WIRTSCHAFTSUNIVERSITÄT WIEN
Vienna University of Economics and Business

Master Thesis

Title of Master Thesis:	Design decisions to migrate a Monolithic software architecture to Microservices-Based architecture
Author (last name, first name):	Petrov, Petar
Student ID number:	h 11729352
Degree program:	MSc Information Systems
Examiner (degree, first name, last name):	Prof. Dr. Jan Mendling

I hereby declare that:

1. I have written this Master thesis myself, independently and without the aid of unfair or unauthorized resources. Whenever content has been taken directly or indirectly from other sources, this has been indicated and the source referenced.

2. This Master Thesis has not been previously presented as an examination paper in this or any other form in Austria or abroad.

3. This Master Thesis is identical with the thesis assessed by the examiner.

4. (only applicable if the thesis was written by more than one author): this Master thesis was written together with

   The individual contributions of each writer as well as the co-written passages have been indicated.

14.10.2019
Date

[illegible]
Signature

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

EFMD EQUIS ACCREDITED

Master Thesis

Design Decisions to Migrate a Monolithic Software Architecture to Microservices-based Architecture

Petar Petrov

Date of Birth: 17.07.1993
Student ID: 11729352

Subject Area: Information Business

Studienkennzahl: h11729352

Supervisor: Prof. Dr. Jan Mendling

Co-Supervisor: Dipl.-Ing. Philipp Waibel

Date of Submission: 15.October 2019

Department of Information Systems and Operations, Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria

DEPARTMENT FÜR INFORMATIONS-VERARBEITUNG UND PROZESS-MANAGEMENT DEPARTMENT OF INFORMATION SYSTEMS AND OPERATIONS

Contents

List of Figures

List of Tables

Abstract

Nowadays, as the users started to require higher speeds, constant updates and no downtime from the applications, the software systems had to evolve as well in order to meet the demands. Each system is required to have increased scalability and robustness. The amount of users that have access to internet has grown significantly during the recent years and naturally, this has led to higher internet traffic. The applications now need to offer their service to substantially greater amount of users at the same time, in comparison to the previous numbers. However by using the old software development patterns this seems like an impossible task. The way the earlier software was developed, was designed to handle much fewer requests than today. Nevertheless, the good news is, that the IT is a fast-paced sector and new technologies are emerging constantly. New software architectures and paradigms emerge and new ways to create software systems are developed. A perfect example for the evolution in the field is the microservices architecture. It is addressing many of the issues of the previous standard - the monolith. It is rapidly getting adopted by not only the leading software companies, but the smaller ones as well. Furthermore, this is the go-to pattern when one wants to offer something on the cloud, which has become the easiest way for the users to use some technologies offered by a provider. In this master thesis we will provide information about both the approaches, including use-cases and comparison between them. The main goal of the thesis will be to provide an approach to migrate the legacy monolithic software architecture to the state of the art microservices. In order to provide meaningful examples, we will use the open-source ERP system Odoo, which will make it easier for the reader to understand the concepts.

Keywords: Microservices, Monolithic, Software Architecture, Cloud, Scalability, Decomposition, Migration, ERP System

1 Introduction

In order to have a competitive application nowadays one of the most important factors that needs to be achieved is the periodical release of updates. The customers always need something extra, or need something to be removed, or maybe there are some bugs in the system that need to be addressed. At the end, the success of the company is depending on these updates. The problem however, is not the writing of the code, but its deployment. The bigger the system, the harder the deployment of new releases becomes. Until recently, the software systems tended to grow unobstructedly with time and, therefore, became increasingly harder to maintain. The monolithic software architecture has been around for a long time. As is the case with the word monolith, when used in the context of construction, it refers to the same when it comes to software. We can imagine a large, single block of code [1, 2]. And that is where the issues with the deployment of new functionalities come from. By using the monolithic architecture design, this means, that the entire software system needs to be stopped and updated, but this is not all. If a single newly-introduced database constraint or line of code is interfering with the already existing system, the entire system fails. This leads to downtime, rollbacks, loss of information and customer dissatisfaction.

Besides the constant need of updates and the difficulties they bring along, the requests that the applications are serving has grown exponentially during the recent years. The web servers are getting overloaded and need load balancing. Furthermore, the database can act as a bottleneck if many requests need to be served at the same time. Another significant change, when we are comparing the current situation with the one that was a few years ago, is the offering of more and more cloud computing and storage services [3]. Companies are offering their resources on-demand to users, who can just access the respective service they need, get their job done and close it. No need to install applications, buy expensive servers with large storage and computing capabilities and maintaining an infrastructure. The only thing that the user has to do in exchange for using the selected service is to pay for it depending on the rate. We will look into the cloud services in the next section.

We could continue enumerating examples of the evolution of the software environment, but the mentioned ones will serve the purpose to prove our point well enough. All of the quoted cases share a common issue - the inability to be realized using the monolithic software architecture pattern. Therefore, we can clearly see the need of a new approach, which will address these issues and will provide a more agile and error-resistant way to deploy code and update the system regularly. That is why the microservices architecture was invented and has become a de facto standard for the cloud solutions nowadays. Essentially, microservices are small independent services, which communicate between each other through API calls [4]. Each of these services can be updated separately and without interfering with the others, which reduces the risk of system failures. Another big advantage, that comes with the microservices is, that they can be scaled independently, based on current traffic and needs. Furthermore, they enable us to create more understandable and robust software systems [5, 4].

1.1 Motivation

With everything mentioned, we can clearly see, that the microservices architecture has multiple advantages over the monolith and will assume, that when we want to design our future system, it will be based on this methodology. However, there are too many legacy systems using monolithic design, which we cannot just leave out and substitute with fresh, new systems on their places. We must instead find a way to migrate the software, which includes all components - from backend to frontend and make it comply with the microservices architecture. There lie many challenges in the process of transition and, therefore, this is an interesting topic to look into. Usually just a plain database migration, or update to a newer version of the same software may cause trouble. In most of the cases these actions lead to inconsistencies and conflicts. They require an entire team of developers dedicated for this purpose to work only on the migration, test and debug the possible problems. And this comes just after updating the same software to a newer version. The topic, that this thesis is concerned with goes one step further - we will look into the migration from one architecture to another.

This is something, that many companies and developers will benefit from and is a rather new research topic. Nevertheless, there has been prior research in the topic recently, as the microservices have boomed in the last few years and this is an intriguing field for research (e.g., [6, 7]). However, with this master thesis, we will take a new approach and will expand on the research, that has been made already. We will provide in-depth information about all the concepts and will use examples in order to provide a proof of concept for the thesis.

In order to provide use-case examples, we have decided to use an ERP (Enterprise Resource Planning) system as a reference. It has the required depth and most of the companies are still running legacy monolithic systems and are scared to switch to new ones. The reason is, that the systems are usually quite large in size, with sensitive information kept in the database, which may be affected in the migration. Furthermore, as companies grow larger, the need for improvement of their internal systems increases as well. The examples using an ERP system are convenient, because we can clearly identify the various components which correspond to the different processes inside the specific company.

In addition to all the reasons mentioned up to this point, we believe, that the entire field of microservices is still in its early stages of existence and will continue to affect the software environment in the future. Therefore it would be useful to gain deeper understanding for the topic and contribute to it.

1.2 Research scope and question

The research question, that this thesis will work on is: How to migrate the legacy monolithic software systems to the more flexible and scalable microservices? The goal of this research will be to develop a new approach, which will take into consideration prior research in this topic and will use the findings as a foundation for ours. The final solution will be based on documented, scientifically sound arguments.

There are multiple points that have to be taken into consideration and we will cover each and every one of them with the required depth:

- Background Knowledge
- State of the art approaches
- Database Migration
- Code Migration
- Maintenance

After reading the thesis, the reader will have the required knowledge to migrate any given monolithic software system to microservices. The methodology section will include a step-by-step solution to the problem with recommended technologies for each step. By applying the suggested actions, using some of the proposed technologies from the technology stack, a transition can be made without loss of information and functionality.

2 Background Literature

In this section we will provide relevant information about both the architecture styles - the monolith and the microservices. It will serve as a domain analysis and will provide the information required for the reader to understand the topic. We will start with the monolithic approach and then will move on to explain about the microservices. Regarding the monolithic design, we will only provide information about the most important aspects and will show some examples with some well-known systems. In contrast, regarding the microservices, we will give more thorough examination and will dive deeper into the topic. We will analyze different patterns and rules, that need to be followed in order to achieve the desired outcome - to have an efficient software system, which is built using the microservices architecture style. Furthermore, we will give insight into the cloud computing and the CI/CD (Continuous Integration, Continuous Deployment) concept. These are two topics, which are closely related to the microservices and are being enabled by this architecture type. At the end of this section, the reader can find a comparison between both architecture styles, as well as a list of advantages, drawbacks and suitable domains where they should be applied.

2.1 What is a Monolithic Software Architecture?

The way software was build during the years of software development has evolved substantially from the beginning, with multiple patterns changing along the way. A good example for this is the way a software program is organized - decades ago the programming languages used the procedural programming paradigm [8, 9]. However, as programs evolved this approach proved to be limited in terms of reusability, abstraction and encapsulation. The writing of large programs became more and more cumbersome and challenging task, because of all the interdependencies and relations between the procedures. Therefore, a new concept has emerged, driven by the necessity of faster and easier programming - the Object Oriented Programming (OOP). Nowadays most of the programming languages use the OOP principles and it has become the standard to write a computer program [8, 9].

A further example of the evolution of software can be found in the database patterns - Relational DBs vs NoSQL. As data became more and more crucial for the enterprises, the relational databases proved to be the bottleneck and also needed much more storage capacity in order to hold the same amount of records. Therefore, slowly we can see that NoSQL started to gain pace and is becoming the standard for Big Data applications [10, 11].

However, after all these changes, one thing remained pretty much the same since the beginning, with very small changes introduced - the architecture design of the entire software system. The traditional monolithic architecture is still pretty much in every enterprise [12]. There are various reasons for that, but the most obvious one is, that the systems were designed when there were no microservices. Furthermore the monolith is not necessary a bad approach. In fact it can serve multiple purposes and has its advan-

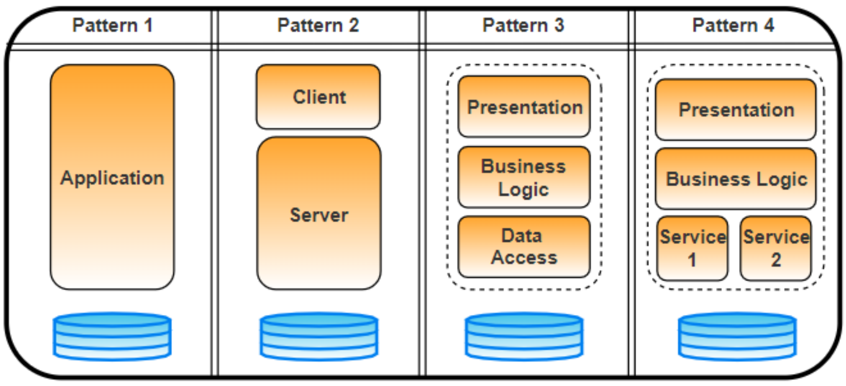

Figure 1: Types of Monolithic Software Systems

Source: Adapted from [2]

tages, which we will discuss later. Now, lets focus on understanding what the monolith is and how it is working.

The first thing that comes to mind when thinking of the word monolith, is a large single block of stone, which is used in the construction. In a sense, in software development, we can think of the monolithic architecture in the same way - a single block of tightly connected layers of code working together to provide different functionalities [1]. In a paper from 2018, the authors identified four different patterns of monolithic systems [2]. As this is not the prime topic of this research, we will not go in-depth with all of them, except with the most widely-used one - the three-tiered application design, which is Software System 3 on Figure 1.

There is a reason why this approach is called three-tiered. It is because the entire architecture is comprised of three layers of logical computing - presentation tier (front end), application tier(business logic) and data tier. This approach is used regularly in applications, which are working as client-server solutions [13, 14]. Each of the layers has its own functions, responsibilities and strict boundaries:

- **Presentation Layer:** "This is the front end layer and consists of the user interface. It is usually a graphical one and accessible via a browser" [13]. The most widely-used technologies are HTML, JavaScript, PHP and CSS.

- **Application Layer:** On this layer is contained the business logic. This is the heart of the application. All the computations and operation logic of the program are performed here. Furthermore on this layer

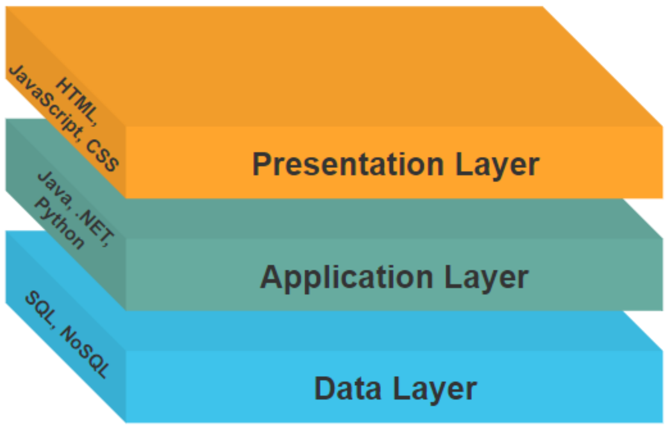

Figure 2: Three-Tiered Architecture

Source: Adapted from [13]

we can find the logic for interaction with the other two tiers and for transforming the data as requested by the user [5]. For the purpose of this layer, a wide variety of programming languages can be used i.e. Java, C#, Python and etc.

- **Data Layer:** This is the bottom layer of an application and is responsible for the communication with the database, storing and receiving information. The most notable programming language here is SQL.

The base operational logic of an application built using the three-tiered architecture will remain the same, no matter the complexity of the system: The Application Layer delivers data to the web browser, operating at the Presentation Tier, which provides the means for users to request information from the application, view it, and (usually) manipulate it. When the Application Layer receives a request from the Presentation Layer it processes the information and returns it again to the user for viewing. If the request contains something related to the database (change or receive something from it), it pre-processes the information and then relays it to the Data Tier. The Data Tier level is where the connection to the database is maintained, the query is executed and the result is passed again to the Application Tier. Afterwards, possibly more computations are performed, based on the request from the user and the result is returned to the Presentation Tier.

A very simple example of the operation of such architecture could be a simple login system of a random website. When a user first accesses the page, he is prompted to enter his credentials - this is the Presentation Layer.

Hospital
Ward
Doctor
Patient
Appointment

Figure 3: Domains in a Medical Software System

Source: Adapted from [1]

Then after we pass our data, it is conveyed to the Application Layer. There we may perform different checks and operations e.g. suggest to send the user a password recovery email, if the credentials were wrong. All the user information - the emails, usernames and passwords is stored in the database, which is accessed via the Data Layer of the application.

After we explained the logic of the three-tiered architecture, let's now discuss the organization of the code in the monolith. As the name suggests, everything is closely connected and the code is put into one place, but we can distinguish several structuring patterns: Module Monolith, Allocation Monolith and Runtime Monolith [15]. The most widely-used one is the Module Monolith pattern [1]. By adhering to it, "we have all of the code for a system in a single codebase, it is compiled together and produces a single artifact." [15] This naturally means, that if there is a single error, the entire system will fail when compiling. Nevertheless, "the code may still be well structured (classes and packages that are coherent and decoupled at a source level) but it is not split into separate modules for compilation" [15]. The other two patterns, that are identified are useful, depending on the situation, but this paper will be focused on the Module Monolith and therefore we will provide more information about it.

In order to grasp the concept, we will use a simple example with a medical software system, which was borrowed from [1]. In general, when a patient wants to go to the hospital, he is making an appointment with the doctor. Figure three depicts the different entities in the current system domain. We can see, that there are clear functionalities, with which each role is concerned. By identifying the various sub-domains, we can structure our code better and achieve higher cohesion and lower coupling [16], than if we were to put the entire codebase in the same place. Therefore, by following this logic, we would put our sub-domains in separate modules. This would make it easier for us to maintain the code and introduce new functionalities by having the connected code elements on the same place. Furthermore, the testing will be improved by using this approach, because we will have smaller, more isolated units [1].

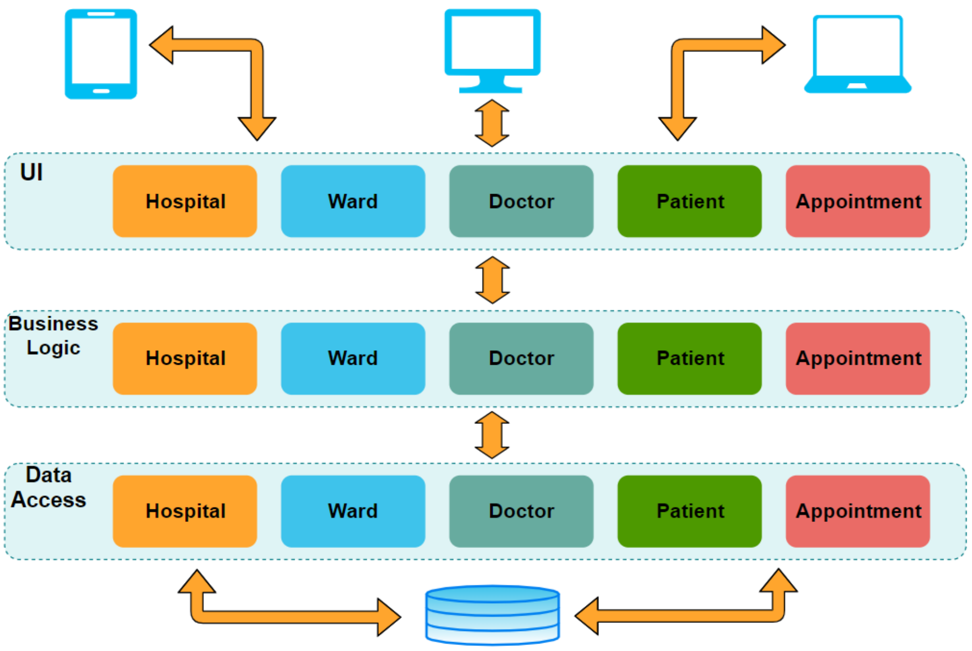

Figure 4: Example of Monolithic Medical Software System

Source: Adapted from [1]

However, as we can see on Figure 4, the system is using a three-tiered architecture and we need to access a database, perform computations and interact with the end-user. Therefore, we will be able to find all of the three-layer code in the specific module. There will be parts responsible for the database, application logic and front-end. This could prove to be a problem, if we decide to make some changes. Imagine there is a new business requirement and we have to introduce new functionality in the Appointment module. We would not be able to build only it, but instead all of the other modules along with it [1]. Aside from that, we can assume, that with time, the codebase will grow. The modules will become less and less decoupled and will be increasingly dependent on each other. As a result, if we were to make a change in the Doctor entity, this would most likely affect all the others in a way and we must watch out for the possible collisions and the errors they may lead to.

