able to control which shipping options are available for the customers.	Dependancy	Whenever the merchant changes the available shipping methods, the changes are made visible in the store upon reloading the page. However, if the checkout is already loaded for the customer then all the presented shipping methods are still allowed. Interaction is therefore safe.
A customer must be able to track their order with methods that allow it.	A merchant must be able to control which shipping options are available for the customers.	Dependancy	The customer can only track their orders if the merchant enables the shipping methods that allow tracking.
A developer must be able to quickly make changes to static code.	A customer must be able to track their order with methods that allow it.	Complementarity	Static code changes can make the tracking solutions more accurate and fix URLs that are obsolete.

A developer must be able to add new shipping and payment options quickly in order to meet the market demand.	A merchant must be able to offer payment options ranging from credit card payments to bank links and gift cards.	Complementarity	New payment methods enhance the available options for the merchants
A developer must be able to add new shipping and payment options quickly in order to meet the market demand.	A merchant must be able to control which shipping options are available for the customers.	Complementarity	New shipping methods enhance the available options for the merchants
An administrator must be able to scale parts of the system differently, due to the variety in usage load.	An administrator must be able to monitor the system and identify problems in real time.	Conflict	When scaling parts of the system, the real-time monitoring may be affected and not work as supposed to before reconfigured. Can not be avoided.

None of the interactions brought out any major problems that would require architectural rework and therefore all the candidate architectures are valid options without any changes. A conclusion of all the scenario interactions is covered in section 4.1.6.

4.1.6 Overall Evaluation

None of the interactions covered in section 4.1.5 require any changes to any of the three architectures. The interactions that were covered are mostly unavoidable and do not cause problems. They do not bring out any issues that would require any components in the architecture to be changed. Therefore all the architectures could theoretically be implemented as they have been described in section 4.1.2 of this analysis and how they are displayed in the design class diagrams found in the same section.
The scenario evaluation part of the analysis brought out two architectures that would be the best fit to complete the set of scenarios. These architectures are the theoretically

proposed hybrid system and the microservices architecture. More specifically, the hybrid architecture achieved the best results and got a total score of 218 out of 250 in the scenario evaluation. The microservices architecture got 217 out of 250, so essentially the two solutions are equally as good. The current monolithic architecture managed to obtain 193 points out of 250. The hybrid and microservices architectures got the same scores in the most important evaluations, which were the merchant and customer scenarios and therefore weighing the scores is not needed.
The SAAM analysis concluded that both the hybrid and the microservices architecture could be potential practical solutions for the WooCommerce module. However, as the disadvantages and description of the microservices architecture made apparent, the complexity and resources needed to implement such a solution are not feasible. Therefore only the hybrid architecture and the current monolithic architecture are left as potential system architectures.
From the SAAM scenario evaluations, the monolithic architecture got a lot fewer points than the hybrid architecture. Due to that, realistically the hybrid architecture would be the best choice. The resources that are needed to implement such a solution are within reason and the time it takes to finish the project is also acceptable. Having a few distributed microservices will cause more complex communication between all the isolated services and the rise of infrastructure costs but the advantages of quick deploys, managing all the static code in the cloud, making the overall code of the application more readable, and having independent scalable components outweigh the disadvantages and also fix some of the original problems that were presented in section 2.5.2. Therefore the chosen architecture to move forward with will be the hybrid architecture, which can be seen in figure 5.

4.2 Partial Migration To Microservices

One goal of this thesis was that in case of a new architecture choice, a microservice will be extracted from the original monolithic system architecture. Therefore the hybrid solution is taken into use and will be expanded with more microservices in the future. However, these are out of the scope of this thesis and will not be implemented or described further. The service that is migrated to the cloud as a microservice is the tracking link service. Further and a more in-depth description is in section 4.2.1. This section covers the different steps taken, modifications made and advantages achieved with the new microservice.

4.2.1 Microservice Description

The plan for creating a microservice was described in section 3.2. This plan was executed and the steps were followed closely. The end result is a tracking link microservice that allows all the different Maksekeskus' modules, like the ones for WooCommerce,

Magento, and PrestaShop platforms to create shipment tracking links that are redirected through the new service. This eliminates the unnecessary amount of duplicate and hard-coded values in each of the modules' codes. Furthermore, this provides the ability for developers to quickly make changes and instantly deploy them for the customers. As the code for calling the microservice in the modules will be static, new microservice versions do not require any changes to the original modules, and therefore no new release is required in order to implement the changes. The overall plan was executed as follows:

1. The service is defined as follows - The tracking link microservice will provide a fast and reliable way for e-commerce merchants to create tracking links for their customers. All the traffic will be directed to our service, which will then redirect every request based on the carrier, the country and language, and lastly the unique shipment identifier itself. The goal is to reduce duplicate code in our modules and provide quick updates for the merchants without having to release new versions of the modules. In order to match the rest of the projects, the service will be written in PHP.