The entire concept of the monolith is not that bad, but the problem is that it is just outdated [5]. Nowadays people have multiple devices, each one of them running on a different OS, resolution, web browser and etc. And by using the monolith, we create the entire block of code, compile and deploy it. As a result, all the different devices will have access to all the resources, even the ones, that are not meant for them. For example, a smartphone user will have access to the functionality of the PC user. This leads to ineffective usage of resources and lower level of security and isolation [5].

The biggest problem however is the continuous deployment of new releases [17, 18]. We can have the entire codebase distributed in modules, or not, but this does not matter, as the entire application needs to be rebuilt when we release new functionality. Even when updating a single line of code, we have to restart the entire application and most of the times also update the database in order to apply the changes we made. This hides many risks and requires testing of the entire application, because this added line may be related to some other part of the program. If an exception occurs during the build there is a big chance for it to lead to system failure, depending on the exception, which may prove to be fatal in a production environment.

Another problem with the monolith is the scalability [19]. Here we are not able to just scale some parts of the system, which are getting the highest traffic loads, instead we need to improve the entire application scale, which is a costly and unnecessary solution, if we have another option.

2.2 What is a Microservices Software Architecture?

In the previous section, we have provided information about the monolithic approach. We can see, that aside from its benefits, it has multiple flaws, which needed addressing. Nevertheless, the software world is progressing with enormous pace and there were multiple approaches, that tried to fix some of the problems, which arose when using the monolithic design. One of the most successful architectures in the last twenty years was the Service Oriented Architecture (SOA) [20]. It addresses many of these imperfections of the monolithic approach.

"SOA is an architectural style whose goal is to achieve loose coupling among interacting software agents. A service is a unit of work done by a service provider to achieve desired end results for a service consumer. Both provider and consumer are roles played by software agents on behalf of their owners" [21, 20]. SOA focuses on defining separate services for the various business requirements and processes. We can recognize two main roles within the SOA: [21]:

- **Service provider:** "The service provider is the maintainer of the service and the organization that makes available one or more services for others to use" [23].

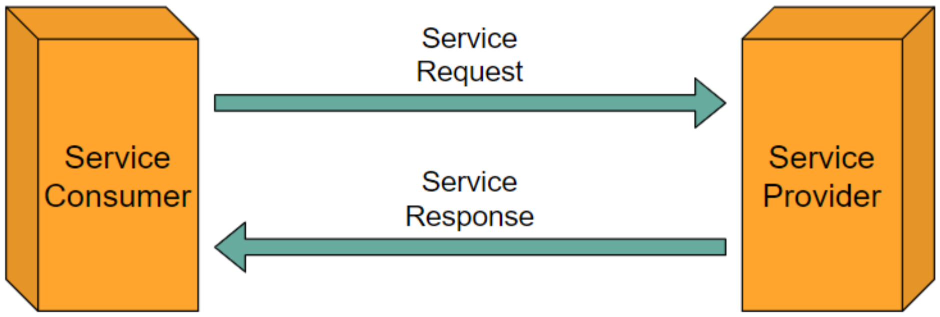

Figure 5: Service-Oriented Architecture

Source: Adapted from [22]

- **Service consumer:** The service consumer requests information from a service provider. The information can be of different size and complexity. A usual interaction between both of service types is as follows: A request is passed from service consumer to service provider. The service provider performs the required operations and returns the requested information back to the consumer. A service provider can also be a service consumer and vice-versa [22].

"Unlike traditional object-oriented architectures, SOAs are comprised of loosely joined, highly interoperable business services" [20]. By having these decoupled and separate services we can have proper stand-alone modules. These modules are not depending on the others and don't share the same codebase. Instead they are communicating between each others with service calls. This is already an improvement over the monolith, since we can now isolate the different sub-domains to a greater extent than before. SOA is the natural transition and upgrade to a monolithic architecture.

However it comes with a design flaw of its own - the service bus [24]. "It is a software architecture which connects all the services together over a bus like infrastructure. The bus acts as communication center in the SOA by allowing linking multiple systems, applications and data and connects multiple systems with no disruption" [25, 24]. The modules inside the SOA have their own specific functionality, but are all connected and to this aggregation layer (bus). Thus they must all adhere to the same interface standard and are somewhat dependent on the bus. With time, the systems built based on the SOA grew and therefore their complexity increased as well. As a result, the SOA Bus became the biggest challenge to handle. The issue was the addition of the operational logic to the bus. As this layer grew bigger with the

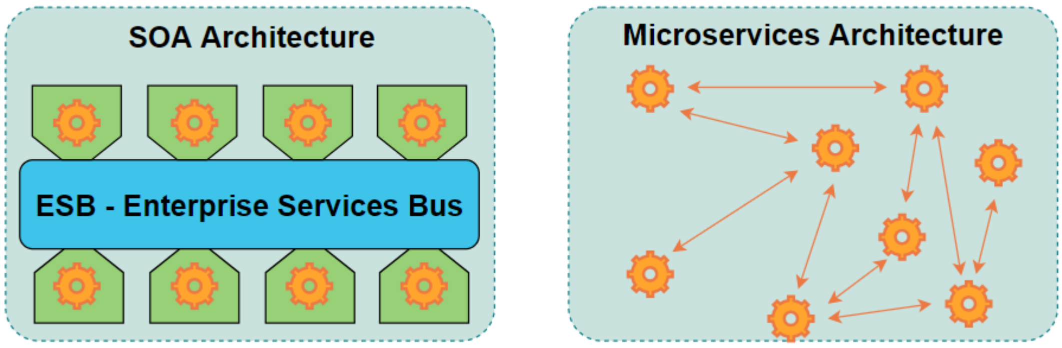

Figure 6: SOA vs Microservices

Source: Adapted from: [27]

addition of more components, so came the issues of system coupling [26]. At the end, the software architects had similar problems as with the monolith, but this time with even increased complexity - they had multiple dispersed services, but in a way they were coupled with the bus. Therefore a new idea started to emerge and to become the standard in the software architectures in the 2010s - the Microservices Architecture.

After discussing the SOA, we will now introduce the concept of microservices and create a differentiation between both the architecture types. Figure 6 shows the main advantage of the microservices - the lack of the service bus. As we can see, in SOA all the services are connected to the bus and are thus dependent on it. In contrast, the microservices are independent and not sharing such an aggregation layer. By removing this bus and decoupling the services, we can achieve better results, when our system grows.

"Microservices are small, autonomous services, that work together" [28]. A microservice's main aim is to be focused on doing one thing only and not being concerned with the functions of the other microservices. The different services communicate between each other with APIs and don't share irrelevant information. This idea can be accomplished by most importantly good initial software design and well-described requirements. The main driver in this architecture is the cohesion - the aim to have related code grouped together. We have mentioned the cohesion and coupling terms a few times already, but what exactly are they?

- **Loose Coupling** "When services are loosely coupled, a change to one service should not require a change to another" [28].

- **High Cohesion** "We want to design components that are self-contained: independent, and with a single, well-defined purpose" [16]. Both of the terms usually walk hand in hand and we are aiming to achieve High Cohesion and Low Coupling. We can't have High Cohesion together with High Coupling and the opposite.

In order to achieve high cohesion and low coupling, we need to have our code grouped according to some standard. One of the best approaches can be found in the book Domain-Driven Design by Eric Evans [29]. The book aims to give directions and serve as a guide on how to build systems based on real-world domains. The most intriguing part is the idea of bounded contexts. "The idea is that any given domain consists of multiple bounded contexts, and residing within each are things that do not need to be communicated outside as well as things that are shared externally with other bounded contexts. Each bounded context has an explicit interface, where it decides what models to share with other contexts" [28]. Put in simpler terms, the author argued, that we should identify different parts of our domain (e.g., business case) and put the similar parts in the same place. Afterwards, we need to identify their interactions and expose only the information, that is needed.

Achieving loose coupling and high cohesion is the main driver in the microservices. This means that the services are not depending so much on each other and addresses the aforementioned issue with the deployment. The same goes as well for the database, in order to reach this state of decoupling, we need to maintain multiple databases [30, 17]. The update of one service doesn't require the restart of the entire system and the other services, instead just this single service, which is being updated. Figure 7 gives us a good idea of the structure of a microservices application. We can see that each subdomain is being assigned to a respective service (e.g., the accounting subdomain has its own service). Each of those services has its own database and is communicating with the services, that it needs to communicate with. The interaction with the client in this case is done via two websites (the shopping website and the internal support website) and all the requests are passing through the API gateway. It serves to direct the different requests to the appropriate services. This is the main concept of how applications built using the microservices architecture operate. We will give more information regarding the mentioned patterns and requirements in the following sections. Instead in this one, we will focus on more general ideas.

An important difference between the microservices and the monolith is the way their tiers are organized. We have already discussed the well-known Three-Tiered Software Architecture, which is used in the monolith. The microservices, however, employ a different way of operation and therefore

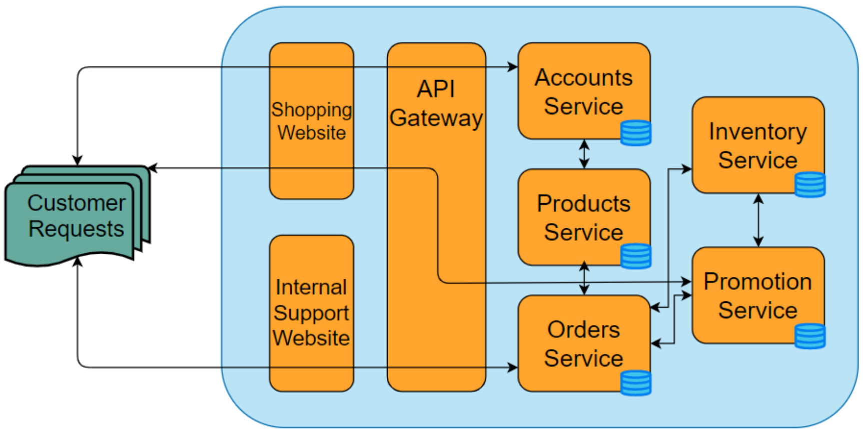

Figure 7: Microservices Architecture Example

Source: Adapted from [31]

need a different approach on organization and the communication between the layers. Figure 8 presents a good overview of the different tiers in the microservices architecture. The most notable difference is, that instead of three, we have four layers here:

- **Client Tier:** "The most dramatic difference in this new model is the client tier, as modern applications need to think about the user-facing layer as its own independent set of functionality that leverages the delivery, aggregation, and services layers beneath it to create device-specific and highly tailored experiences" [5]. This layer is aimed at optimization for all the different types of devices, thus allowing better user experience, with better UI and ad-hoc APIs for each different device type.
- **Delivery Tier:** It "is responsible for optimizing delivery of the digital experience to the user using intelligence received from the client layer" [5]. Information is gathered from the user and the best delivery option is selected by the system. A good example of this layer are the Content Delivery Networks (CDNs) [32].
- **Aggregation Tier:** "An aggregation tier integrates internal and external services and transforms data" [33]. It acts as an equivalent to the application layer in the three-tiered architecture. There is the center of the business logic, performing various computational-heavy tasks

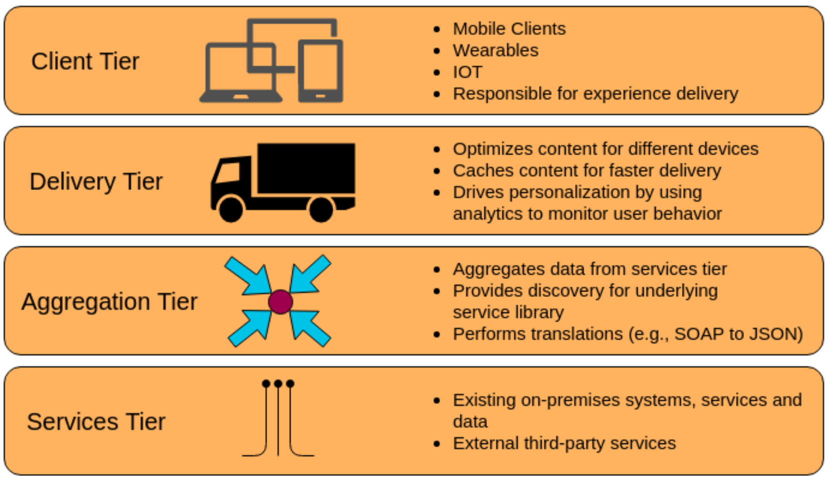
Client Tier
Mobile Clients
Wearables
IOT
Responsible for experience delivery
Delivery Tier
Optimizes content for different devices
Caches content for faster delivery
Drives personalization by using analytics to monitor user behavior
Aggregation Tier
Aggregates data from services tier
Provides discovery for underlying service library
Performs translations (e.g., SOAP to JSON)
Services Tier
Existing on-premises systems, services and data
External third-party services

Figure 8: Microservices Tiers

Source: Adapted from [5]

[5]. An example of applications, which work on this level are business intelligence and analytics.

- **Services Tier:** "This final architectural element dynamically composes data and business processes through a set of continuously deployable services. This tier provides data to the layers above without concern for how that data is consumed" [33]. There is no difference for the services tier , whether data is stored in a structured, or non-structured way. This tier will offer the required functionality to support both the database types and enable the interaction between the database and the system [5].

We will briefly mention some of the advantages and the drawbacks of the microservices here, but a more thorough examination can be found in the dedicated section (Section 2.8). First of all the biggest benefit of all is the scaling. We can scale only those services, that we want to, in contrast to the monolithic applications, where we are forced to replicate the entire codebase [34]. Furthermore, this architecture approach allows for smaller size of the code base, which makes it easier for a developer to understand [35, 17]. Another important aspect is the fault isolation. Here, if a specific service fails, it won't have an effect on the others, as is the case with the monolithic architecture.

Of course besides the benefits there are drawbacks. The most obvious of them is the additional complexity, that the developers have to deal with. This is basically an entire distributed system instead of the good old monolith, where it can be run locally and in the IDE. With the microservices approach, the inter-service communication is the most important aspect and is the hardest one to achieve. The concurrency and synchronization between the different services and respectively their databases is crucial in order to have a well-working system. Nevertheless, after everything mentioned, if we deal with the increased complexity the microservices architecture is the better one for the current and future IT environment: scalable system, variety of devices, users demanding fast delivery of data and regular updates.

2.3 Microservices Patterns

In order to have a properly working system built according to the microservices architecture standards, we have to adhere to some patterns. There are multiple options to choose from when building a software system (e.g. database selection, deployment patterns, service orchestration vs choreography and etc.). In this section we will take a look at some of the most important patterns and will provide reasoning on why it is better to follow them. We will provide just basic information and guidelines, but very well structured and thorough information can be found at the microservices.io website [36] and the book [17].

- **Decomposition Patterns:** After we have decided to setup our system using microservices the first thing we need to do is to make a domain analysis. When we are aware of what we want to have in the system, we need to split it into loosely coupled elements that have bounded contexts [37]. Here are two patterns, which can help us decompose our domain into smaller elements:
  - **Decompose by business capability:** "Define services corresponding to business capabilities. A business capability is a concept from business architecture modeling. It is something that a business does in order to generate value" [38, 17].
  - **Decompose by subdomain:** Define services corresponding to Domain-Driven Design subdomains. The idea is that the entire business area is presented as the main domain. This domain will have some subdomains, which are responsible for different aspects of the business [39, 17]. A simple example with a web-shop is, that the entire shop is the domain. The shopping card is one of the subdomains, the catalog is another subdomain, the Inventory is a third one and etc.

- **Database Patterns:**
  - **Database per service:** One of the most important aspects when dealing with microservices is the database organization. People have tried using different approaches during the years (e.g. shared database [40, 17]), but the Database per service approach proved to be the most successful one. Basically, this pattern refers to using a separate database for each separate service. The data is private for the service and can be accessed only via API calls from the other services. Furthermore a service's transactions only involve its database [30, 17].
  - **Saga:** After we have chosen to keep each service's data private, then naturally will come the problem of how do we keep it consistent across all the services. For example, if a user wants to update his payment information, this information needs to be entered in two separate services - the User Management and the Payment Management. Furthermore, if some changes are made to this data, it again needs to be updated. The solution to this is to "implement each business transaction that spans multiple services as a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions" [41, 17]. There are two ways to keep coordination:
  - **Choreography vs Orchestration:** These two concepts are not used only for the database consistency. In fact, "as we start to model more and more complex logic, we have to deal with the problem of managing business processes that stretch across the boundary of individual services" [28]. "With orchestration, we rely on a central brain to guide and drive the process, much like the conductor in an orchestra. With choreography, we inform each part of the system of its job, and let it work out the details" [28]. There is no better or worse decision when it comes to selecting which of the approaches to use. Instead this is one of the areas where one could choose which suits him more and is more confident of implementing it, but it is good to know the way they operate.
- **API Gateway:** In a microservices architecture, we can expect for client to interact with more than one front-end service [42]. The client

needs to know which endpoints to call, how to reach them and basically to interact with the application. That is where the API gateway steps in. It sits between clients and services and acts as a reverse proxy, routing requests from clients to services [42]. The API gateway is the single entry point for all clients. A variation of this pattern is the Backends for frontends pattern. It defines a separate API gateway for each kind of client [43, 17]. By using such an approach, we will have a separate API for the different clients - Web, Mobile and 3rd Party applications. This will help us optimize our interaction with the clients by introducing a separate customized API for each of them.

- **Service Discovery Patterns:** In the microservices architecture, services usually need to call each other. "In a traditional distributed system deployment, services run at fixed, well known locations (hosts and ports) and so can easily call one another using HTTP/REST or some RPC mechanism. However, a modern microservice-based application typically runs in a virtualized or containerized environments where the number of instances of a service and their locations changes dynamically" [44, 17]. Therefore we need to inform the services about the location of the other services. Typically for that reason we are using a service registry [45, 17]. It is essentially a database of services and holds their locations and instances. Depending on the implementation, services are found in the service registry either by polling by the client [44, 17] or by the server [46, 17].

- **Circuit Breaker Pattern:** The communication between services will become complex, as new requests can appear every second. In case of synchronous inter-service communication, there is always a service consumer and a service provider. Each time a service makes a synchronous call to another service, it passes through a proxy [47, 17]. This proxy is located in the service consumer and counts the number of successful and unsuccessful request attempts. There is a pre-defined threshold for consecutive unsuccessful attempts and if it is reached, the circuit breaker returns an error to the user and issues a timeout period. During this timeout no new requests are initiated and if the service provider was overloaded, may recover and continue operating as usual. However, if such a pattern isn't introduced and the service consumer keeps constantly requesting the information, it will become blocked as well, because will wait for the response. Eventually, a single service failure may cascade to other services and lead to consequences for the entire system [47, 17].

- **Deployment Patterns:** One of the most popular deployment patterns for microservices is the Service Instance Per Container. We will give more information regarding containers in the dedicated section, but for now it will be enough to know, that a container is a lightweight virtualization method. The reason this deployment pattern is so popular, is that it is really simple to scale the services by changing the number of service instances [48, 17]. A service instance is simply a replica of the service, responsible for a single process [49]. By changing the number of the instances, we increase the number of concurrent processes, which can be run simultaneously. In order to achieve this scaling, we can benefit from tools like Kubernetes [50]. They are container orchestration tools, which create clusters of containerized service instances, based on the current load. This method is preferred to other deployment methods as service instance per VM [51, 17], single service instance per host [52, 17] and multiple service instances per host [53, 17], because it uses much fewer resources per instance, as compared to them.

As we mentioned, there are many more patterns, that need to be followed in order to reap the best benefits from the microservices. However, by citing the ones we deem most important, it should be enough to understand the basics and the purpose of this paper. Further information can be found at [36, 17].

2.4 CI/CD

This is the section, where we will discuss one of the biggest enablers of the microservices architecture. By building our system following the microservices standards, we will be able to make use of Continuous Integration and Continuous Deployment or in short CI/CD. It is a concept, that lays out some practices to follow in order for the code that is written to arrive more quickly and safely to the users and ultimately add value [54, 55]. As we can see, there are two sides of this methodology, which are tightly connected. We will start by looking into the CI side first and then will continue with the CD.

Here is a CI definition by ThoughtWorks, which encompasses the term very well: *"Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early. By integrating regularly, you can detect errors quickly, and locate them more easily."* [56]. The keyword in this definiton is auto-

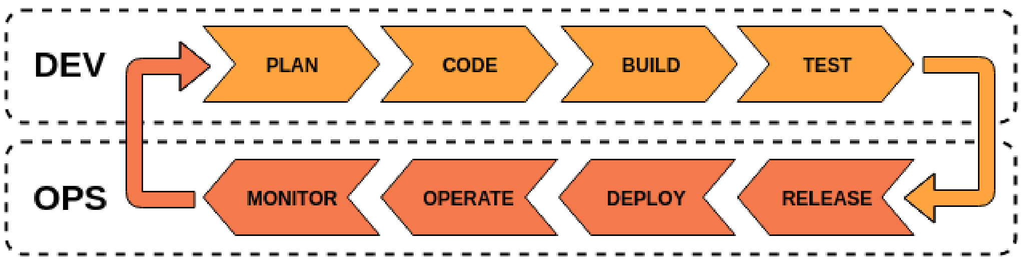

Figure 9: CI/CD and DevOps

Source: Adapted from [57]

mated build. The code is automatically built after a commit is made and afterwards some pre-defined automated tests are run. The test coverage report is published and if it is deemed passed, we can continue with this build, if not - we revert the changes. Another important aspect is, that all the members of a team are using the same code repository [57]. This repository is usually the master branch and that is where all the new changes are added daily. This is in contrast to the monolithic deployment principles, where multiple branches are used for development, staging and production. Instead in CI practice, the master branch is used to push code into, because that's the branch that's going to be used to release software. The "push to master" stage is also known as trunk-based development. We will not dive into the topic any further, as it is not the focus of this paper, but you can check the reference for more information [58].

On the other hand, "the Continuous Delivery aims to get changes of all types into production as quickly and safely as possible" [57]. This is achieved with the help of CI, after all the integration tests have been passed and the code is in deployable state. The CI is a responsibility of the developers within a company and the CD - of the operators [57]. Everyone, who has been involved in a software development project is aware, that the deployment of the code can be hard. There are many issues that can arise when we deploy new releases in a monolithic software system. That is why, a preferred approach used to be to have fewer deployments, distanced from each other [57]. However the longer we wait, the more changes we tend to deploy at once and this increases the possible risks for the system. In contrast to that, the CI/CD's idea is to have many small releases, which reduces the risk of rollbacks. On top of that, the deployment is done with as much automation as possible, which further simplifies the process and reduces the room for error.

During the recent years, the mixture between both of the methodologies has opened a new position in the software world - DevOps. Usually the DevOps engineers are responsible for the development lifecycle in the company. We can see in Figure 9 the actual lifecycle of the daily releases. It all starts with planning, moves on to coding, building and testing the new functionalities. These four general steps are part of the Continuous Integration and respectively the Development part of DevOps. Afterwards, the process is passed to the Continuous Deployment side of the process with the release, deployment and monitoring of the changes. The most widely-used CI/CD software tools are: Jenkins [1], Gitlab CI [2] and Azure Pipelines [3]. They offer their users various benefits, but most importantly automated pipelines, that make the entire process easier and faster.

To wrap it up, the entire idea of the CI/CD is to work in small batches with as much automation as possible. That way we can have fast everyday releases of new releases and bug-fixes, which improves the entire system. The CI/CD pipelines started to gather pace with the introduction of the microservices and the two concepts go hand in hand. It is easier to maintain such a methodology, if the system is split into smaller pieces. That way the deployments are affecting only a single piece of it and the tests that need to be performed are smaller and therefore faster. One can also try to adopt the CI/CD for a monolithic application, but it will not be that effective [59].

2.5 Containerization

CI/CD is a useful concept, which is helping us realize microservices architecture, but it wouldn't be nearly as effective, if the idea of containerization was not introduced. A crucial aspect of the microservices architecture and the CI/CD methodology, is to have each service running in a separate environment. This is addressed with putting the services in different containers, holding the dependencies required to run the service without external intervention. However, before discussing containerization, it is important to understand the concept of virtualization. Virtual machines (VMs) have been around for quite some time now and have become a popular solution for a wide range of infrastructure problems. A VM is essentially a separate Operating System (OS), which runs on top of the host OS [60]. The host OS allocates a specific amount of hardware resources, which will be available to access from the Guest OS. This Guest OS can be used for a number of purposes and is logically separated from the Host OS. Virtualization allows

[1]`https://jenkins.io/`

[2]`https://docs.gitlab.com/ee/ci/`

[3]`https://azure.microsoft.com/en-us/services/devops/pipelines/`

running an independent environment, which is most commonly used to run applications, which require a different OS, than the one the host machine is running. For example, a Linux VM can be run on top of a Windows machine, without having to install a separate booting option and in this VM, applications, that require Linux can be installed. In Figure 10, we can see the architecture of a VM. As already discussed, the Host OS is responsible for the Hardware resources. One level above the Host OS, we can find a Hypervisor - a program responsible for creating and running the virtual machines [61]. On top of the Hypervisor can be run multiple Virtual Machines with separate resources. Each separate VM can hold different Operating System, bins and also run multiple processes and apps. There are obvious benefits from the implementation of VMs - they are easy to setup, run and maintain. Furthermore, as they are running on a separate kernel than the host OS, VMs are known for for addressing security concerns through isolation [62]. However, multiple drawbacks can be mentioned as well. VM instances use isolated large files on their host to store their entire file system and run typically a single, large process on the host [62]. "It needs full guest OS images for each VM in addition to the binaries and libraries necessary for the applications, i.e., a space concern that translates into RAM and disk storage requirements and is slow on startup" [62]. It becomes evident, that in order to support a large number of VM environments, we are going to need a huge amount of hardware resources - storage space for the images, RAM for the memory and also virtual memory for the visualizations. Therefore, it is not feasible to have a separate VM for each of the services in a microservices architecture, which we would like to deploy.

This issue is addressed with a concept, which has gathered pace in the last few years - the containerization. "Instead of virtualizing the hardware stack as with the virtual machines approach, containers virtualize at the operating system level, with multiple containers running atop the OS kernel directly." [60] By having the containers running directly on top of the Host OS, a lot of resources are spared. They use much less storage space, memory and can be started in much shorter time. Each container holds packaged applications and parts of applications. Furthermore, binary files and libraries can also be packaged into the container to serve the dependencies of an application, if necessary [62]. Let's imagine, that we want to run a specific application, that needs MySQL database and the scikit-learn Python library for machine learning [4]. We put the application image, along with the images of scikit-learn and MySQL and after building the container, the application should run smoothly. Containerization is particularly efficient, when we need to deploy

[4]`https://scikit-learn.org/stable/`

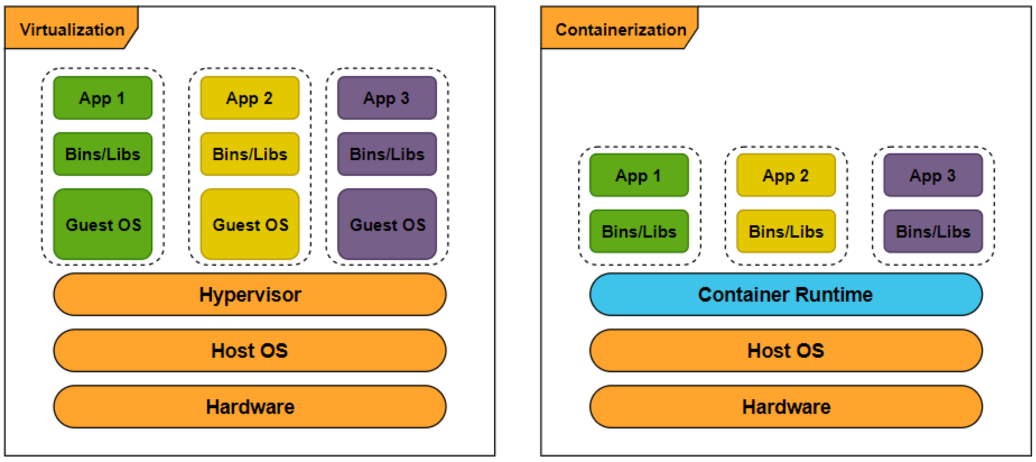

Figure 10: Comparison between Virtualization and Containerization

Source: Adapted from [60]

multiple applications with their specific dependencies and need to make sure, that they will run properly on every machine. That is indeed the biggest benefit of containerization - if an application is running in a container on one machine, and this container is copied to another machine, the application will run in the same manner [60]. This was a huge obstacle in the past, as the deployment of applications took a lot of time to tune the target environment to the one, where the application has been developed. The most widely-used tool, which helps in the entire containerization process - from packaging an application with its dependencies into container, to running and maintaining the deployed containers is Docker [5] [63]. On Figure 10, we can see the architecture of the containers. On the bottom again is the Hardware, which is being managed by the Host OS. In contrast to the VMs, however, on top of the OS, we have a Container Runtime. It is responsible to create and run the different containers. The most notable difference with VMs comes above the Container Runtime - the Guest OS part is missing. Again, we are able to run multiple containers simultaneously, but they are a much more lightweight version of the VMs.

Nevertheless, besides all the benefits, containers come with some drawbacks as well. As they share the same kernel as the Host OS, there exist security issues. The container might get infected with a virus and then the Host OS is vulnerable and can be attacked as well. This would not be possible with the VMs, as the VM does not share the kernel with the Host

[5] `https://www.docker.com/`

OS [64]. Furthermore, it is harder to work with containers as compared to VMs, as VMs provide GUI and represent a stand-alone OS, to which the users are used to. In contrast, the containerization tools (e.g. Docker) are command-line tools and one will need some time to get used to them. However, when the security issues have been addressed and the persons using the containerization tool have been trained, it becomes a far better solution for the microservices architecture than VMs. It is now possible to place each service in a separate container, including all the dependencies, that it will require, using a fraction of the space and memory of a similar VM solution. This, in the context of microservices, is quite helpful, as we aim to achieve high level of autonomy for each of the services. Furthermore, the deployment process is eased with the introduction of containerization, since the DevOps team doesn't need to worry about dependencies and discrepancies between environments.

2.6 Cloud Computing

Now after we have covered the concepts of CI/CD and containerization, which enable the microservices, we will now provide information regarding another trending technology, that is benefitting from this architecture pattern. The Cloud Computing has gathered pace during the recent years and has become a preferred way for many organizations for their software solutions.

"Cloud computing is the delivery of computing services — including servers, storage, databases, networking, software, analytics, and intelligence over the Internet" [65]. "Information and programs are hosted by outside parties and reside on a global network of secure data centers instead of on the user's hard drive. This frees up processing power and allows secure mobile access regardless of where the user is or what device is being used" [66]. The cost for the service is calculated based on the usage of the resources, with no extra costs - you pay only for what you have used. There are multiple beneficial aspects, that are of interest for the users of the cloud services: cost cutting, speed, reliability and security [65]. By using the cloud, companies no longer have to spend large amounts of money for servers, data centers and software infrastructure in general, instead they can make use of the provided one and pay based on their usage. Furthermore, the cloud solutions are hosted on state of the art servers, that are able to process huge amounts of information in short amount of time. The only requirement for using the cloud efficiently is the good internet connection, as all the information is passed through it.

We can identify three different types of cloud computing [67]:

- **Infrastructure as a service (IaaS)** is the most basic category of services offered on the cloud [65]. The thing that is offered at this level is only the infrastructure - servers, virtual machines, storage and etc.
- **Platform as a service (PaaS):** This type goes beyond the idea to provide only the infrastructure and instead offers an entire platform, which can be used for software development. It includes libraries, servers and development tools on top of the already offered infrastructure from IaaS [66]. Users can benefit from these services by using a configurable environment, which provides everything needed in order to start developing, without needing to invest time to set up the infrastructure from scratch.
- **Software as a service (SaaS)** is the most widely-used type of cloud computing. It delivers complete applications, ready to be used by the end user. The applications are maintained by the provider and the payment scheme here is typically subscription based [65].

"The Cloud computing dates back to the 1950s, and over the years, it has evolved through many phases" [68]. This statement from IBM shows us, that the cloud computing is not a new idea and has been around for quite a while. However the usage of it has boomed during the recent years, which is correlated with the increased adoption of the microservices in the companies. The main issue with cloud services offered using monolithic architecture was the scaling. Scaling monolithic applications is a challenge because they commonly offer a lot of services, some of them more popular than others. If popular services need to be scaled because they are highly demanded, the whole set of services also will be scaled at the same time, which will lead to non popular services consuming large amount of server resources even when they are not used [69].

In contrast, the microservices offer us the opportunity to focus our scaling efforts on specific services which we can decide. In a paper from 2015 [69], the authors are making a comparison between monolithic cloud solution and a microservices one. In short, they made two tests: one with 30 requests per minute and one with 1100 requests per minute. The monolith performed better on the test with fewer requests, but was outperformed on the other test. This serves to prove, that microservices enable the cloud computing to be offered to greater amount of users without significant delays.

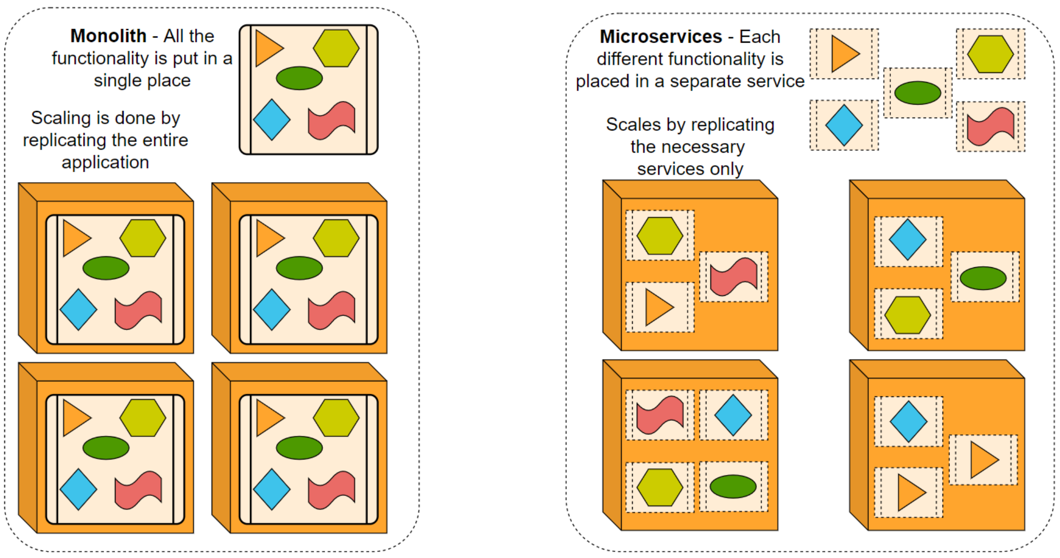
Monolith - All the functionality is put in a single place
Scaling is done by replicating the entire application
Microservices - Each different functionality is placed in a separate service
Scales by replicating the necessary services only

Figure 11: Comparison Between Monolith and Microservices

Source: Adapted from [5]

2.7 Differences and Similarities

Until now we have covered the main aspects of the monolithic and the microservices architectures. In this subsection we will put them side by side and compare them, by mentioning the aspects, that are similar between them and those that differ. This and the following subsection aim to serve as a generalization, assuming the topics discussed have already been understood by the reader. There aren't many points, which are similar for both of the architectures, but nevertheless we will name some of the most obvious.

The both approaches are used in software development to create software systems. Both of them provide the required set of instruments to create user interface, business logic and have database access. Furthermore, if the monolithic code is written following the module monolith pattern [15], both of the approaches will have well-structured codebases with high cohesion and low coupling. However, as mentioned, these are pretty generic similarities, which come primarily from the domain that the both approaches are sharing. The differences start to show, when we start looking at the details and when we look deeper, more and more contrasting characteristics start to reveal.

We will start by discussing Figure 11. It clearly shows, the main difference between a system that is built using the monolith and respective the microservices - the scaling. We can see, that when we want to improve the

Table 1: Comparison Table between Monolith and Microservices

Monolith	**Microservices**
Gets bigger and bigger over time and becomes difficult to manage.	The size of each service will remain substantially smaller and easier to maintain. When we want to add new functionality it will be deployed on separate service.
Continuous deployment becomes difficult, as a small change to a component in the application, requires rebuilding of the entire application.	Continuous deployment is possible here, as each service can be deployed independently.
Scaling of the application requires entire application scaling(i.e.: launching multiple instances of the big monolithic application to serve the needs), even though only one small part of the application is resource intensive.	Scaling of individual component is possible here, as each different component is de-coupled to different services, and can be scaled up horizontally and individually.

scaling of a monolithic application, we replicate it on as many servers as we want. The application and its components remain the same and are scaled equally. Using this approach makes us lose resources and we are unable to focus only on those parts of the system, that need scaling. In contrast, the microservices allows us to have each functionality, that we want deployed into a separate service. When we want to scale the system, we distribute the services across the different servers, by replicating only those, that we need [5]. Furthermore, the organization of the microservices tiers differs from the monolith - see Figures 2 and 8. Of course, there are countless other points, that can be mentioned here, but we wanted to make a general differentiation between both architectures and therefore won't describe each of them. Nevertheless, we would like to include a comparison table, found in an article written in 2015 [70]. The author uses as an example a company that offers the service of transcoding videos from one format to another. It is a CPU-intensive process and using the microservices, we can scale only this part of the entire process, without needing to replicate everything else accross a number of servers (i.e. login system, payment system and etc.). The most important differences the author outlined can be found in Table 1 [70].