2. New private GitHub repository was made for the service in order to use automatic pipelines for new deployments.

    (a) The readme file was configured to contain all the necessary information - a brief overview of the service, a description of how to use the local development environment, and all the changes that have been made to new versions of the service.

    (b) The domain that will be used for this service is tracking.makecommerce.net

    (c) Main and test branches were created. The main branch is for production and the code will be merged into it after thorough testing. The test branch is for testing the new updates and code changes through manual and automatic tests.

    (d) Rules are enforced and each new development or code change gets a new branch where all the changes will be implemented. After local testing and a code review, these modifications will then be merged into the test branch.

    (e) Code reviews are done by senior developers.

3. A local development environment was created for developers to be able to run their own tracking service in a local container and see changes they have made in real-time.

    (a) The official PHP Docker image with an apache2 server addition was chosen for the tracking service.

(b) Build files like the docker-compose.yaml and environment files like the .env file containing the domain names were created in order to run local containers with Fast Reverse Proxy.

(c) Dockerfile was created at the root of the project to allow quick deployments locally as well as automatic deployments in AWS through GitHub pipelines.

(d) Xdebug has not been implemented.

(e) The folder structure that has been implemented is as follows:

i. Folder "src" contains all the code, whether it is PHP, Javascript, or CSS and all the other files that are required in the server like .htaccess.

ii. Folder "tests" contains all the PHPUnit tests for the service.

iii. Folder "build" contains the scripts for building the containers, docker-compose.yaml for example.

iv. Folder ".config" contains all the environment variables.

v. Folder "doc" contains all the documentation for the service.

vi. Folder "res" contains all the static resources for the service.

(f) PHP 8.2 was used as it was the latest at the time of implementing the service.

4. An initial working version of the service was created in order to process the requests.

(a) Code that has been created can be seen in the "src" folder.

(b) Unit tests that have been created are in the "tests" folder.

5. Service was deployed to AWS.

The code for the tracking link microservice itself is private and will not be displayed in this thesis. However, it will be available for the defense committee and anyone else who is required to see it. This availability will be removed after the defense of the thesis or earlier if deemed possible. The written service can be found at:
`https://tinyurl.com/2xp7zrux`
Due to confidentiality reasons, the access to the file is password protected. The password can be requested from the author.
For an overview, a brief description of the developed service will be provided. As this tracking service is not private and does not use an API gateway to communicate with the rest of the system, the overall configuration is a bit different from a regular or standard microservice. It also uses a file system as a database due to the need to access data fast, and the very small amount of it. Therefore all the necessary shipment providers tracking links are stored in a *.json* file. With the JSON format, it is very simple to add new values, remove old values or modify existing ones. As this was one of the goals for this service,

the ease of data modification is important, and this way it is achieved. Using a relational database would use a lot more resources that are not needed in this case and would just make the service slower.
The overall functionality is managed by the index PHP file, which will act as a router. All the service's traffic and requests will be handled by it, regardless of what the specific paths are. It then checks whether the correct data was passed along and how many parameters were added to the path. For example, when all the data is added correctly, there are four parameters: the carrier, the language, the country, and the shipment identifier itself. The most important ones are the carrier and the identifier, as without them, the service can not redirect the customer to the correct shipping provider's page. However, when the language, country, or both are missing, the JSON file of all the links is constructed so that the first values are the default values. So for example, if a shipping provider ships to Estonia and Latvia, it will have either one of those countries as the first option in the array. And each of these countries could include multiple values as well. For example, the Estonian country array can contain the Estonian and English languages. So if the Estonian is first, it will be counted as default and whenever the language is missing from the parameters, the service will automatically default to the first value.
In order to handle bad requests, the current version of the microservice will redirect the customers to another PHP file, which will provide them with the information that their request can not be handled as a suitable shipping carrier and the shipment itself was not found. They are prompted to check the URL that they have tried to reach and whether it has all the necessary information that is required.
Other parts of the service include environment files, and different Docker-specific files for building the service, configuring the service, and running it on a developer's computer as a local service. These files are essential for the microservice to function properly on the Kubernetes engine and get properly deployed as new changes are made. However, as these configuration, environment and build files do not change how the service itself handles requests then these files and the code inside them will not be analyzed further in depth in this thesis.