2.8 Advantages and Drawbacks of both approaches

So far we have mentioned briefly the advantages and the drawbacks, that the solutions bring along with them. However in this subsection we will expand the field. Furthermore we will give example situations and will discuss which approach will be more appropriate and beneficial for the specific case. There is no single best option to choose. Everything depends on the context and the use-case. Therefore before deciding on which one to choose from, we should ask ourselves and look at the benefits that we could take from either of them. We will provide two tables, Table 2 and 3, which have been put together, using information from various sites [35, 34, 71]:

Table 2: Advantages and Drawbacks of the Monolith.

Monolith	
Advantages	**Drawbacks**
Simple to develop. Developers are used to this approach and in the beginning it is easier to use it.	Maintenance issues - the codebase grows with time and therefore becomes harder to maintain
Simple to deploy - we can transfer the application package to a server and it is ready to use, without further setup required.	The entire application must be redeployed on each new update
Simple to scale horizontally by running multiple copies behind a load balancer.	Scaling becomes an issue, even though it is mentioned in the advantages. As it is easy to scale the monolith, this scaling is inefficient. We are scaling the entire application, instead of just the parts that we need to.
Simple to test. End-to-end testing is possible.	Continuous deployment is difficult.
	Reliability issues - a failed update can crash the entire system.
	Requires a long-term commitment to a technology stack
End of Table	

Table 3: Advantages and Drawbacks of the Microservices.

Microservices	
Advantages	**Drawbacks**
Enables the continuous delivery and deployment of large, complex applications.	Added complexity to the application - we have to deal with a distributed system already
As each microservice is relatively small, it is easier to be understood by the developers.	Database transactions become more difficult and consistency can be an issue.
Because of the small size, the maintainability issues are fixed.	The testing of the application becomes much more complex, as we are not able to test.
Each service can be developed independently by a team that is focused only on it.	Deployment complexity - no matter how beneficial the continuous delivery might be, it adds to the complexity of the application. When updating the application, we have to watch out for multiple services and their instances, in contrast to the monolith. Each of those services must be configured and monitored.
Improved fault isolation - If a service fails, the others continue operating	Increased memory consumption - each service runs in a separate container and the combined resource consumption is therefore higher than with the monolith.
Each service can be scaled independently, depending on the needs.	
Eliminates any long-term commitment to a technology stack.	
End of Table	

After taking all things into consideration and having a clear idea what we want our system to do, we will be able to choose the solution that will suit our needs the best. For example, if we need to create a simple application for our company, which will be used internally by a small amount of people within it, we should go for the monolithic approach. It is easier to setup and to maintain if the codebase is small. Furthermore we wouldn't need to develop complex APIs and manage multiple databases. On the other hand, if we wanted to offer a more complicated application to the end-users, we might reconsider using microservices. If we would like to have a large and concurrent user-base, which usually demands regular updates and bug-fixes, we should go for microservices. We will have to sacrifice the simplicity of our application in order to achieve better scalability and therefore higher user satisfaction.

With this, we have wrapped up the background knowledge section and up to this point, the reader should be aware of the concepts of the monolithic software architecture and microservices. In our next section, we will look into relevant literature and papers, that have been written in relation to the main topic of this thesis - the transition from the monolith to microservices.

3 Related Work

In this section we will review a few papers and books, which are relevant to the topic we want to explore. Currently, the microservices have become a trending topic and there is a number of papers describing them, how they work, best practices and etc. However there is still not that much research made on the decomposition of the monolith architecture and the transition to microservices. Nevertheless, there are some relevant papers describing the process. We will take a look into them and will examine the most important aspects they cover. Many of these aspects will be taken under consideration when we are developing our approach. We will combine different components from the different papers and will expand on them in order to achieve our target - to decompose the monolith in the most efficient way.

The first paper we are examining is however not strictly connected to the migration process. Instead it focuses more on the reasoning, the motivations and the issues of the companies, that have decided to migrate from monolith to microservices. We decided to include this paper, because it is always beneficial to learn from the experiences of others. Furthermore it helps us to decide, if it would be worth the time and resources, by taking into account the issues that have arisen in other companies. The paper [72] was written in 2017 by Davide Talibi et. al. and presents the results of a survey the authors made regarding the mentioned points. The participants in the survey were 21 practitioners who adopted a microservices-based architectural style at least two years ago [72]. They had different roles ranging from project managers, through software architects to CEOs. The participants were interviewed with questions according the information needed. We will quote three tables with results from their survey and will provide explanatory information about each of them.

Table 4: The motivations for migration identified in the survey

Motivations	**Entire Dataset**		**Consultants**		**Others**	
	#	**Median**	#	**Median**	#	**Median**
Maintainability	15	4	5	2	10	4
Scalability	15	2	3	3	12	2
Delegation of team responsibilites	11	3	1	3	10	3
DevOps support	8	3	2	1	6	3
Follow the trend	6	4	2	3	4	4
Fault tolerance	2	4	2	4	-	-
Technology experimentation	2	3	2	3	-	-
Delegation of software responsibilites	1	4	1	4	-	-

Table 4 is adapted from [72] and shows the distribution between the different motivations for the adoption of microservices architecture. There were no significant differences between the different roles of the participants. However, the authors recognized two groups of participants - working for migration companies and all other persons interviewed. The authors used five-point scale. The answers ranged from 0 to 4, with 0 being 'irrelevant' and 4 - 'fundamental'. The table shows the frequency and the median of each aspect for each group. Here we will look into the Entire Dataset columns only, in order to understand the general importance of the different motivation drivers. After the results were calculated, the maintainability came on first place with 15 out of the 21 interviewed persons giving it at least some importance. The median importance is 4, which means, that it is considered fundamental by most of the people, who have mentioned it. Next comes the scalability, again with 15 mentions, but with lower value for importance, which is why it comes on the second place. Next come two motivations, which enable improvement in the teamwork and deployment. There is an interesting point, which comes at the fifth place - 'Because everybody does it'. This shows, that teams and companies would like to keep up with the latest technologies offered and not lag behind. The last three motivations mentioned were pointed out only by the migration consultants as important. That is because the consultants are more interested in the bigger picture, in contrast to the developers, who are looking for a way to ease their develop-

Table 5: The migration issues identified in the survey

Issues	**Entire Dataset**		**Consultants**		**Others**	
	#	**Median**	#	**Median**	#	**Median**
Monolith decoupling	7	3	-	-	7	3
Database migration	6	4	-	-	6	4
Communication between services	4	3.5	2	4	2	3
Effort Estimation	2	4	2	4	-	-
DevOps Infrastructure	2	4	-	-	2	4
Library conversion	2	4	-	-	2	4
People's mindset	2	4	1	4	1	4
Expected long-term ROI	2	3	1	3	1	3

ment and day-to-day tasks. In conclusion to this part of the survey, we could say that the most important motivations are Maintainability and Scalability.

Next, the authors asked the participants about the issues, that they most often encounter while migrating to microservices. We can see the results on Table 5, which has been adapted from [72]. The main issues reported are the complexity to decouple from the monolithic system, followed by migration and splitting of data in legacy databases and communication among services. Effort overhead was only considered by non-consultants. People's minds are another personal reason against migration, followed by concern for the lack of return on investment (ROI) in the long run [72]. We can once more see the differences between the consultants and the developers in this table. Again the consultants have reported more issues from the conceptual perspective (Effort Estimation), as opposed to the practical perspective of the developers and the other participants (Monolith decouling and database migration).

After examining the motivations and the issues, the authors were also interested into the most beneficial aspects of the implementation of microservices architecture. Table 6 shows a table, that presents their summarized findings. The maintainability and the scalability are leading the way here again. This serves to prove, that the motivations, that forced the companies to migrate to microservices turned to actual benefits after implementation. Practitioners reported that in the beginning they were aware that such a com-

Table 6: The benefits from the migration identified in the survey

Benefits	**Entire Dataset**		**Consultants**		**Others**	
	#	**Median**	#	**Median**	#	**Median**
Maintainability improvement	9	4	5	4	4	3.5
Scalability improvement	7	2	5	2	2	4
ROI	6	4	-	-	6	4
Reduced architectural complexity	6	3	-	-	6	3
Performance improvement	2	1	2	1	-	-
Testability	2	3		-	2	3
Separation of concerns	2	3	2	3	-	-
Suitability for Scrum	2	3	-	-	2	3
System understandability	1	4	1	4	-	-

plex architecture might have a negative impact on software maintenance, but in the long run (at least one year, based on participant answers), they discovered that the architecture is less complex to understand and the system requires less maintenance [72]. Furthermore, the ROI is improved, because in the long run the microservices-based systems are less expensive than the monoliths. More notable benefits mentioned are the testability, performance improvements and the reduced architectural complexity. Less appreciated, but still important benefit is that microservices are more suitable for Scrum and have improved system understandability [72].

The authors also interviewed the participants about the migration process. By summarizing the responses, three different migration patterns were identified. The aim of the first two patterns is to migrate an existing monolithic system to a microservices-based one by reimplementing the system from scratch. "The aim of the third pattern is to implement only new features as microservices, to replace external services provided by third parties or develop features that need important changes and therefore can be considered as new features, thus gradually eliminating the existing system" [72]. The differences between Process 1 and Process 2 is the prioritization principle. When companies are using pattern 1, the prioritization is only based on the customer value. The higher the customer value, the higher the development priority [72]. In contrast, when using pattern number 2, higher priority is given to the existing services with more bugs. Out of these problematic services, the ones with fewer dependencies are selected first, in order to reduce the impact on the system and to deliver the new implementation faster [72].

By reviewing this paper, we can conclude, that it gives us a good foundation for the process of the transition from the monolith to microservices. Furthermore the authors included multiple perspectives and opinions. The issues and benefits reported during the transition process may help us identify, whether we need to migrate at all, or if we do, which of the mentioned approaches may be most appropriate for us. However it is important to note, that there results may be biased due to the small sample of participants in the survey.

The next paper we would like to look into is "Microservices migration patterns" by A. Balalaie et al. [6]. The main concept of the paper is that every enterprise must choose their own approach for migration based on multiple factors. There is no single approach and pattern, that is working for every company and Situational Method Engineering (SME) is needed [6]. The authors propose to populate a pattern repository with the reusable process patterns and methodological steps in order to extract the program logic and distribute the system in blocks of interdependent processes, which will become the microservices [6]. After having this pattern repository, the migration engineers will be able to select appropriate patterns and complete the transition in the easiest way according to their needs and preferences.

The paper starts with techniques for identifying patterns in the company and building the aforementioned pattern repository. In order to effectively extract the patterns and the reusable processes, the authors proposed a pattern-extraction metamodel [6]. These patterns can be extracted from every company and can be used specifically for the its needs. In this thesis, however, we won't give too much attention to this part, but will instead focus on the patterns, that the authors identified. For each pattern, we can find the context, problem, solution, reuse situation and intention. Furthermore the authors give give examples for technologies, that can be applied for maximum optimization. Here are the fifteen patterns they recognized:

1. **Enable continuous integration** - In order to enable the microservices architecture one of the first prerequisites is to have continuous integration. This will help us to deploy our changes faster and in a more efficient manner. The authors suggest to put each new service in a separate repository, in order to have better transparency and to keep track of each service's changes easier. Afterwards, for each service a

CI job must be prepared, including automated tests. With each new commit to the repository, the job is triggered [6].

2. **Recover the current architecture** - The entire system must be examined and the big picture should be captured. All the dependencies and the connections between the components should be explored.
3. **Decompose the monolith** - Decompose the monolithic architecture using the DDD principles [6, 29]. This is the biggest challenge and the longest pattern to implement.
4. **Decompose the monolith based on data ownership** - Decompose the monolithic architecture, based on the data relations. The idea is to find highly cohesive groups of data entities and based on that, separate services from the monolith [6].
5. **Change code dependency to service call** - All the code-level functions, which invoke methods from external libraries must be reimplemented with service calls [6].
6. **Introduce service registry** - Create a service registry, which will act as a database of service instances' addresses. Each service registers itself upon initiation and then sends periodic heartbeat [6].
7. **Introduce service registry client** - In order to be able to register itself, each service must have a service registry client implemented. It provides the functionality to register, send heartbeat and delete itself from the registry [6].
8. **Introduce internal load balancer** - Each service must know the locations of the other services' instances. For that purpose, we use the internal load balancer, which keeps track of these instances and redirects the requests to the most appropriate one [6].
9. **Introduce external load balancer** - This pattern is similar as the previous one, with the difference, that it is not located in each separate service, but instead is a separate component. Its function remains the same as the internal one [6].
10. **Introduce circuit breaker** - When a the services communicate and a user request spans two or more service, we must include fault tolerance in order to reduce the risk of system failure [6].
11. **Introduce configuration server** - A separate repository is created, which stores the configurations for each separate service. The configuration server is responsible for the synchronization between the repositories, when some change occurs in the configuration options [6].

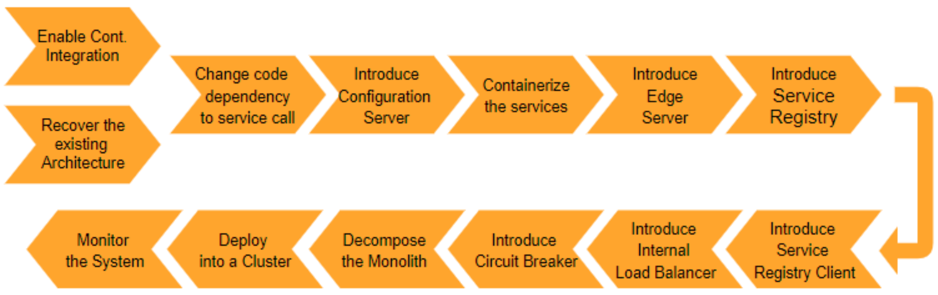
Enable Cont. Integration
Recover the existing Architecture
Change code dependency to service call
Introduce Configuration Server
Containerize the services
Introduce Edge Server
Introduce Service Registry
Introduce Service Registry Client
Introduce Internal Load Balancer
Introduce Circuit Breaker
Decompose the Monolith
Deploy into a Cluster
Monitor the System

Figure 12: The order of the pattern application proposed by the authors

Source: Adapted from [73]

12. **Introduce edge server** - This is actually the API Gateway microservices pattern, which we discussed earlier [43, 17]. The role of the edge server is to create another layer between the system and the user and enable the dynamic routing capabilities of the system [6].
13. **Containerize the services** - In order to make sure, that the environments are the same in production and development environments, we need to use a container for each new service [6].
14. **Deploy into a cluster and orchestrate containers** - After all the services have been containerized, this pattern addresses the scaling of the system. Depending on the current load, the orchestration engine creates new containerized service instances, or removes them [6].
15. **Monitor the system and provide feedback** - After the migration process has been completed, the monitoring phase begins. Multiple tools can be used to track the performance and give the developers information, which can later be used for improvements [6].

As we can see some of the patterns, that were identified are eliminating each other (e.g., Patterns 3 & 4). Therefore, when we want to decompose a monolithic application to microservices, we have to make a domain analysis of our system and choose what will suit us the best in order to achieve higher efficiency based on our current system structure. Of course each of those patterns has it's own context, problems and use case, but nevertheless, the overall idea of the authors is to have a structured guide, which will help to decompose the monolith efficiently, keep it up to date and fault-free.

The paper continues with an actual implementation proposal of the mentioned patterns. On Figure 12 is depicted the recommended order by the authors, in which the different patterns should be implemented. They start with the Enabling of the Continuous integration in parallel with the recovery of the current architecture. These two steps are actually the easiest ones to implement and can be done in parallel, because they don't interfere with each other. After the recovery is done and all the code is adjusted to work with service call architecture, the process continues with implementation of the new components. All of the following steps after the decomposition introduce new technologies, which are necessary in order to have fully functional and properly working application following the microservices architecture standards. The proposal continues with the introduction of configuration server and containerization of the services. Up to this point, the selected steps did not affect the end users. However, subsequent steps would affect them, and an isolation mechanism was needed to make the following changes transparent to them. This could be achieved by using an edge server [6]. Then the process continues with introduction of service registry and client, load balancer, circuit breaker. Afterwards comes the most difficult and critical step of the process - the decomposition. At this point we have introduced fault tolerance and enabled communication between services, therefore, it is assumed safe to start the decomposition of the codebase. After this step has finished successfully, the process can continue with a deployment into a cluster. At this point, the entire system should be operating, with all the desired functionality. The process finishes with the monitoring phase, which helps observe the system, fix its issues and improve it.

This paper [6] is very helpful, because it provides general guidelines, which, if used correctly can lead to a proper decomposition of the monolith and the migration to microservices architecture. However, it is necessary to say, that we have to pick and choose among these patterns and are also free to choose the steps, that may suit our company and needs the best.

Another paper worth mentioning is 'Migrating to Cloud-Native Architectures Using Microservices: An Experience Report' [73], written again by the same authors as [6]. As the title suggests, this is a report about the authors' experience in a company to migrate an on-premise monolithic application named SSaaS(Server Side as a Service) to microservices architecture. In contrast to the papers discussed previously, this one is more specific and describes the migration steps required specifically for the application they were working on. The approach presented in this paper is pretty much the same as in the one, that was discussed previously. As noted, however, the entire approach is really specific and is concerned with the certain system and its decomposition. For example the technology stack is provided and justifications are given, why exactly these technologies are used. One of them is

Ribbon load balancer [6], which is chosen, because it works with the Spring framework [7], which was used initially in this project. However, we can look into the bigger picture and generalize the approach, which could prove useful in a similar scenario. We could change the recommended technologies from this paper with ones, that are more suitable for us and apply the general methodology instead.

They start by investigating the current system structure and gathering the requirements. Next follows the refactoring of the current data. Their choice was to go with the Domain Driven Design [29] decomposition approach [39, 17] in order to break down the monolith. After the authors examined the current situation and decided to migrate to microservices, they described the migration steps in a separate chapter. The process started by enabling the continuous integration pipeline. This allows faster and more effective updates of the growing number of services. They suggested containerization with Docker [8] for this phase of the process and preparing a pipeline in Jenkins [9]. After the CI was prepared, the transformation of one of the libraries to service was performed. This was done by switching from method calls to service calls in the form of REST API [73, 74]. After having working containers and CI pipeline, they introduced Continuous Delivery. Next followed the Edge Server (API Gateway), the Service Discovery, Load Balancer and the Circuit Breaker. With this step, the system now had some fault tolerance and it was easier for the developers to work on it. As a next step, the authors proposed the introduction of a resource manager, which would help satisfying the different service requirements. The process continued by introducing new services, which were formerly part of the monolith. The final step the authors proposed was the deployment of the services on a cluster.

We can see, that even though the authors of this paper [73] and the one discussed before [6] are the same, the migration steps, that were proposed are slightly different. For example, they used a resource manager here, in contrast to the Figure 12 proposal. Furthermore, the decomposition of the monolith in the first paper is following the Data Ownership pattern and in this one - following the DDD decomposition pattern. The paper, that was outlining the patterns gave us the general idea and guidelines. In contrast, here we see the application of these patterns in practice. This serves to prove, that we are free to choose our own approach and select the most appropriate patterns to achieve our goals.

[6]`https://github.com/Netflix/ribbon/wiki`
[7]`https://spring.io/`
[8]`https://www.docker.com/`
[9]`https://jenkins.io/`

Table 7: Microservices Patterns - Advantages and Drawbacks

	Pattern	**Advantages**	**Drawbacks**
Coordination	General	Increased maintainability Can use different languages Flexibility Reusability Physical isolation	Development complexity Implementation effort Network-related issues
	API Gateway	Extension easiness Backward compatibility Market-centric Architecture	Potential bottleneck Development complexity Scalability
	Service Registry	Increased Maintainability Communication Software Understandability Failure Safety Understandability	Interface Design Flaws Service Registry Complexity Reusability Distributed System Complexity
Deployment	Multiple Service per host	Scalability Performance	
	Single Service per host	Service isolation	Scalability Performance
Data Storage	DB per service	Scalability Independent Development Security mechanism	Data needs to be split Data consistency
	DB per cluster	Scalability Implementation easiness	Increased Complexity Failure Risks
	Shared DB server	Migration easiness Data consistency	No data isolation Scalability

Next, we will turn our attention to another paper, which again focuses on the microservices architectural patterns [75]. However, this time the goal of it is to present a systematic mapping study. The authors of this paper are the same as the first one we discussed in this chapter [72]. The aim of this study was to create a microservices pattern catalogue by analyzing relevant papers. The study started with pattern identification process, which comprised of literature review, based on specific search patterns (i.e. titles including microservice* and etc.). The query returned 2754 unique papers, but after eliminating a large part of them, the final selection resulted in 42 accepted papers, published up to the end of 2016 [75]. We won't provide information on how the study was conducted, but will instead focus on the final results. Table 7 is adapted from [75] and presents the most commonly-occurring patterns in the papers, found relevant in the study. There we can find the most frequently mentioned general advantages and drawbacks of the microservices. Furthermore the different patterns are divided into groups and for each of them are given pros and cons, that come along with the implementation of it. The table gives us a general idea regarding the most frequently adopted patterns and could be used as a good starting point to compare the different approaches (i.e. the database storage patterns, or deployment patterns).

Another strong source for information regarding this topic is the book Building Microservices by Sam Newman [28]. The book contains thorough examination of the microservices and by reading it, one can get a solid understanding of this architecture pattern. In this thesis, however, we are going to focus only on the chapter regarding the decomposition of the monolith, as this is our main research topic. The author suggests to start the decomposition by finding isolated portions of code. In order to describe such portions, he uses the term seams, which is borrowed from the book "Working Effectively with Legacy Code" by Michael Feathers [76]. By identifying these seams in the monolithic application, we would have a foundation to define the service boundaries. In order to describe the decomposition process, the author used as an example a monolithic application from the musical domain. The approach begins by dealing with the database decomposition and provides very good suggestions on how to break it up. A very useful tool, that is proposed is SchemaSpy [10]. It reproduces the database schema and helps to group the different tables into groups. Regarding the database decomposition, it is suggested to separate the database and use the database per service pattern [30, 17]. This will increase the database calls and will, therefore, result in decrease in performance. That is where, the author states, that we should decide if we can tolerate slower transactions. If we are not able to - we can use different database pattern to achieve the decomposition.

Particularly helpful is the idea on how to break up the foreign key relationships. It is suggested to remove them altogether. Instead they can be replaced with constraints in the services, instead on database level. Another important aspect in the decomposition, which is addressed in this book is the decomposition of shared tables, which are being populated by two different domains in the system. After decomposing, these domains will turn into separate services, but they will still share the same database table for CRUD operations. The proposed solution is to remove the existing table and create one for each of the services. This will lead to data redundancy, but the data

[10]`http://schemaspy.sourceforge.net/`

will be consistent and easy to access without the need of too many API calls.

Furthermore, in the book [28] we can find a way to address a really interesting issue - the reporting. Usually reporting uses data from large parts of the system in order to process it and produce the required report. However by using microservices, this will produce an untolerable amount of API calls and generating a report will take a lot of time. To tackle the problem, first a reporting service should be created, with its own database. It is proposed to create Data Pumps to periodically push data from each service to the reporting database. This will reduce the calls when creating the report, by spreading them to longer periods [28].

The decomposition chapter in the book [28] is mainly concerned with database refactoring. The author provides thorough information on how to split the database and presents multiple useful cases, along with proposed solutions. Aside from the decomposition chapter, the entire book is worth reading and contains valuable information regarding the microservices.

In a paper from 2019 [77], the authors provide two different ideas in addition to the bounded context [29] decomposition of the monolith, which we discussed above already. The first one is to use transformers to migrate a legacy system. It is suggested, that the entire application be split into blocks, which are sorted into into a predetermined order when the blocks will be available in the new target environment [77]. Each of the blocks was either developed from scratch and releasing it as a microservice, or by transforming the existing code using a transformer solution. "Utilizing a transformer enables an automatic migration of legacy code such as Cobol or PL/I to Java on the target platform without impacting on its existing functionality, while performing integrated automated tests" [77]. This is a mixed approach and is trying to keep as much as possible the legacy code and make it work with the microservices architecture.

The other theory, that can be found in this paper is to use a low-code development platform [78] instead of decomposing the monolith. The authors' idea is to take into account all the challenges and issues, that may arise when migrating legacy code. If these issues prove to be too much, we can use the mentioned low-code platforms to scale and introduce new features to our organization. By utilizing such a platform, there is no need to transform the existing code, instead the platform serves as a middleware between the legacy monolithic codebase of our organization and the scalable microservices, on which the low-code platform is hosted. This entire idea seems interesting if we would want to focus the scaling only on some parts of our system and are not willing to risk with downtime and bugs. However for a long-term solution, a proper migration to microservices will prove to be the best approach.

The final paper we would like to review in this chapter is ”Challenges When Moving from Monolith to Microservice Architecture” from 2018 [7]. As the title suggests, it is focused on the challenges that may arise when we are migrating a software system to microservices. The authors have separated them into two groups - technical and organizational challenges. In order to summarize the information, we are going to put them into a list and quote some of the most important points for each of the groups.

- **Technical Challenges:**
  - **Time-related concerns** - the decomposition and migration to microservices may take a lot of time. The solution proposed by the authors is to do it in small chunks.
  - **Test coverage** - before starting the process, sufficient test coverage should be created. The testing should be automated instead of manual, as in such a way large parts of the system can be tested in short amounts of time.
  - **Finding the seams** - arguably the most difficult part of the decomposition - to find the boundaries between the services. The authors suggest to start the decomposition from the most obvious services and then move on with the more complicated ones.
  - **Microservices sizes** - if we focus too much into the decomposition, this will result in very small fine-grained services. However this may prove to have bad consequences in the future, as too many inter-service calls will be made in order to complete a simple operation.
  - **Integration between services** - it is suggested, that the interface be made simple and with backward compatibility. This means, that when new functionality is added, the client doesn’t need to be updated.
  - **Scaling and Service discovery** - when we are using microservices, there will be many different services running in parallel. Furthermore, each service can have multiple instances, depending on the usage of the service. We should think of a way to manage the resources and the authors address this issue by suggesting to use Kubernetes [11].
  - **Disaster Recovery** - eventually a service will fail and we need to have a backup plan. The authors suggest to setup a good logging of the microservices and put all the logs together in one place. This way we should follow only this single, aggregated log and check if there

[11]`https://kubernetes.io/`

is a problem with a specific service. The idea is to anticipate the problem and fix it as soon as possible.

  - **Disaster Recovery (cont.)** - if a service is unavailable, or has slow response time, this could hinder the entire system. Therefore, the authors suggest introducing a circuit breaker, which "monitors for failures and when there are enough failures the subsequent calls to the dependency won't be made and instead an error is returned" [7].
  - **Database issues** - managing multiple different databases may prove hard. We need to take care of distributed transactions and database consistency.

- **Organizational Challenges:**
  - **Organizational Structure** - The authors discuss the Conway's law [79]. It states, that "the organization which is designing a system will produce a system which structure is a copy of the organization's structure" [7, 79]. The authors state, that "if the structure of the organization is monolithic then microservices approach does not work. The organization must split these big teams to smaller teams which can work autonomously" [7].
  - **DevOps mentality** - it is proposed, that when microservices architecture style is adopted by an organization, the DevOps approach should be used as well. This way, the company will be able to reap greater benefits from the microservices.

After reviewing all these papers, we feel more confident and are able to start putting together our approach. We are going to use the information from the papers, synthesize it and use it as a foundation to build upon it. Our approach will be focused on filling some gaps, that can currently be found in these scientific works. In contrast to most of these papers, we are going to propose a structured step-by-step process, which will aim to decompose the monolith into microservices. The aim of this thesis will be to present a general solution by using a concrete example with an ERP system. In the next chapter we will present our migration proposal, along with justifications for each migration step. Afterwards, in the discussion chapter we will outline the key differences and the gaps, that we managed to fill with the proposed process.

4 Research Method

After providing all the relevant information, that is needed to know in order to understand the purpose of this thesis, in this chapter we will present our ideas on how to decompose the monolith. We will try to present the information in the most non-technical way possible and will put our focus on the general concepts. In order to strengthen our proposal, we will use an open-source ERP system as an example, which will help the reader to better comprehend the concepts. Nevertheless, our aim is to enable this migration proposal to be applied in any domain. The chapter will start with a section regarding the ERP system of interest - Odoo[12], introducing the relevant functionalities for this thesis. We will then start with the approach section, which will be organized in the following manner: First, a grouping of the different parts of the monolith will be performed, based on domain analysis. Next will follow the database decomposition process, based on these groupings. We will move on with the application of various microservices patterns, including their implementation order. For each of the steps of the proposed process, first a general solution will be outlined and then we will provide an example with Odoo. Furthermore, we will provide information regarding useful technologies, that can be used from the beginning until the end. The chapter will conclude with a section concerned with the maintenance and CI/CD after the migration has been performed successfully.

4.1 Introduction to Odoo

"Enterprise Report Planning (ERP) systems are highly complex information systems" [80]. They aim to reduce the complexity of the operations within an enterprise, at the cost of an increased system complexity. Nowadays, to have an efficient and well-working ERP system is one of the critical success factors for a company. By having it, we are capable to store everything in one place and monitor the performance of the enterprise. One of ERP's main responsibilities is to automate as much as possible of the daily processes. However, each company usually has company-specific processes and, therefore, needs customization of the ERP system. Furthermore, for example, different countries have different accounting standards and regulatory requirements, which our company must adhere to when producing our reports. This shows, that there cannot be a single universal ERP system, which can be deployed in every enterprise and work efficiently. By customizing the ERP, we tune it to our company's needs and are able to reproduce the system based on our

[12]`https://www.odoo.com/`

needs, instead of changing our processes, because of the lack of specific system capabilities.

There are many ERP systems available, which offer the required functionality. Some of the most widely-used ones are provided by software giants SAP [13], Oracle [14] and Microsoft [15]. These ERP systems are used in most of the big corporations and large enterprises. And there is a reason for that - they are reliable, receive regular updates and have solid customer support and active community. However, they have a few downsides - increased complexity, bad user interface and most importantly - usage costs [81]. Depending on the size of the company and it's needs, it may not be necessary to have these complex systems. Instead, if we are running a small to medium enterprise, we could use a free to use open-source ERP system, which will be enough for our needs [82]. Such a system is Odoo (formerly OpenERP) [83].

"Odoo is a powerful open source platform for business applications. On top of it a suite of closely integrated applications was built, covering all business areas from CRM and Sales to Accounting and Stocks" [84]. As the platform is open source, this means, that we are capable of expanding it and tuning it to our own needs. There is no need for expensive consultants and external developers, instead we can setup a small team, which will be able to efficiently implement new functionality, based on the needs of the company. Odoo provides SaaS solution as well, but we are not focused on that, instead we will turn our attention on the way the ERP system operates.

Odoo has fairly simple architecture and uses well-known software languages to implement it. In the following lines, we will look at the way Odoo operates and its components. We won't provide specific details, but in order to understand the later examples in this chapter, there are some knowledge prerequisites, which must be cleared. There are three main components in the Odoo ERP architecture [85]:

- **PostgreSQL Database Server** - stores the data and the ERP configuration
- **Application Server** - it is written in Python and contains all the business logic.
- **Client** - the only option to connect to Odoo is with a web client. The users can connect via standard web browsers [86].

Let's now briefly take a look at the Odoo architecture on Figure 13 to further explain the different components. As we can see, at the bottom is

[13]`https://www.sap.com/uk/index.html`

[14]`https://www.oracle.com/applications/erp/`

[15]`https://dynamics.microsoft.com/en-us/erp/what-is-erp/`

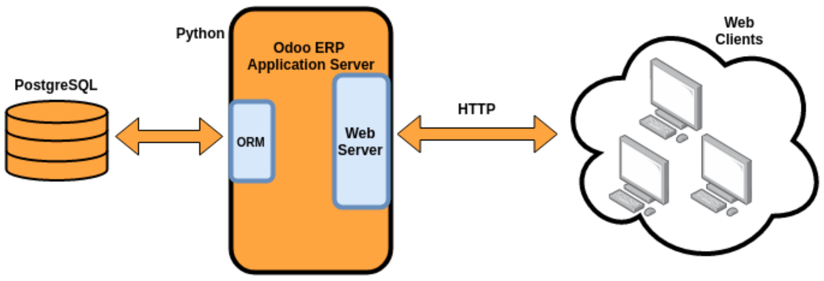

Figure 13: Odoo Architecture

Source: Adapted from [86]

the PostgreSQL database server, which essentially represents the data tier of Odoo. PostgreSQL is a open-source relational database. The connection from this layer to the application is done with the help of Object Relational Mapping (ORM) [87, 88]. In short, the ORM helps us execute SQL queries, by using different programming languages. Odoo provides convenient support and the mapping provided is fairly simple and easy to understand. For example, the creation of a new database table is done by creating a new Python class. The ORM resides in the Application Server layer and is the bridge between data and application logic. On top of all the enterprise logic, that is contained in the Application Server, it further provides complete development framework [85] to write business applications that work with the server. That is exactly the most significant aspect of the system and the main reason, why the companies choose Odoo. Another component, that is integrated into the Application server tier, is the web server. It handles the HTTP queries and works, based on Web Server Gateway Interface(WSGI) standards. The final part of the architecture, which we see on Figure 13 is the clients sector. The client is working as a JavaScript application. It is relatively simple, as all the operations are done server-side and is used mainly as an interface between the user and the ERP system.

The architecture design of Odoo is virtually a classical three-tiered monolithic architecture, with it's own specifics. However, it is not the interesting part of the ERP System. Instead, it is the way the different software components are represented, their interactions and dependencies. "The model is essentially the data that makes up your Odoo installation, which is stored in the PostgreSQL database. Odoo is unique, in that, database structures are typically defined by the Odoo modules at the time they are installed. The Odoo framework takes the model definitions and automatically creates the necessary table structures inside the PostgreSQL database" [89].

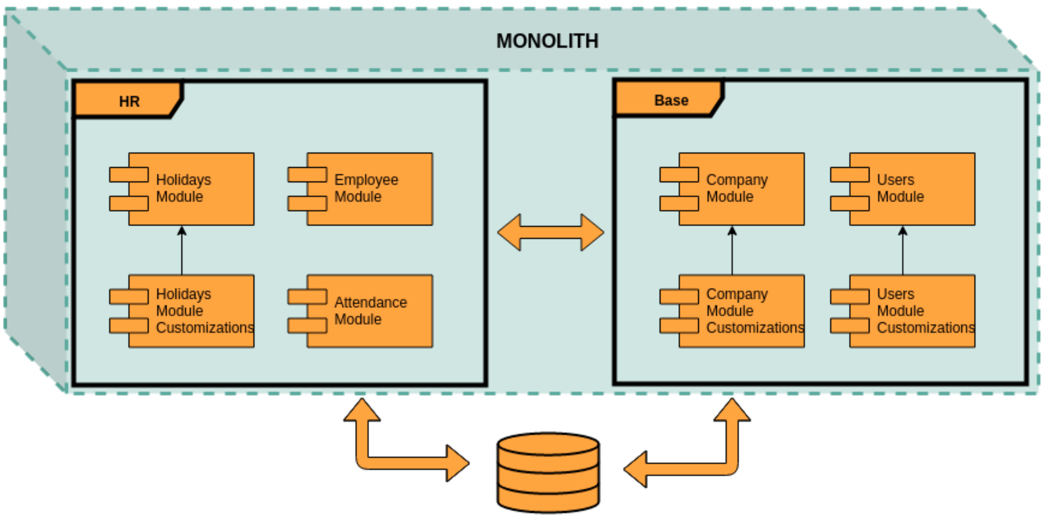

Figure 14: Odoo Module Relationsips Example

Each of the modules is simply a new, additional functionality to the base system. Odoo comes with a large number of them by default - Sales, Invoicing, Warehousing and etc. In addition to them, there are even more modules developed by the community and released online [90]. However, for each organization, it is the company-specific modules, that have the biggest business value. Figure 14 depicts an example of different modules within Odoo. The entire relationship structure of an ERP system is quite complex, therefore, we have selected just a simple example from a part of it to explain here. We can see, that the modules are logically separated in different domains. This separation is needed by the developers in order to see the boundaries of the different aspects within the system and more easily implement new features. We can see, that inside each domain, multiple modules exist, which represent the various subdomains. These modules can interact between each other within their domain and also with other modules across the domains. However, we are assuming, that most of their communication will be inside their specific domain, as the modules are more coupled within it. On Figure 14 is displayed the HR domain and the basic modules, that are installed by Odoo. Each employee has their profile defined in the employee module. They are able to enter leave requests in the Holidays module and enter their attendance times in the relevant module. However, in order to enter these attendances and leave requests, the employess will need to have an account to log in the system. The user accounts are created in the other domain of the system - the Base. After creation, it is linked to the specific employee and now all of the mentioned operations can be performed without the need to refer to the Base model again. This is a good example of the domain boundaries, that exist within the ERP system. Many more requests will be made within the domain in which a module resides, as compared to other domains.

Another aspect, that draws our attention on Figure 14, are the customization modules. As we can see, they inherit the main module, that holds the default logic and extend it with new functionality. It is possible to add functions, override existing ones, or completely remove some of them. Practically, there is also the option to implement the new features by simply extending the default module, but the recommended way to enhance the system is by using inheritance and creating new module [91]. This is an important feature of Odoo, which will come in handy, when we start with our migration proposal. There, we will use the different modules in order to suggest a grouping approach, which will then be used as the base for the decomposition. In order to keep the logical flow, we are going to use only one part of the system in our migration - the Project domain. The reason for that is, that it has the appropriate complexity and is of the perfect size to demonstrate the concepts of the approach.