4.2.2 Changes in the WooCommerce Module Workflow

The new tracking link service removes the need for creating the tracking link inside the module itself. In this section, the functionality for creating a link for the customer is described. However, as that function was migrated into a microservice, the module has to be reconfigured to take advantage of such a service in a way that is static and does not require frequent changes.
In order to bring out the changes made to the creation of the tracking link, figure 7 and figure 8 display the differences that were made. The graphs consist of steps that are taken in order to successfully create a tracking link and display the shipment page to the user. These graphs are simplified in order to show the differences more understandably.

They do not contain all the small functionalities that the module uses. The black nodes signify starting or ending points for the flow. The yellow nodes display actions that have multiple outcomes and these are compared in the orange nodes. Pink nodes denote the regular steps that the module takes in order to create the link.
The tracking link creation and usage flow for the user originally consists of a few steps and can be seen in figure 7. At first, the customer has to make a purchase with a shipping method that has order tracking enabled. After the client successfully pays and the shipment is registered, the module creates a tracking link by combining the order data with predefined carrier URLs. This link is then sent to the customer. Clicking on the link will then provide the user with the shipment tracking information on the official carrier page.
The new flow will contain a few extra steps between the user making a purchase and clicking the tracking link which can be seen from figure 8. However, the user experience stays mostly the same. The extra steps are due to the module directing the customer to the microservice, which in turn acts as a redirection and directs the customer to the correct carrier's shipment tracking page. Initially, the tracking link gathers the order data the same way as before, but it does no error handling but rather just sends all the data it managed to gather to the microservice. For example, the link might be missing a language and a country. Then the microservice has to determine the default carrier URLs and direct the customer to them. Between the initial tracking link and the redirection, the customer has to wait a bit longer than in the original flow. However, this does not cause any major inconveniences and helps Maksekeskus provide a more personal approach to the user by displaying accurate information about what is currently being processed. And whenever a problem occurs, it is possible to display an accurate error message to the customer, instead of just having a broken link to the carrier's shipment tracking page.

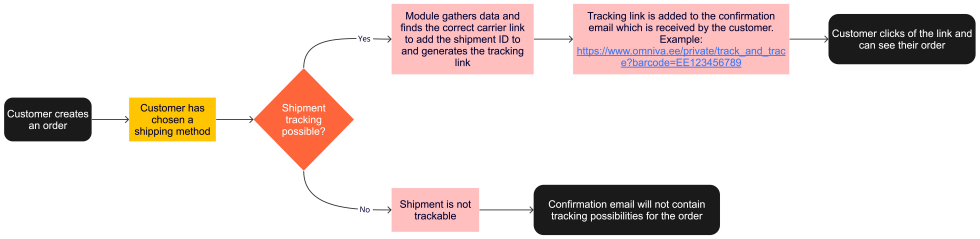

Figure 7. Original creation flow for the tracking link

Due to migrating the tracking link functionality mostly to the cloud, changes in the original functionality are required. The service is extracted from the current monolithic architecture and set up in the cloud. This can also be seen from the modified architecture