4.2 Approach

In this section, we will present our solution for the migration from monolith to microservices. It will be organized in logical steps and each step will start with general theoretical approach. After the ideas have been exhibited, we will narrow them down and display a simple practical example with a specifically chosen part from Odoo ERP system - the Project Management domain. The reason for the selection of this specific part, is that it is not too complex and in the same time provides the required functionality, which will help us justify our beliefs. At the end of each step, there will be a technology stack, focused on the Python programming language, because this is the main language Odoo uses to operate. The process of migration is time consuming and many obstacles can appear while transitioning. Therefore , before everything else, we should arm ourselves with patience and be ready to meet the challenges, that we are about to face.

4.2.1 Phase 1 of the Migration

Theoretical Part

Before starting the decomposition of the monolith, it is important to first ask ourselves, if we really need microservices, rather than follow the trend for no reason and migrate, without the need to. In order to do that, we should assess the current software system, that our company has and see, if there are actually any flaws in it. If such flaws exist, we should structure them and check, whether they can be fixed with the introduction of the microservices architecture. For example, a list of flaws and desired features can be made and it can be compared against one of the tables from section 2.7. If the microservices offer the required fixes to these problems, then it is acceptable to start the long process of migration. Nevertheless, it is important to take into consideration various factors before starting, such as the skills of the team and the personnel that are going to be in charge of the process. If the team is inexperienced in distributed systems development, it may be necessary to hire external consultants and developers for the purpose. This will result in increased costs during the migration process, even though, the entire idea of microservices implementation might prove to be a bargain in the future. Another obstacle could be the existing hardware - if it has to be upgraded to perform better with the new architecture, it is highly likely to result in extra costs. We should also not forget, that aside from the migration, there exist some other options. One of them is to leave our monolithic application behind and start developing the microservices from scratch. Depending on the existing system, the data it stores and manages, this may be an acceptable solution. However, it may prove too radical approach and usually companies will want to keep their existing applications and refactor them instead. Another viable option is to replace the existing system, or the part of it we want to change, and use SaaS solution instead. This may be the cheaper option and is also quite flexible, taking into account the options offered by the current SaaS market. Nevertheless, this is just a small portion of the considerations, that must be taken before we decide, if we are ready to give up our current system and migrate to microservices. The decision, that is taken will have huge effect on the future of the organization, therefore, the situation should be carefully assessed.

After all the pros and cons have been weighted and the decision to move on with the microservices has been taken, the process of migration can be started. The first stage of the process is to analyze the current system. By gaining deeper understanding about the existing system, we will be able to see the patterns, dependencies and the interactions between the software components. The analysis will show us the parts of the system, which are closely related. These parts will represent the different domains in our system and the actual bounded contexts [29]. The idea is to find the so-called seams [28], which will reduce the inter-component interaction. By identifying such segments, we will decrease the system complexity and could write a generalized component diagram with all the different domains. Remember, at this stage we are still focused on understanding the main domains in our system, not specific parts of it.

In order to conduct such a system analysis, we can use the help of various software solutions, which check the dependencies in the code and produce diagrams. As we mentioned earlier, one of the most widely-used solutions for database schema extraction is the open-source tool SchemaSpy [92]. It is based on Java and produces thorough reports for the relationships and dependencies in a database. SchemaSpy supports most of the popular relational databases such as Microsoft SQL Server, MySQL and PostgreSQL. In order to create a report, SchemaSpy provides a command-line interface [92]. The output of the tool can be accessed via a HTML file and opens in the browser. There we can examine the different database entities in table format, with information regarding their relationships and dependencies. Furthermore, if we have the DOT language [93] installed, we can see a visual representation of the relationships. Indeed, this is SchemaSpy's most powerful feature for the purpose of this thesis, as it enables us to identify the most decoupled parts of the system by looking at the database schema. It is important to note, that aside from the open-source SchemaSpy, there are multiple alternative options. A good example of one is DbVisualizer [16]. It provides similar functionality as SchemaSpy, but in a more user-friendly GUI.

Aside from the database structure, the other aspect of the software system, that is of interest for the decomposition, is the code structure. Luckily, there are multiple tools, that we can use for this purpose as well. One of them is Structure101 [17]. It helps to visualize the source code and find its underlying structure. The tool comes in the different forms - integrated within the IDE, stand-alone version with graphical user interface and as a command line tool. The company, that is behind Structure101 states, that it is language independent, however, there is predominantly information regarding Java and .Net. The output is in the form of a graph, depicting the structure of the source code. The nodes represent the software components and the edges - the dependencies between them. Another alternative to Structure101 is NDepend [94]. It provides the same graph visualization functionality as

[16]`https://www.dbvis.com/`

[17]`https://structure101.com/`

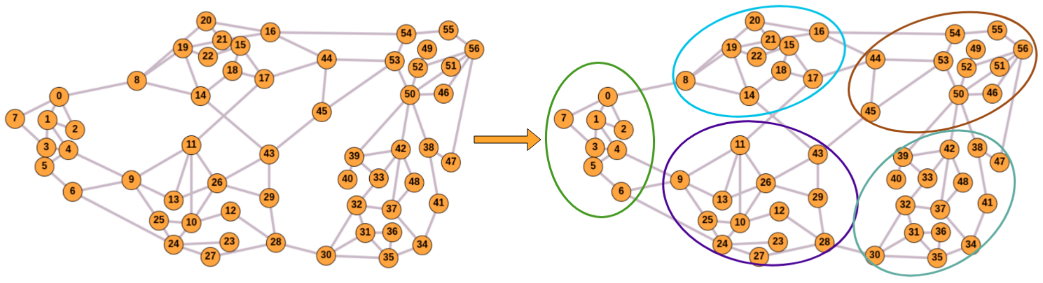
Figure 15: Example of Component Grouping

Structure101, but builds on it with a new useful feature - a dependency matrix. The matrix is particularly useful when the system of interest is large and the graph becomes hard to understand. It looks like a heatmap and significantly simplifies the component dependency interpretation. The only downside of NDepend is, that it currently only supports .Net source code. However, similar solutions with the matrix output can be found for different languages, for example - Java [18].

After we have run the tools, which were chosen, and have their output available, it is time for us to start finding patterns in the current architecture. The main goal of this stage of the process of decomposition is to find the domains within the system. We can assume, that the more the relationships (edges) we see between a group of entities (nodes), and the closer they are, the bigger the chance, that they belong to the same domain. And the opposite, if two different modules are far away from each other and share a low number of relationships, we can infer, that they are part of two different domains. Figure 15 depicts a simplified version of a possible component grouping. Usually the graph output will be much more complex, with less evident borders between the domains. In order to address such confusing graphs, the Ndepend tool offers matrix representation for better understandability and better insight into the structure. It is vital to take into consideration and examine the outputs from both the code dependency and the database schema extraction tools. They should be quite similar, but if some inconsistencies are found, we should address them by selecting the better logical option of the two. As this is one of the most important concepts in this thesis, we would like to stress on it and summarize:
We aim to group the components, represented in the graph into domains, by judging on the distance between them and the number of relationships, which connect them. After the grouping has been made and we have identified the

[18]`https://www.jarchitect.com/dependenciesview`

domains in the system, we are ready to move to Phase 2 of the transition.

ERP System Example

ERP systems are convenient to decompose into domains, because of the easily recognizable bounded contexts, that they are comprised of. For each different aspect of the enterprise operations, there is a designated part of the ERP system. There is a section responsible for the sales, another one responsible for project management, third one - for manufacturing and etc. We can assume, that the software components, that are accountable for the serving the needs of a specific domain in the company, will have much more interaction within the group, as compared to outside its borders. Let's take Odoo as an example and more specifically its Project Management domain. We can expect a high degree of interaction between tasks and issues, and a lower one between issues and manufacturing. On the one hand, it is likely, that issues will arise from the manufacturing context, but they will be a small percentage of all the issues reported. On the other hand, we expect most of the issues to have a task related to them in order to be solved. Therefore, it is evident, that we have a good foundation to start the process. However, these domains are still too big to qualify for the term microservices after migration. Each of them is still a substantial part of the system and is hard to maintain and improve by following the CI\CD principles.

Therefore, we have to analyze these domains and split them into subdomains of the appropriate size, which will turn into microservices on a later stage. For this purpose, we should use the mentioned tools and investigate the patterns. It is clear, that when we run the tools against such a big system we can expect a really complicated graphs for output. We have run SchemaSpy on the Odoo database, which consists of over 1000 tables and included a fraction of it in the Appendix (Figure 23). It is obvious, that based on such an output, it will be hard to recognize any patterns and subdomains. However, the discussed tools provide options to zoom in into the specific parts of the graphs. For example, SchemaSpy provides, aside from the complete database schema, decomposed output for each table. This function can be helpful, after we have identified the main domains and zoom in into the relevant tables. Then we will be able to see the relationships and dependencies of each element in greater detail. It is very important not to create too small microservices, but to find the sweet spot for granularity instead. If the size of the microservices is too small, we will have too much calls between them, which will result in slower requests. That is the reason, that this first phase of the process is the most important. We need to find the balance and identify the bounded contexts with the required level of granularity. It is a hard and sensitive task, especially for a big system such as an ERP, but if done well, it will enable us to achieve greater success with the migration to microservces. By using this approach, we can identify the subdomains and move on to separate our first microservice.

Technology Stack

- **SchemaSpy** - Powerful open-source command-line tool, that extracts the database schema of the existing application.
- **DbVisualiser** - Provides similar functionality as SchemaSpy, but with a GUI.
- **Structure101** - Used to visualize Java and .Net code architecture, including relationships and dependencies.
- **Ndepend** - Identifies .Net code structure and produces reports in the form of graph or matrix.

4.2.2 Phase 2 of the Migration

Theoretical Part

After we have analyzed the current system architecture and placed the software components into groups of the appropriate size, we can continue with the migration process. At this point, we are still at a very early stage in the process and it is possible, that a large part of the team, responsible for the migration, will not be experienced enough. Therefore, we suggest to start by finding the most peripheral component groupings and decide which one will be the easiest to start with. This is essential, because by starting the decomposition with such a subdomamain, we won't interfere much with the core system functionalities. Furthermore, the team will have a chance to understand the process and if any errors are made, they wont affect essential parts of the system. The selection can be made after carefully examining the graphs, produced from the previously mentioned tools. The main goal will be to identify such component groupings, which interact with a minimum number of other groupings and are placed in the outer parts of the produced graphs. By doing so, we will reduce the room for error, by minimizing the parts of the system, that can fail after the decomposition of the first service.

When we have selected the sub-domain we want to focus on, it is time to perform the required transformations in order to detach it from the monolith and convert it to a stand-alone service. All the work should be done in background, without changing the current system functionality before the changes are fully tested. The entire system in production is left unchanged and the transformation efforts are done in parallel. Before we start implementing the new features, we should again do analysis. However, this time it is focused only on the subdomain we have selected and is therefore more specific and in greater detail. We must focus on all the source code, which is connecting this particular subdomain to others. A good example for such a code statement are the package imports. We should be particularly careful with the examination of the imported packages and check if they are part of a different domain. If a package is part of the subdomain, that we are working on, basically no changes should be done. The source code location of the imported package will stay the same and therefore, the way it is called won't need transformation. However, if a imported package has been identified to be part of a different subdomain, the situation becomes more complicated. We must change the way this interaction is being done by introducing service calls instead of simple import statements. We must find and replace all the function calls, which point to different subdomains with service calls. The reason is, that by decomposing part of the monolith, the location of the codebase becomes spread in different locations and we now have to deal with a distributed system.

There are multiple frameworks, which help us implement this behavior and we will mention some of them in the following lines. The most popular microservices framework for Java is Spring [95]. It helps with the entire development lifecycle of distributed applications, but it mostly simplifies the implementation of the API. Spring offers a way to build web services following a RESTful approach. The way it works is that it accepts HTTP requests at a specific location, say http://localhost:8080/greeting and responds with a JSON representation of a greeting: {"id":1,"content":"Hello, World!"} [96]. This information is then used by the service, that is calling the API call - the service consumer. The general idea when we are migrating, is to create a different address for each separate service and then specify the returned values, based on the received GET requests. Let's imagine the greeting function will accept a parameter with the persons name. We can pass it by simply writing http://localhost:8080/greeting?name=Petar. The returned value will be Hello, Petar. This is the basic approach of how a microservice-based application works. There are HTTP requests instead of function calls and the information is passed in the appropriate format (in this case - JSON). Obviously, there is much more depth and functionalities offered by the framework, but we will not focus on them, as the aim of this thesis is to define the general approach, instead of explaining the details. Aside from Spring, there exist multiple frameworks already for most of the available programming languages. In the case of Python, we can distinguish two notable frameworks, that help with the creation of APIs - Flask [19] and Django [20]. They provide similar functionality as Spring, by offering a simple way to develop a distributed application. The difference between them is that Flask is easier to start with, as Django provides more functionality for the price of higher complexity.

Another crucial aspect from the migration process is the database decomposition. We suggest to use the database per service pattern [30, 17]. In order to implement such a solution, we need to identify the database tables from the monolithic database, that are connected to the specific subdomain we are currently decomposing. Afterwards, a new service-specific database must be created, consisting of these tables. All the data from the monolithic database should be migrated into the new database in order to prevent loss of information. As the database is being split into pieces, its constraints and relations must be split as well. The decomposition of dependencies is the most sensitive part of this stage of the process. Foreign key relationships, triggers and all the restrictions, which are defined on database level must be dropped and new rules must be set up. In order to deal with database constraints, we suggest to replace the database logic with source code implementation. For example if a constraint is set not to allow duplicate entries in two database tables, we can put that logic into the source code and the check can be performed there. Triggers can be fixed using the same approach, i.e. by dropping the ones, which span across the monolith and microservice databases and implementing the logic using source code.

To address the issue with the foreign key relationships, the best option is to break them and replace them with service calls. The role of the foreign keys is to normalize the data within the database and prevent redundancy. Usually, when we want to select data from the table where the foreign key reference is made, we create a join between both of the tables and in this way, we retrieve the required information. Clearly in this situation only one database query is needed, which looks in two tables. However, this is not feasible when we are using the microservices architecture and such a join is impossible, as the two tables are located in different databases. The solution is to move the join logic to one level above the database - to the business layer. Then when a query is called, requesting data from another service's database, a service call will be executed to the specific service location, forwarding the request. After receiving it, the other service will reach to its database, pull out the required information and send it back to the requesting service. Of course, these API calls have to be

[19] `http://flask.palletsprojects.com/en/1.1.x/`
[20] `https://www.djangoproject.com/`

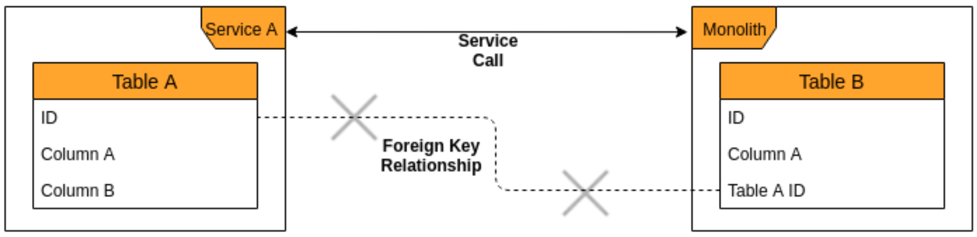

Figure 16: Changing Foreign Key Relationship to Service Call

pre-defined and adhere to some standards in order to have understanding between both of the services. For example, a call can be in the form: http://localhost:8080/service_name/search_function/searched_table/target_id. By using such a pattern, we sacrifice one more database select statement in exchange for getting to keep our services with separate consistent databases. In this situation, as we will also see in the following lines, one trend can be recognized: In the microservices architecture, the databases are used primarily as a storage mechanism, instead of performing many operations. The responsibility is shifted towards the business layer, as most of the functionality becomes distributed and spans multiple domains.

Transactions are another aspect, that we might have trouble dealing with when decomposing the database. The idea of a transaction is to either apply all requested updates, or apply none even if a single requested change fails. They are crucial to the efficiency and the performance of the database. However, as we can expect, the transactions can span multiple domains at once and this will be a problem, when introducing different microservices, each with its own database. But as we have mentioned above, the general logic is to transfer the responsibility to the business layer and the service calls. Instead of having the transaction performed between tables, we do it between services. Such distributed transactions can be either orchestrated or using the choreography pattern. Using the first suggestion, a separate service can be created, which will be responsible to gather the information from all the services involved in the transaction and keep track of all the statuses. If even a single service sends a signal, that something has failed, a rollback is performed everywhere. The choreography solution doesn't involve creating a separate service, but in this way each service must support the tracking functionality. Each service informs the next one regarding the current status and if a single fails, the process is repeated in the opposite order, with rollback instructions.

A further possible obstacle when decomposing are the tables, created by many to many relationships. They use information from two separate tables and need to be updated when the data in the tables they are referencing is changed. If such tables exist in the same service, there is no problem, but it is quite common, that they span two separate domains. In order to deal with such tables, we suggest two different options. One of them is to keep the table in one of the services and use API calls to access it from the other location. It makes sense to leave the table of interest in the service, where it will be referenced more. In order to identify which one will use it more, we have to analyze the current situation. By keeping it in one of the locations, we increase the number of API calls needed to retrieve data, because if a request is made from the service, in which the table is not located, it has to call the other service's function in order to get it. However, the beneficial part from this approach is that there is no redundant data in two separate databases. Another solution can be to have the table in both of the locations and update them both when a change is made. This will lead to fewer calls when we need to get some data out of the database but there will be redundant information in the databases. Therefore, as we can see, there is an obvious trade-off between both of the options and we should select the one, that will be the best fit for our needs. The entire database decomposition is an art of its own and we can provide just basic guidelines. However, we can recommend a book for further reading, which focuses on the topic in detail - Refactoring Databases: Evolutionary Database Design [97].

We have encountered other approaches [6, 73], where the authors have suggested to implement the supporting patterns first, before introducing any microservices. This is possibly a better approach, when we are dealing with smaller systems, which don't have that many peripheral domains. The reason is, that when the system of interest is smaller, we can expect it to have more core components, which need to be reliable and robust during the decomposition process. Implementing these patterns help us achieve the required reliability. However, when dealing with a large and complex system, such as an ERP, we aim to improve the confidence in the team by starting the decomposition with a less important part of the system. This first, separated service will not require much of the necessary microservices patterns, which will be needed, when the number of services grows. For example, the API Gateway won't be necessary yet, because the client won't interact directly with the selected service.