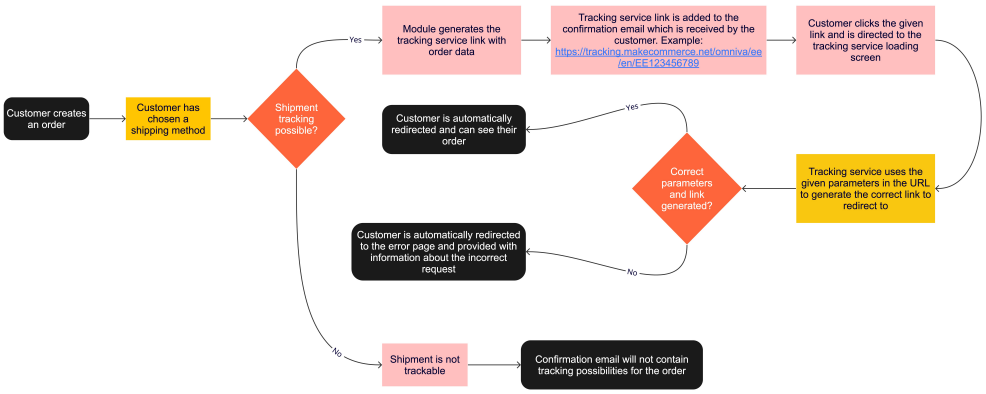

Figure 8. New creation flow for the tracking link

class diagram in figure 5, where one of the microservices is called "TrackingLink". The original extracted part of the code from the monolithic architecture can be found after this paragraph. It has been taken from Maksekeskus' WooCommerce module and demonstrates the issues with the hard-coded values. These arrays for tracking links are included in all the modules, not just the WooCommerce one. This creates a big array of duplicate values that requires fast changes whenever one of these values becomes deprecated. And due to these values being duplicates, changes need to be made in each module, and then a new version has to be delivered to the customers after which they can download these newest releases and upload them into their e-commerce stores. Overall a small change can create a lot of work for all the parties involved. In order to make these kinds of changes more comfortable for the merchants, this part is turned into a microservice. Whenever any link in the tracking system becomes deprecated, it can be quickly updated in the new service and the changes are instantly available for the customers after a new deployment. The original and old part of the code is the following:

```
/**
 * Returns the correct link to be used for tracking link
 *
 * @since 3.0.0
 */
public function get_tracking_link( $carrier, $order_id, $shopLocation
    = false ) {

  //do we use shop location or delivery location
  if ( $shopLocation ) {
    //get_base_country returns either EE, LT or LV. Everything else
       is irrelevant
    $tld = substr( strtolower( WC()->countries->get_base_country() ),
```

```
        0, 2);

    $userLang = $tld;
} else {

    //get order delivery location. Returns EE, LT or LV. Evertyhing
        else is irrelevant
    $tld = substr( strtolower( get_post_meta( $order_id, '
        _shipping_country', true ) ), 0, 2 );
    $userLang = substr( strtolower( get_post_meta( $order_id, '
        wpml_language', true ) ), 0, 2 ); //returns nothing if it
        doesnt exist, otherwise returns en, et, lt, lv, ru
}

//
//  create array for different options [carrier][tld][lang]
//  first one in the list must be default, this goes for the tld as
    well as for the userLang.
//

/* Omniva */
//EST
$trackingURLS["omniva"]["ee"]["ee"] = 'https://www.omniva.ee/abi/
    jalgimine?barcode=';
$trackingURLS["omniva"]["ee"]["en"] = 'https://www.omniva.ee/
    private/track_and_trace?barcode=';
$trackingURLS["omniva"]["ee"]["ru"] = 'https://www.omniva.ee/
    chastnyj/otslezhivanie_posylki?barcode=';

//LT
$trackingURLS["omniva"]["lt"]["lt"] = 'https://www.omniva.lt/verslo
    /siuntos_sekimas?barcode=';
$trackingURLS["omniva"]["lt"]["en"] = 'https://www.omniva.lt/
    business/track_and_trace?barcode=';

//LV
$trackingURLS["omniva"]["lv"]["lv"] = 'https://www.omniva.lv/
    privats/sutijuma_atrasanas_vieta?barcode=';
$trackingURLS["omniva"]["lv"]["en"] = 'https://www.omniva.lv/
    private/track_and_trace?barcode=';
$trackingURLS["omniva"]["lv"]["ru"] = 'https://www.omniva.lv/
    chastnyj/mestonahozhdenie_posylki?barcode=';

/* DPD */
//EE
$trackingURLS["dpd"]["ee"]["ee"] = 'https://tracking.dpd.de/status/
    et_EE/parcel/';
$trackingURLS["dpd"]["ee"]["en"] = 'https://tracking.dpd.de/status/
    en_EE/parcel/';
```

```
//LT
$trackingURLS["dpd"]["lt"]["lt"] = 'https://tracking.dpd.de/status/
    lt_LT/parcel/';
$trackingURLS["dpd"]["lt"]["en"] = 'https://tracking.dpd.de/status/
    en_LT/parcel/';

//LV
$trackingURLS["dpd"]["lv"]["lv"] = 'https://tracking.dpd.de/status/
    lv_LV/parcel/';
$trackingURLS["dpd"]["lv"]["en"] = 'https://tracking.dpd.de/status/
    en_LV/parcel/';

/* Smartpost */
//EE
$trackingURLS["smartpost"]["ee"]["ee"] = 'https://itella.ee/
    eraklient/saadetise-jalgimine/?trackingCode=';
$trackingURLS["smartpost"]["ee"]["en"] = 'https://itella.ee/en/
    private-customer/parcel-tracking/?trackingCode=';
$trackingURLS["smartpost"]["ee"]["ru"] = 'https://itella.ee/ru/
    chastnyj-klijent/otslezhivanije-otpravlenija/?trackingCode=';

//FI
$trackingURLS["smartpost"]["fi"]["en"] = 'https://www.posti.fi/en/
    tracking#/lahetys/';
$trackingURLS["smartpost"]["fi"]["fi"] = 'https://www.posti.fi/fi/
    seuranta#/lahetys/';
$trackingURLS["smartpost"]["fi"]["sv"] = 'https://www.posti.fi/sv/
    uppfoljning#/lahetys/';

//LT
$trackingURLS["smartpost"]["lt"]["lt"] = 'https://itella.lt/private
    -customer/krovinio-sekimas/?trackingCode=';
$trackingURLS["smartpost"]["lt"]["en"] = 'https://itella.lt/en/
    private-customer/track-shipment/?trackingCode=';

//LV
$trackingURLS["smartpost"]["lv"]["lv"] = 'https://itella.lv/private
    -customer/sutijuma-meklesana/?trackingCode=';
$trackingURLS["smartpost"]["lv"]["en"] = 'https://itella.lv/en/
    private-customer/parcel-tracking/?trackingCode=';

//carrier exists
if ( isset( $trackingURLS[$carrier] ) ) {
  //country exists
  if ( isset( $trackingURLS[$carrier][$tld] ) ) {
    if ( isset( $trackingURLS[$carrier][$tld][$userLang] ) ) {
      return $trackingURLS[$carrier][$tld][$userLang];
    } else {
```

```
            //selected language not found, return carrier tld default
                language.
            return current( $trackingURLS[$carrier][$tld] );
        }
    } else {
        // selected tld not found, return tld based on store country
        $tld = substr( strtolower( WC()->countries->get_base_country()
            ), 0, 2);
        if ( isset( $trackingURLS[$carrier][$tld] ) ) {
            $defaultCarrierTLD = current( $trackingURLS[$carrier][$tld] )
                ;
        } else {
            // no store country was found either
            $defaultCarrierTLD = current( $trackingURLS[$carrier] );
        }

        return $defaultCarrierTLD;
    }
  }

  //not even carrier was found. Return nothing
  return "";
}
```