ERP System Example

After giving the theoretical background of our idea, it is now time to turn our attention to the ERP system and provide more concrete examples. As outlined earlier, initially we have to zoom in to a part of the system, which we deem will have small significance on the entire system performance. We suggest to focus on one of the domains, then look into the component groupings, that have been identified within it. The idea is to select such sub-domains, that interact only with other sub-domains within the boundaries of the current domain. If a component group needs to interact with a group residing in a different domain, or is depending on such a group, we should leave it unchanged for the moment and focus only on those, which have relations strictly within the cluster, we are focused on. For the purpose of this example, we will focus on the Project Management domain and the Todo Lists module in particular. Figure 17 depicts a simplified schema of some of the modules in the selected domain. As we can see, the Todo Lists depend only on the Tasks Module, in contrast to the other three modules, which are interconnected. The tasks are created, based on issues and user requirements. The same goes for the issues, which can be reported by the users and entered in the system. The Todo Lists in Odoo are working only in addition to the Tasks module and a Todo can be created only from an existing task. Furthermore, as we can see on the Figure, there is a customization module, which is extending the functionalities of the default Todos, based on the company's requirements. For the purpose of this approach, this customization extension will not be counted as a separate module, but instead as an addition to the Todos. Therefore, we suggest to combine both of the modules into one service in order to keep the functionality in the same place and when changes are made, they will be applied only once. In the decomposition process, this approach can be generalized and used in every situation, where inheritance is involved. By keeping everything related on one place, we are moving closer to realizing the idea of the microservices architecture and achieving autonomy of the separate services.

Now that we have selected our first part of the system we would like to decompose, we can start examining it in detail in order to check for the database relations (e.g., foreign keys, triggers and etc.), source code dependencies and interactions with the other modules. In this case, the only module, that is related to the Todo Lists is the Tasks. As a first step, we can check the database schema produced by SchemaSpy or one of its alternatives. We can find information regarding each of the tables within the Todos module, along with their direct dependencies on the Tasks module. This information will be used to split the database. The tables, that belong to the Todos module will be removed from the monolithic database and a new, small database will be created specially for the future Todos service. After examining the module of interest, we have seen, that only two relationships exist between it and the Tasks. They were identified as foreign key relationships and as

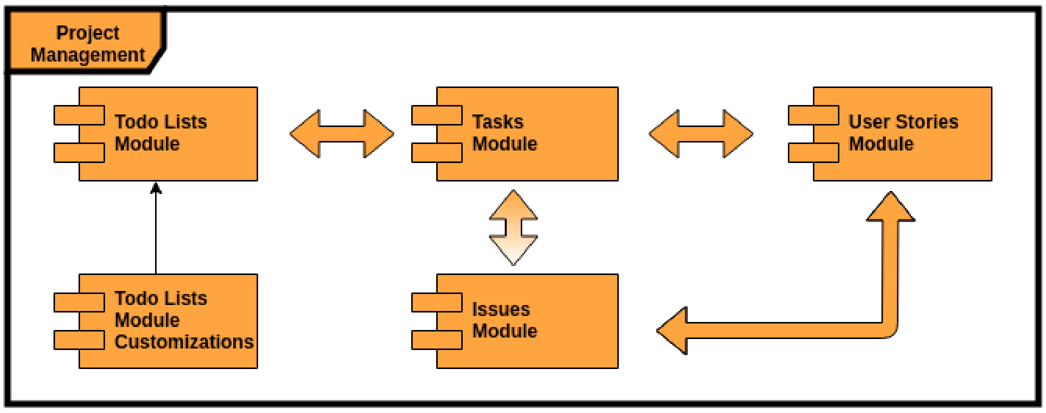

Figure 17: Todo Lists as part of the Monolith

proposed in the previous section, the first step for their decomposition, is to drop them. When there are no more database relationships interfering with the migration, it is time to replace their logic with service calls. This will be our first steps in API creation and distributed service calls. The approach, in which such relationships are transformed is always the same, therefore, we are going to provide an example only with one of the two available ones - the one, that is responsible for the context of Todos. It serves the purpose to store the Task IDs in the Todos table, so a connection between both can be made and it is clear which todo entry to which task refers. A single todo entry can have only one task related to it, but each task can have multiple todo entries associated with it. If an attempt is made to insert a value in the Todos table task_id column, which doesn't exist in the ID column of the Tasks table, an error will appear. We now have to replace these restrictions and apply them using service calls. As the programming language of Odoo is Python, we suggest to use Flask for easier API creation. As a first step of the API creation, it is necessary to create addresses for both of the locations: /localhost:8080/todos and /localhost:8080/monolith/tasks. When the locations are specified, it is time to start implementing the functionality. Let's imagine we want to look at the task details after we have seen it in our todo list. We reach to the Todos service's database and get the value from the task_id column of the current list. Then, as it is not possible to simply create a join between the tables anymore, we need to ask the Tasks module to reach to its own database table and find the task by the provided ID. After retrieval of the necessary information, it is passed back to the Todos service for displaying or further processing. The other relationship, that is responsible for the time, is decomposed by following the same principle.

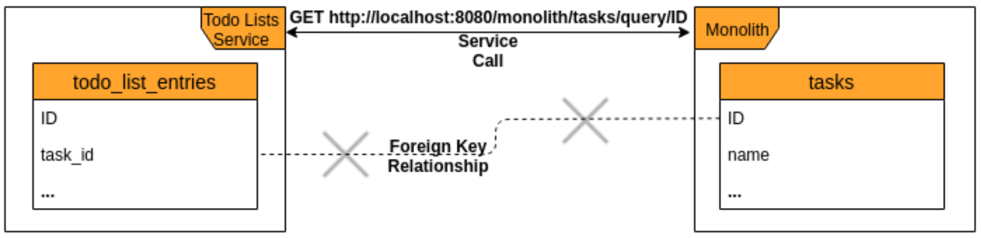

Figure 18: Todo Lists Foreign Keys

After we have finished transforming our database logic to service calls, it is time to start with the code refactoring. It will most likely always be the harder and more cumbersome part, as there will be more function calls to external modules, than database dependencies. The main idea in this step is to check for all the imported modules in the beginning of the files, which were migrated to the separated service. There will be some modules, that are imported and come along with the standard language libraries (e.g. Python standard libraries). They won't need any changes, as they are already available after installing the specific programming language distribution. However, most likely, there will be part of the imported modules, that does not come with the default libraries and is not part of our system's modules. Such modules must be installed on every service, that needs them in order to operate. Furthermore, there will be imports, that are part of our system, but are not located in the current domain. These modules need to be called with API calls in order to be reached, because they will represent different services in the future. It is crucial to make a differentiation between the different groups of imported modules and deal with them accordingly. Here is a simple example with the Odoo ERP system:

```
from datetime import date
from flask import Flask
from monolith import Tasks
```

The datetime package is available from the standard Python distribution, therefore we won't need to do anything to address this line. However, Flask is the Python micro web framework, which will help us create our APIs and doesn't come with the default language installation. Therefore, it needs to be installed on our service in order to be able to import it. This can be done by manually installing such modules after identifying them. Usually, in case of such dependencies, as is the case for Flask, it is advisable to include it in a container image and have it available after running the container. This option, however, will be discussed in the next phase of the migration process and for now, as we haven't set up any containers yet, such dependencies must be installed manually. The third statement - the importation of the Tasks module, is the one, which should be payed most attention to. This import statement is deleted and we should now start examining the rest of the source code and find functions called from the Tasks module. Wherever we find a function call pointing to the Tasks module, we should replace it with an API call in a similar manner, as we did with the database dependencies. However, the process doesn't end here. We have made the calls from Todo Lists service to the Tasks module possible, but we need to perform the same procedure at the other end and make the calls from the opposite direction available. Furthermore, we have to tune the code in the Tasks module, to understand which function is requested with a specific API call. Again, Flask must be installed on the monolith and imported into the Tasks module. For all the requests, that were already defined in the Todos service, responses must be created. We won't delve into this matter, as it is too technology-specific and the information exists in the documentation of the selected API creation package. Then, we repeat the process with the identification of the functions, which are called from the Tasks module, pointing to the Todo Lists service and replace them with service calls. As a final step, we go back to the Todo Lists and tune the source code in order to enable it to understand the service calls, that will come from the Tasks module and respond to them. In general, this is how the code-level dependencies are decomposed and the process can be applied when separating arbitrary services from the monolith. Here is a simplified step-by-step summary of the source code decompositon process:

1. Identify the import statements in the beginning of all the files in the separated service (e.g. Todo Lists), which don't come with the standard programming language distribution.

2. Install external libraries and dependencies - e.g. Flask.

3. Replace the function calls from the service (e.g. Todo Lists) to other modules within the system (e.g. Tasks) with service calls.

4. Tune the code in the monolith (e.g. Tasks module) to understand and respond to requests, instead to service calls.

5. Replace the function calls from the monolith (e.g. Tasks) to the decomposed service (e.g. Todo Lists) with service calls.

6. Tune the code in the service (e.g. Todo Lists) to understand and respond to requests, instead to service calls.

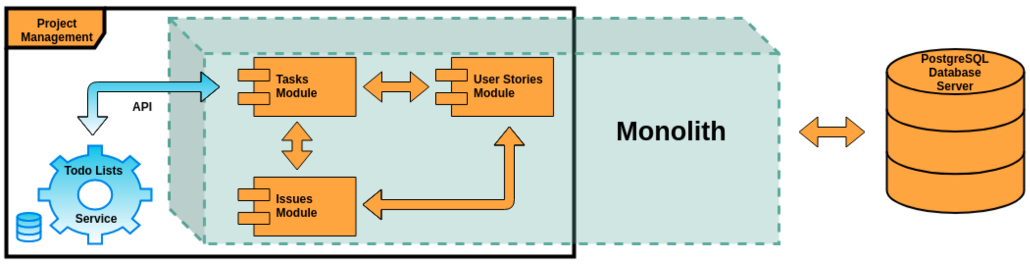

Figure 19: Todo Lists as a Separate Service

The replacement of all the source-code and database level dependencies with service calls means, that we have separated our first service from the monolith. The current situation can be seen on Figure 19. The Project Management domain now contains a the Todos Service, which has taken the place of the module with the same name. This new service has its own database, which has been split from the existing database belonging to the monolith. The communication with the Tasks Module is now being made via a specially designed API, which can be understood from both of the participating software components. As we can see, the resulting service is already outside of the monolith, which holds everything else, except the Todos. All the components inside the monolith remain the same. The interaction between the different modules and the database calls remain the unchanged. It is evident, that the core functionality will remain the same after the separation of such a small part of the system, but this was exactly our goal. The entire process is cumbersome and will take a lot of time, therefore, we have to make small steps in order to achieve full decomposition.

After we have our first service set up, there are two alternative paths to choose from. The first one is to continue decomposing all the peripheral services in the Project Management domain. They, as is the case with the Todo Lists module, won't interfere with the other parts of the system, but only with a single module, that they depend on. By using this approach, we will increase the number of our services and the decomposition skills of the migration team members. However, this will come at a cost - as the number of microservices grows, it will become harder to maintain them all without introducing the required software components and patterns to help us manage them. In order to fix this, the second alternative is to move to Phase Three of the approach and introduce the required patterns, that will ease the deployment of new services and their maintenance. Clearly, we have to weigh up the different alternatives and our priorities and select the more suitable option for the current system's situation. If we are dealing with a small to medium-sized system, it will be acceptable to decompose all the peripheral modules into services. However, if the system is large, as is an ERP, we suggest to move on with the introduction of the microservices patterns in Phase Three.

Technology Stack

- **Flask** - A micro web framework, used to build distributed applications with the Python programming language. Facilitates custom API creation.
- **Django** - Python web framework providing more functionality than Flask.
- **Spring** - Comprehensive Java framework, which can also be used for API creation for microservices applications.

4.2.3 Phase 3 of the Migration

Theoretical Part

At this stage of the process, after the system has been analyzed and the first service has been decomposed, we suggest to continue with the introduction of the microservices patterns, that will help us with the introduction of further services and their future management. We will organize them in a list, with the suggested order of implementation.

1. **Containerization:** Each service, that we start to decompose, must be put into a separate container. As we have one already existing decomposed service, we should start by creating a container for it and include all its dependencies there i.e. software languages, libraries, frameworks and etc. We should note, that on top of all the dependencies, there is the option to containerize the database as well. If we decide to apply this solution, each microservice will be easily setup with all the required components. However, we should be careful regarding the database containerization and we should consider the overhead, which can slow down our application [98]. Therefore, each microservice should be assessed individually and the ones, which will require fewer I/O operations, with more static databases, can use the database containerization pattern. However, if a lot of I/O operations are required, we should deploy our database outside the container, as otherwise, we will add further complexity to the service. By introducing the containerization, we will reduce the room for error when deploying the services, as the containers will support the same

environment on each machine, on which they are run. This is a crucial aspect of the microservices architecture, as we aim to achieve small, regular releases and the deployment is much harder, when the environments differ. Therefore, this is considered as one of the prerequisites of CD/CD Pipeline, which will be discussed next. The proposed tool for creation and managing of containers is Docker [99]. It is the most popular containerization solution and is platform-independent. Furthermore, there is already a large number of ready-to-use Docker images already available online in the Docker Registry. They can simply be included in the container, without the need to perform any additional operations and create images from files.

2. **Setup CI/CD Pipeline:** One of the greatest advantages of the microservices, is that they enable us to apply regular updates to each service independently. There is no need to rebuild the entire system for a simple update of a small part of the it. CI/CD is the main driver behind this functionality and we suggest to setup a pipeline before moving any further, so we can take advantage of the automation, provided by the methodology. The process starts by separating the codebase of the service, that is about to be decomposed in a separate code repository. As we already have one existing service, it is essential to create a separate repository only for it and put it there. By doing this, we will improve the degree of separation of the service and will keep the related code in the same place. After we have the codebase separated, we can prepare the CI Pipeline, which will automate the deployment of new updates. We can configure it according to our needs, we can include automated tests, send notifications and much more. As new changes to the code are committed, the pipeline will start executing. If some tests fail, the responsible developers and other stakeholders will be notified about the issues. They should then fix the errors before starting to work on a different task. We suggest to select the tool Jenkins [100], as it has become fairly popular during the recent years and there is a large community, that can resolve issues and answer questions. The proposed solution for configuration and running of automated tests is Selenium [101], as it can be integrated with Jenkins.

3. **Service Registry:** Now that we have achieved automated deployment of the services into containers, we have met the basic software infrastructure standards to create microservices. However, there is a lot of work left to configure the communication and the coordination between these services. Therefore, as a next pattern we would like to introduce a Service

Registry. It serves the purpose to store the location of all the available service instances in a list and provide that information when needed. On startup, every service, that is being decomposed from the monolith, will be automatically added to the Service Registry, in order to be found by other services, the API Gateway or the Load Balancer, both of which will be introduced on a later stage. Each service instance sends a heartbeat with a pre-defined timeout. If the heartbeat stops, or the service is stopped, it is removed from the service registry and the reason is written in a log file. The Service Registry must be implemented in a robust manner, as it will be used by the entire system as a reference and could become a bottleneck. There are many available libraries online, which help with the service registry implementation for most of the programming languages. A good example for Java is Netflix's Eureka [102] and for Python there is also one available on PyPi [103].

4. **Circuit Breaker:** After the introduction of the service registry, it is time to consider improving the reliability and robustness of the system. This can be achieved with adding a new functionality to the system - a circuit breaker. Its role is to keep track of the consecutive failed requests from a service consumer to a service provider and if a specific threshold is reached - a timeout is initiated and an error is returned [47, 17]. The reason for the importance of this component is, that if a specific service is overloaded and we keep forwarding requests to it, it may not recover. Furthermore, if even a single service fails, this can result in cascading to other services and soon the entire system may be down, just because a single service is not responding [47, 17]. For example consider a synchronous communication between service A and service B. Service A (the service consumer in this context) sends a request to service B (the service provider in this context) requesting some information. Until service B returns the required information, service A is waiting for it and is not performing any other operations. Then, imagine service C, which is requesting information from service A. However A is not available and service C also starts waiting for service B indirectly. The circuit breaker tackles this problem by introducing a timeout after a pre-defined number of failed requests. This timeout gives time to the problematic service to recover and prevents the entire system from failing. Instead of the requested information, it is possible to return an exception or a cached data from the service provider [6]. It is obvious, that the introduction of such a pattern is critical for the success of a microservices-based system and therefore we recommend implementing it before continuing with the introduction of further services. A recommended solution for Java is

Hystrix [104] and for Python there is also one available on Pypi [105]

5. **API Gateway:** Up until now, all the work has been done under the hood and without direct effect to the users of the system. We have tried to delay the changes, that will be in direct contact with them as much as possible. With the introduction of this pattern, however, we are going to create a new layer in the communication between the user and the system and prevent the direct interaction between the two. All the requests made from the users will pass through the API Gateway and won't interact directly with the service needed. When a request is made from a client, it passes through the API Gateway and reaches the service registry, where it looks for available instances of the requested service. Then, after retrieving the required information, it is transmitted back to the user again via the API Gateway. This abstraction is needed, because as the decomposition is being made, a lot of work will be done to the system, which must not affect the end user. Furthermore, the routing when using microservices architecture is way more complex, as the service addresses change frequently and this additional layer is needed to provide the dynamic routing functionality. Probably the most popular solution is provided by NGINX [106] - NGINX Plus, which is language and platform-independent and offers the required functionality. Furthermore, if the NGINX solution is chosen, we could also benefit from the load balancing functionality, which we will discuss next.

6. **Load Balancer:** After the introduction of the API Gateway, the system starts to feel more flexible, robust and we are drawing closer to realizing the microservices architecture. However, we need to address one more aspect, which is actually one of the main reasons, that makes people migrate to microservices - the scalability. In order to improve the performance, we need to distribute the requests, which a service needs to serve to the different service instances. This job is performed by the Load Balancer. It queries the service registry for the available instances of the requested service and using a special algorithm distributes the load between the separate service instances. By implementing this solution, we are making sure, that the requests are served in an efficient manner and are equally distributed between the available service instances, which enables more processes to be executed in parallel and improves the productivity of the entire system. The load balancing software, which is recommended, is again NGINX or NGINX Plus.

7. **Clusterization:** We have mentioned multiple times until now the term service instance. Here we will explain what exactly it is and how it is

created and maintained. The service instances are at the base of the way the microservices architecture actually scales. Each instance is essentially a replica of the service, which can be created or deleted, based on the current load and needs. We have proposed earlier to put each service in its own container and deploy it. The deployment of the separate containers is straightforward, but the introduction of service instances and their management is a troublesome task. In order to help us deal with this problem, we have chosen to use the tool Kubernetes [50]. It aims exactly to scale containerized microservices, based on the current load of the system. The way it works, is that it creates clusters of containers, which contain single service instances, adhering to the Service instance per container pattern [48, 17]. Kubernetes scales each cluster's instances up and down, according to the requests for the specific service and the network load. Furthermore, it provides the means for orchestration and management of these instances. For example, if a specific instance fails, it brings it back up. The scaling of separate microservices is, as we have mentioned earlier one of the greatest benefits of this architecture's adoption. Therefore, the introduction of this pattern is a critical factor for the success of the system we are trying to build.