Firstly, in order to incorporate the new microservice, the redirection URL was defined in the module's code. This could have been done by either a constant in PHP or by just using a local variable in the function that is set to the base URL of the tracking service. To emphasize the importance of the service, it was decided to define the URL as a global constant. The code for that can be found below.

```
//Tracking service link
define( 'MC_TRACKING_SERVICE_URL', 'https://tracking.makecommerce.net
    /' );
```

More specific changes were made to the original function itself. As one of the main problems was the hard-coded array of tracking links for every shipping method provider, this array was deleted and moved to the cloud. Furthermore, the errors and wrong requests will also be handled by the microservice in order to avoid duplicate code between all the modules yet again. Therefore the new functionality will generate a URL based on similar data as it did before. Country and language values are checked and added to the request if possible. This way the service can redirect the client more accurately, just like the previous version in the monolithic architecture did. After a link has been generated for the customer, it will direct them to the new microservice, which in turn will redirect them to the desired shipping provider's package tracking application. The new code that was implemented in the WooCommerce module is the following:

```
    /**
   * Returns the correct link to be used for tracking link
   *
   * @since 3.0.0
   */
  public function get_tracking_link( $carrier, $order_id,
      $shopLocation = false ) {
    // For filtering
    $country = $userLang = false;

    //do we use shop location or delivery location
    if ( $shopLocation ) {
      //get_base_country returns either EE, LT or LV. Everything else
          is irrelevant
      $country = substr( strtolower( WC()->countries->
          get_base_country() ), 0, 2);

      $userLang = $country;
    } else {

      //get order delivery location. Returns EE, LT or LV. Evertyhing
          else is irrelevant
      $country = substr( strtolower( get_post_meta( $order_id, '
          _shipping_country', true ) ), 0, 2 );
      $userLang = substr( strtolower( get_post_meta( $order_id, '
          wpml_language', true ) ), 0, 2 ); //returns nothing if it
          doesnt exist, otherwise returns en, ee, lt, lv, ru
    }

    $params  = [$carrier, $country, $userLang];

    return MC_TRACKING_SERVICE_URL . implode( '/', array_filter(
        $params ) ) . '/';
  }
```

4.2.3 Service Advantages

One of the most significant advantages of the new tracking link microservice is the removal of duplicate code and hard-coded values. The previous tracking system was implemented separately inside modules like the Maksekeskus' WooCommerce and Shopify modules. However, now there is no need to maintain arrays full of tracking links inside the modules themselves but only the code that directs the customers to the microservice. Furthermore, each new module for a different platform can quickly utilize this service and there is no need to write more duplicate code in order to get new integrations to work.

What is more, the tracking link microservice allows for quick modifications inside the service itself, providing very good modifiability. Whenever a shipping provider changes their order tracking URLs, these modifications can instantly be made in the microservice itself and the store owners do not have to update their plugins. Overall this gives much better control over the tracking functionalities code base, allows for cleaner code throughout the modules, quick responses to potential issues, and allows easier monitoring due to it being a separate and fully modular service.
Lastly, the tracking link service allows further expansion into a bigger separate product. There are many possibilities for providing such services for all kinds of shipping orders. Integrating more shipping methods into the microservice and having it publicly available can help with brand exposure. If the product gets bigger and produces enough traffic, it could be made into a paid service, which could then be offered to all kinds of e-stores to help their customers look up their order statuses. Even a free service on platforms other than Maksekeskus' helps highlight the company's brand and potentially attract more clients.

5 Discussion

This section covers the afterthoughts of the software architecture analysis and partial migration to a new architecture. Subsection 5.1 will reiterate the results and compare them to the goals set for the thesis. It also references the work to the theoretical base that was used, how it relates to other literature, and the possible applications that the results of this thesis could have in the software engineering field. Subsection 5.2 covers the drawbacks of the proposed architectural solution and what could have been done better. Moreover, it discusses the limitations of the resulting hybrid architecture and the overall work of the thesis. Subsection 5.3 covers the improvements that could be made over the solutions that were created in this thesis and suggestions for further research in a similar field.

5.1 Interpretation of the Results

The thesis at hand set a goal to analyze the current software architecture of Maksekeskus' WooCommerce module and compare it to other theoretical solutions that either include a monolithic architecture approach or a microservices architecture. In order to analyze the middle ground, a theoretical hybrid solution was proposed as well. The second goal was to practically implement a part of the new architecture if any of the proposed theoretical solutions were to be more beneficial than the existing monolithic architecture.
The thesis used SAAM methodology in order to analyze the three different architectural solutions that were proposed within the thesis. These included the existing monolithic architecture, a hybrid architecture, consisting of a few microservices and a monolith, and a microservices architecture, which has most of the module's functionality as microservices and only WooCommerce-specific parts as a monolith. The results of the analysis showed, that the two theoretical solutions were superior to the currently used architecture. However, the evaluation set both the hybrid architecture and the microservices architecture to be a suitable option. Therefore other factors had to be considered and in the end, it was the difference in the number of resources and time needed to implement the solutions that lead to the hybrid variant being the favorable architecture.
The hybrid architecture, consisting of a few microservices and a monolith, was chosen as the architecture to migrate to and the second goal of this thesis was to implement a new part of this plan. The chosen part was the tracking link microservice. To show that the analytical results of this thesis were taken into account and implemented practically, the shipping tracking functionality was removed from the monolithic part of the existing architecture and moved into the cloud as a separate service. It will provide all the modules with a redirection service, that allows for quick implementation of shipment tracking in new modules and provides better code quality and modifying speed to the existing modules. The static code that is the URLs of all the shipping providers can be updated in real time and does not require the merchants to update their installed module versions.

The advantages and disadvantages of microservices were briefly described in section 2.2 of the thesis. The resulting architecture brought out both of these quite well and demonstrated the need for working through theoretical literature and materials. Microservices offer easier development, understanding of the code base, and separate deployments in an isolated service environment. All these apply to the tracking link service that was created as well. Working with the service is easy since the code base is very small and all the code changes affect only the service itself. New versions can quickly be deployed without having to make any changes to the rest of the modules.
However, the disadvantages emerge as well. Setting up the microservice was more complex than just creating the same functionality inside the monolith. New infrastructure needed to be set up and it required configuring. It also adds another point of failure and the need to create separate monitoring and testing for it. What is more, the service itself required some organizational changes and adjustments in the way that the service is developed compared to developing the modules. Lastly, the infrastructure that the service uses brought costs that did not exist before. As a fully monolithic application, the WooCommerce module was fully on-premises and installed on the e-store's server. With the addition of the tracking link microservice, new cloud costs were added.
The results of the analysis gave a good real-world example of why many microservices might not be the best solution. Even though the resulting architecture has microservices included, the main part of the system was still best left as a monolith. This thesis is an example where a monolithic architecture majority is better than a fully microservices one and other works seem to conclude the exact opposite. Furthermore, since there does not seem to exist a specific theoretical analysis of WordPress plugins and their possible migration into the cloud, this thesis could be an example of how to successfully migrate a part of a plugin into the cloud.

5.2 Shortcomings of the Proposed System

The current monolithic architecture of the WooCommerce module creates no financial obligations to Maksekeskus. The module itself is a monolith that is installed on the e-store owner's server. However, with the hybrid architecture solution, which contains a few microservices in addition to the monolithic part, will come infrastructure maintenance costs. Each service will create expenses regarding its usage amount and efficiency. These expenses were not present before and therefore the new architecture has to balance them out with its advantages.
What is more, the new hybrid architecture system is more complicated. The communication between the microservices and the monolithic part has to be thought through and fault detection needs to be implemented in order to avoid halting the work of certain services due to one of them being non-functional. Therefore there are more points of failure and further tests have to be implemented in order to detect issues early on. This in turn creates a longer implementation time and ties up resources that have to be put

into the development and environment configuration.
Moreover, developing microservices requires some changes in the overall structure of the company as well. As described in section 2.2.3, each service should have a specific team working on them. As Maksekeskus has very few developers then it might be required for each of them to work on each microservice. This in turn could lead to confusion and complexity within the code. To make this architecture work, internally there have to be certain modifications in the way that these services are developed and the adherence to best development practices is more important than ever.
Lastly, while theoretically, the new hybrid architecture is the best possible solution for Maksekeskus, the practical benefits will not materialize until it is fully implemented. Therefore the time and resources put into this project could be wasted if it turns out to be the wrong choice in the long run. There is no way of exactly knowing the outcome of the actual implementation and therefore there is a possibility that an architectural change is a mistake.

5.3 Lessons Learnt and Points of Improvement

This section provides an introspective examination of the research journey, drawing from the challenges and successes encountered throughout the analysis. This critical analysis not only serves to shed light on the overall research experience but also aims to contribute to the ongoing discussion of choosing the best software architecture.
There are many different software analysis methods available and it is almost impossible or at least very time-consuming to consider as many as possible. This thesis briefly introduced three different methods and SAAM was chosen for the analysis of the architectures. Taking more analysis methods into consideration and comparing them more in-depth in upcoming theses can help make more accurate and meaningful evaluations. Some of these methods would of course need a bigger team for analysis which was not possible for this thesis.
Moreover, while there are multiple other analysis methods for software architecture evaluation, there are many different architectures as well. Microservices and monolithic architecture are just a few of them. However, as with the analysis methods, it would have been very time-consuming to evaluate much more of them. In future theses, it would be wise to create a broader understanding of more possible architectures and briefly analyze their advantages and disadvantages. Afterwards, the inadequate architectures can be left out and a thorough analysis can be done with just a few architectures. While it was agreed upon to analyze microservices and monolithic architectures in this thesis, perhaps some other overlooked architecture type would have been even more efficient and beneficial.
Lastly, while the description of the candidate architectures is important, it could be done in a way that is easier to comprehend. Design class diagrams are meant to give a deeper understanding of the system as the classes and their overall structure are described.

However, when comparing theoretical architectures, the structure of the classes is not yet set. The important thing is to know how and which parts of the systems are cooperating. As can be seen from the design class diagrams that were created in this thesis, the specific structure inside a microservice was not shown as it is yet to be implemented but more importantly, the communication and association were shown. Therefore simpler models like a domain model could give subsequent theses the same output while providing less unnecessary information.

6 Conclusion

The primary objective of this thesis was to analyze the current architecture of Maksekeskus' WooCommerce module and two other candidate architectures in order to find the best-suited solution. And if the analysis concluded that a new architecture would be better, then another goal was to partially implement the said architecture.
As a result of the SAAM analysis, it was concluded that a hybrid architecture combining a monolith with a few microservices would be the best solution for the WooCommerce module. This architecture was then partially implemented by extracting a tracking link microservice from the original monolithic architecture. The tracking link microservice helped improve the modifiability, modularity, and development speed of the module and reduced the amount of duplicate code inside all of the Maksekeskus' modules.
A large part of the literature that was presented and used to describe the theoretical background of this thesis concluded that the microservices architecture would be beneficial in many cases. However, this thesis proved that while some microservices are beneficial, a total overhaul and migration into the cloud and independent services would not be advantageous. Therefore this thesis provides a good and uncommon example of an in-between architecture solution for WordPress plugins.
For further research on the same topic, more analysis methods should be considered. What is more, this thesis only covered monolithic, microservices, and hybrid architecture. More and different architectures could be considered for writing forthcoming papers, as there may be other solutions that could prove to be more beneficial. The results of this thesis are inherently constrained by the selected analysis method and candidate architectures. Hence, broadening these parameters could lead to different outcomes and deeper insights.
Overall the thesis reached the goals set and managed to present a new architecture model for Maksekeskus' WooCommerce module. This will improve the efficiency and resource usage within the company. While there were certain aspects that could have been approached differently, the results proved to be useful for the module's development and can act as an example for future theses.

References

[ADM18] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154. IEEE, 2018.

[atl23a] How to build microservices. https://www.atlassian.com/microservices/microservices-architecture/building-microservices, 2023. Accessed on 20.07.2023.

[atl23b] What are microservices? https://www.atlassian.com/microservices, 2023. Accessed on 20.07.2023.

[BGT16] Björn Butzin, Frank Golatowski, and Dirk Timmermann. Microservices approach for the internet of things. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–6. IEEE, 2016.

[BL13] Thomas Boillat and Christine Legner. From on-premise software to cloud services: the impact of cloud computing on enterprise software vendors' business models. *Journal of theoretical and applied electronic commerce research*, 8(3):39–58, 2013.

[BLBvV04] PerOlof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. Architecture-level modifiability analysis (alma). *Journal of Systems and Software*, 69(1-2):129–147, 2004.

[BOP22] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022.

[CLL17] Rui Chen, Shanshan Li, and Zheng Li. From monolith to microservices: A dataflow-driven approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475. IEEE, 2017.

[DL19] Lorenzo De Lauretis. From monolithic architecture to microservices architecture. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 93–96. IEEE, 2019.

[DN02] Liliana Dobrica and Eila Niemela. A survey on software architecture analysis methods. *IEEE Transactions on software Engineering*, 28(7):638–653, 2002.

[doc23] Docker overview. https://docs.docker.com/get-started/overview/, 2023. Accessed on 15.04.2023.

[EFLR14] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Developing the uml as a formal modelling notation. *arXiv preprint arXiv:1409.6928*, 2014.

[Gli07] Martin Glinz. On non-functional requirements. In *15th IEEE international requirements engineering conference (RE 2007)*, pages 21–26. IEEE, 2007.

[GZ20] Konrad Gos and Wojciech Zabierowski. The comparison of microservice and monolithic architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153. IEEE, 2020.

[IHO02] Mugurel T Ionita, Dieter K Hammer, and Henk Obbink. Scenario-based software architecture evaluation methods: An overview. In *Workshop on methods and techniques for software architecture review and assessment at the international conference on software engineering*, pages 1–12. Citeseer, 2002.

[KKB+98] Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, and Jeromy Carriere. The architecture tradeoff analysis method. In *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No. 98EX193)*, pages 68–78. IEEE, 1998.

[KMM17] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. Challenges when moving from monolith to microservice architecture. In *International Conference on Web Engineering*, pages 32–47. Springer, 2017.

[kub23] Kubernetes overview. https://kubernetes.io/docs/concepts/overview/, 2023. Accessed on 15.04.2023.

[LBVVB02] Nico Lassing, PerOlof Bengtsson, Hans Van Vliet, and Jan Bosch. Experiences with alma: architecture-level modifiability analysis. *Journal of systems and software*, 61(1):47–57, 2002.

[MBMR21] Nabor C Mendonça, Craig Box, Costin Manolache, and Louis Ryan. The monolith strikes back: Why istio migrated from microservices to a monolithic architecture. *IEEE Software*, 38(05):17–22, 2021.

[NBC+03] Robert L Nord, Mario R Barbacci, Paul Clements, Rick Kazman, and Mark Klein. Integrating the architecture tradeoff analysis method (atam) with the cost benefit analysis method (cbam). Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 2003.

[New19] Sam Newman. *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[PXW13] Claus Pahl, Huanhuan Xiong, and Ray Walshe. A comparison of on-premise to cloud migration approaches. In *European Conference on Service-Oriented and Cloud Computing*, pages 212–226. Springer, 2013.

[TLPJ17] Davide Taibi, Valentina Lenarduzzi, Claus Pahl, and Andrea Janes. Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In *Proceedings of the XP2017 Scientific Workshops*, pages 1–5, 2017.

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Siim-Morten Ojasalu**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Reconstruction of a monolithic application within Maksekeskus AS,
   (title of thesis)

   supervised by Dietmar Pfahl and Reelyka Läheb.
   (supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe on other persons' intellectual property rights or rights arising from the personal data protection legislation.

Siim-Morten Ojasalu
11/08/2023