It is important to note, that some of these patterns (i.e. API Gateway, Service Registry, Load Balancer and Cluster Orchestration Tools) may become the weak points of the system. As they will be constantly under heavy load and responsible for the operation of the system, we must make sure, that they are available all the time. It is not acceptable to have a failed API Gateway bring down the entire system. Therefore, as a solution for that we could have a pool of replicas of the API Gateway all running simultaneously with the one, that is currently active. If the API Gateway fails for some reason, each one of these replicas will be able to step in and serve the end-user [107]. Furthermore, it is also possible to have all of these replicas running and use load balancing for the API Gateway as well, which will distribute the load and will reduce the chance of failure [6]. These two solutions can also be applied to all the other components, that can become a point of failure and break the entire system.

The order of the implementation of the patterns has been selected like this, in order to minimize the effect to the end user. Up until step 4 - the introduction of the API Gateway, there is no change in the way the end user communicates with the system. Even after its introduction, the end user won't notice the difference, but it is advisable to monitor the system and the new components in the first few weeks, in order to check for issues. There are more patterns, that can be introduced, which will further help us manage the

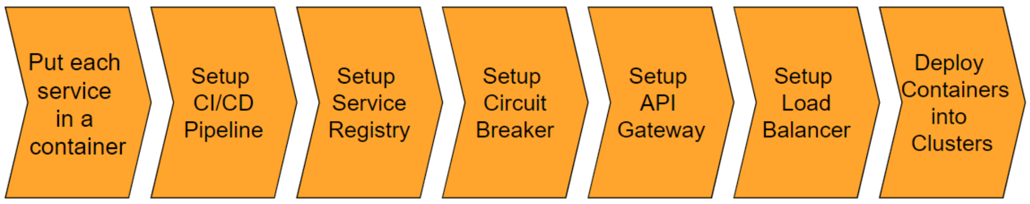

Figure 20: Proposed Order for Introduction of the Microservices Patterns

microservices architecture, but the ones, that were proposed are sufficient to continue with the migration process in Phase four.

ERP System Example

The current situation, as shown on Figure 19, is that there is one autonomous service separated and it is communicating with the Tasks module, which is located within the monolith, with a REST API. In order to continue with the migration, it would be best to apply some patterns. The first step is to create a container with all the dependencies and put the Todos service into it. In this container, we have to include all the requirements, which will be needed to run it i.e. the Python language and the Flask library, which is used to create the API endpoints. On top of these dependencies, we can also include a PostgreSQL database image in the container, because the database, which will be maintained by this service is not expected to be large in size, with lots of I/O operations. The database inclusion is strictly individual for every separate service, depending on the volume of the I/O opeartions. Furthermore, a good approach is to also put a OS image into the container in order to replicate the environment available on all machines as much as possible. For example, if the server's OS is CentOS and the one, where development is being made, is Ubuntu, by putting the CentOS image in the container, we will replicate the target environment, and thus, remove the errors from OS inconsistencies. All of these images are available in the Docker registry and will be easy to deploy them in the container.

After the container is being created, we can continue by preparing the CI/CD pipeline. The first step here will be to create a separate repository for the Todos and separate its codebase from the monolith repository. By doing that, the monolithic codebase will shrink and on its place will appear many smaller, easily maintainable codebases. Now, that we have prepared the repository, we can start configuring the CI/CD pipeline with Jenkins. Usually, it consists of three stages - Build, Test and Deploy [108]. The first stage is the one, where the changes are merged and the application is built.

If the build is successful, the pipeline moves to the next job - the testing. It is necessary to define the tests according to the specific needs, as there can be unit tests, integration tests, regression tests and etc. If all the tests are passed, the pipeline enters the Deploy stage and applies the changes to the production (or the target) environment. From this point on, each service that gets separated, will have its own CI/CD pipeline, fulfilling its needs.

Next comes the introduction of the service registry and its configuration. As we have currently only one service, we need to make the registry aware only of its location. This is done by writing a block of code, located in the service's codebase, which will send a HTTP POST request to the registry upon initiation of the service [109]. This request will include information regarding the service and its location (IP). Afterwards, a periodic heartbeat will be sent, with HTTP PUT request, updating the service status in the registry [109]. If the service fails to send the heartbeat, or it is stopped, the entry is removed from the service registry. This functionality must be included in every further separated service from the monolith.

After the service registry, as suggested earlier, we come to the introduction of the circuit breaker pattern. In contrast to the service registry, however, it is not a single component, which is accessed by all the services in the system, but instead a circuit breaker is located in every service and every service is responsible for tracking its outgoing requests. In the case of Python, it can be implemented by importing the required library [105] in each of the files in the service. Then, when defining a function, which will include API calls to other services, we use a function decorator, which allows us to add the required functionality. Our Todos Service will need to have a circuit breaker included in all the service calls towards the Tasks, as these are its only interactions outside of its scope. We have to define a threshold for the number of requests, before an exception is thrown and the default timeout period. From this point on, it is recommended to include the circuit breaker in all the future services.

API Gateway and Load balancing will be combined into one step in our practical example, as they will be served by the same technology - NGINX. As our service is still in the periphery and doesn't have much load, it doesn't make a big difference for load balancing and rerouting. However, as the number of services and their instances increases, we will need this functionality and must implement it at this point. For now, however, most of the routing will remain static and will lead to the monolith, even though there will be an API Gateway layer between the monolith and the end-user. There will be just one small part of the system, which will be dynamically rerouted and will therefore need the API gateway and load balancing - the Todos Service. When one tries to access it, the request will go through the API Gateway, which will look into the Service Registry for available instances and will be rerouted to the most appropriate one, based on the load balancing algorithm.

Next, we come to the part, where the scalability is addressed and the service instances are initiated and managed. Again, as we have only a single service, this won't be of a great use yet, but on a later stage, the introduction of clusterization will probably be the biggest advantage of the adoption of microservices. As we have chosen to put our services into containers, we will need to orchestrate these containers and replicate them, based on the demand. Kubernetes is undoubtedly the best solution for managing containerized services and their instances, and therefore, we have decided to adopt it for our architecture. It will monitor the load of the available services and if needed will initiate new instances to serve the demand. For example, let's imagine, that we have separated the Issues module and have introduced an Issues service on its place. Usually, we would expect a low amount of concurrent requests, therefore we will keep few instances available by default. However, if there is a problem with some newly introduced functionality in the Sales domain, the service will receive a spike in the incoming requests. Kubernetes will be responsible for replicating the service by producing more instances and thus, dealing with the increased load. After the number of requests decreases for a specific amount of time, the number of service instances will be reduced as well, in order to match the demand. We can see how dynamic and flexible the scalability with microservices is, after applying the necessary patterns.

Figure 22 depicts the current state of the ERP system. If we compare it with Figure 20, which represented the architecture before the introduction of the microservices patterns, it becomes obvious, that the complexity has increased greatly. Currently, when a request is made from a user to create a Todo List entry, it goes through the API gateway, which checks the Service Registry for available instances. If such exist, the Load Balancer will send the request to the one with least traffic on it. If no available instances exist, Kubernetes will create a new one (assuming we have not reached the predefined maximum amount) and then the request will be redirected to it. The request is then returned to the end-user via the API Gateway. The interaction between the user and the other part of the system - the monolith is changed as well. All the requests pass through this new layer - the API Gateway and then are redirected to the monolith. The way of operation within the monolith remains unchanged, except for the Tasks module, which communicates with the Todos Service via an API.

Now that we have the architecture of the ERP system ready to support more services, we move on to Phase 4, where we will continue to decompose it.

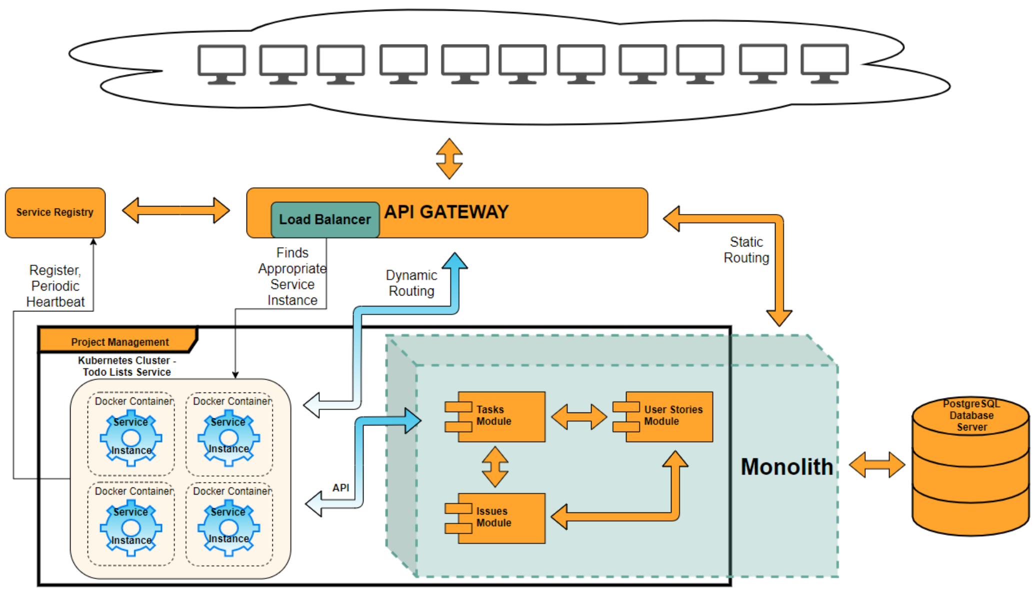

Figure 21: Todo Lists After Introduction of Microservices Patterns

Technology Stack

- **Docker** - The most popular tool for containerization.
- **Jenkins** - Setup the CI pipelines using this tool.
- **Selenium** - Integrate with Jenkins in order to create automated tests.
- **Eureka** - Service Registry for Java maintained by Netflix.
- **Hystrix** - Java library, which helps with the implementation of the Circuit Breaker pattern.
- **NGINX Plus** - Platform-and-language-independent API Gateway. Furthermore provides functionality for load balancing.
- **Kubernetes** - Tool for automatic scaling, deployment and management of containerized services.

4.2.4 Phase 4 of the Migration

Theoretical Part

At this point of the migration, the foundations have been laid for the separation of further services from the monolith. The system will be able to handle the load and the added complexity of the introduction of new services. Furthermore, the developers will be allowed to more easily setup the new services and address their dependencies. Our proposal as a next step, after the introduction of the microservices patterns is to continue decomposing the system by following the guidelines from phase two. There we have outlined the most important rules when separating a service and they should be followed in all further decompositions, not just the first. However, there are a few differences. From this point on, each new service must be put into a separate container, along with all its dependencies. Furthermore, the CI/CD must be enabled. This process will start with creating a separate repository for each new service and moving its codebase there. Then, it will continue with the creation of the pipeline, which includes automated builds and testing. This whole process of setting up the CI/DC pipeline may take some time initially, when separating the service, but it will save a lot more in the future, as the automation will speed up the deployment process. The final difference, that must be added to the actions described in phase two, is that each new service must be registered to the service registry. As mentioned earlier, this happens with a simple block of code, which will register it upon initiation and send periodical heartbeat.

The order for separation of the services follows following logic - we try to clear all the periphery subdomains in the domain, that we have started decomposing. We don't suggest to start decomposing all the peripheral modules from different parts of the system, as this will slow down the process and also the efficiency of the decomposition. It is always easier for a team of developers to focus on a particular part of the system, finish working on it and then move to another one. Usually it will take some time to grasp all the connections within a specific domain. If the team then moves on and examines a number of other domains, they will forget the specifics of the previous domains. This will lead to increased accommodation time with each domain multiple times and therefore we recommend finishing up the domain, which has been started and then starting a new one, following the same principle - *move from the periphery to the core of the domain.*

Nevertheless, there is one exception of this rule. Usually, after decomposing few services and if we see a pattern of one module repeatedly imported into multiple ones, we might consider this to be a general dependency and tackle this in a couple of ways. One way is to separate the module and create a new service, which will be able to be accessed from all the other services. By doing so we will separate this general dependency early and will define the API only once, without needing to update it with each new service. Another possible solution is to create an image with this module and put it in the container, in which the services will be placed. This will reduce the number

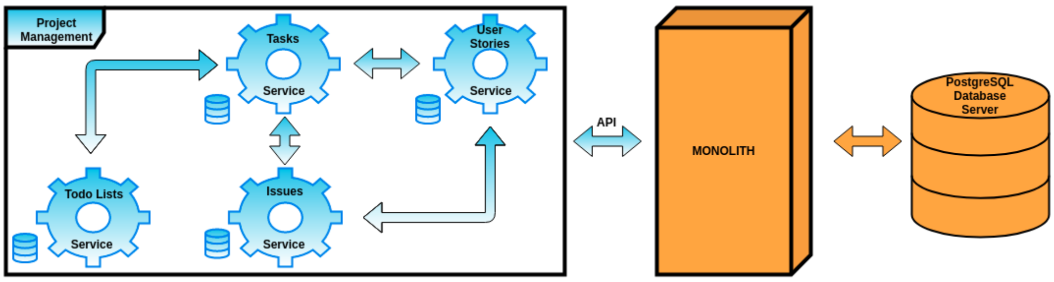

Figure 22: Project Management Domain Fully Decomposed

of API calls, but will increase the size of the containers. In this situation, as with multiple others before, we can see, that there are multiple alternative solutions to the problems. There is no single right approach for each case, but instead we should decide what are we willing to sacrifice in order to benefit from the other. Each system is different and has different requirements and goals. Therefore, we aim to provide a general approach, which can serve as a foundation for the migration from monolithic architecture to microservices, and not set strict rules and boundaries.

ERP System Example

Now, after the Todos Service has been placed in a container and all the supporting patterns are in place, we can continue with the decomposition of more modules. We check for the existence of more peripheral modules in the Project Management domain. As we can see on Figure XX, there are no such modules in our simplified version of this part of the system. Therefore, we can start moving further inside. The next module, that we want to decompose would be the Tasks module. The reason, is that it already has a simple API created, which will interact with the Todo Lists service. We start by putting it into a separate container, check for its dependencies and include them into the container as well. Then follows the introduction of the CI/CD pipeline and we move on with all the actions from Phase two. When we finish with them as well, we add the functionality, concerned with the service registry and with this, we have our second service decomposed. This time, however, the entire database and codebase decomposition process will be way longer, because tasks interact with much more modules, as compared to the todo lists. After the tasks, we repeat the process with the remaining modules in the Project Management domain - User Stories and Issues. When all the subdomains of the chosen domain are transformed to microservices, we can start working on another part of the system, again starting with the peripheral modules. The current situation can be seen on Figure 23, where we can see the different services within the Project Management Domain, which interact between each other and the Monolith with an API. Each service has its own database and is present in the service registry. The components, that are still in the monolith remain unchanged, except for the ones, that need to interact with the newly-created services. Aside from the interaction within the system, all the requests from and to the end-user are again passing through the API Gateway. The entire system architecture and the patterns (see Figure 22), which we introduced earlier remain unchanged, we just increase the number of services. Until this point, we can see some of the general dependencies, which need to be imported. For example with the Odoo ERP, this is the Odoo framework functionality. In order to deal with the import, we have chosen to create a image of it and include it into each container. The other approach mentioned earlier - to create a service with the functionality, will lead to too many API calls in this case and we have decided to increase the size of the container, but reduce the network load.

After all the Domains have been decomposed, we will no longer have a monolith and our ERP system will adhere to the microservices architecture standards. We will have small, autonomous services, which can be scaled independently and the maintenance and development process will be quite easier, as compared with the monolith.

4.3 Monitoring and Maintenance

The process of maintenance exists from the beginning of a system's life to its end. The monolithic system needed to be maintained as well, but this task is way more difficult when dealing with a distributed system, such as microservices. Even after the introduction of the first service, the monitoring should begin and various performance-related data should be logged. This monitoring process continues with even higher importance after the process of decomposition has finished and we are satisfied with the size of our services and the way our system behaves in general. This is the part, where we must continue improving the system by identifying the weak points and addressing the issues. There are tools, which can help us with this effort (e.g. Riemann [110]), that monitor the resource usage and the performance of each separate service instance. When we find some pattern of poor-performing service, we should consider a solution, which can be in many forms - we can increase the number of instances, we can rewrite the API, or even change the size of the service, which may affect the performance. Usually, when microservices architecture is adopted by a company, there should be a separate team responsible for each service [111, 112]. The size of each team varies, depending on the size and the importance of the respective service. The team is responsible for the entire lifecycle of the service - adding new functionalities, addressing existing issues and disaster recovery. This way of organization is quite efficient, because each team has high degree of understanding of the respective service, that has been assigned to them. If there are not enough people to form teams for each separate service, we suggest to assign a team at least to each domain within the system. Otherwise, if there are not enough teams even for that and, for example, we are stuck with one big team we won't benefit of the microservices architecture as much as possible.

5 Discussion and Limitations

In this chapter, we will discuss the alternative solutions, their advantages, drawbacks and will provide reasoning why we chose this approach. Furthermore, we will briefly list some of the limitations of this approach and their possible future solutions.

5.1 Evaluation of alternative solution paths

The first alternative solution, that we would like to mention is to start the decomposition with larger services and then break them down into smaller pieces. This idea is suggested at [113] and the reasoning behind it is, that the teams should be able to define the optimal level of granularity after gaining better understanding of the system. It is stated, that based on observations, a large number of monolith decompositions are actually not that efficient, because of the tiny size of the resulting microservices. Therefore, the author suggests, instead of going for too small services initially, to decompose based on the greater domain and then repeat the process, based on subdomain. During the decomposition and possibly for some period after that, the system will be monitored and information will be gathered regarding each domain. Afterwards, when enough knowledge has been gained and the teams are aware of the appropriate sizes of the services, the steps will be repeated for each domain by increasing the level of granularity and producing the microservices. In the case of the ERP system example, we would first separate all of the domains (i.e. Project Management, Accounting, HR) into bigger services, with their own databases and then repeat the process and separate new services by subdomain (i.e. Todo Lists, Tasks, Issues). The size of the microservices is one of the main things, that needs to be taken under consideration if we would like to have an efficient system. If we have too small services, we will have too many service calls and thus, slower responses from the system. A really simple operation might take long and use unnecessary network traffic, which can be prevented, if the domains and the sizes of the services are defined appropriately. This alternative approach may be good for this purpose, as it aims to gather the required information and create services of optimal size. However, by using this solution, we are almost doubling the effort, that is needed to create an efficient system. A company may consider this approach, if the system is of small size and won't take long to decompose, or it has the required work-force and resources to put aside for a long period of time. The approach, that has been proposed by us will take almost half the time compared to this one to produce an adequate microservices system, assuming the initial analysis has been prepared properly and the domains and subdomains have been identified correctly. Nevertheless, if during the monitoring phase some issues with the performance appear, the services can be further decomposed, or combined into one until they reach optimal size. However, the number of such actions will be significantly lower than if we start do the entire system decomposition twice.

Another possible approach, which we have considered is to use the Shared Database pattern [114, 17]. By adopting such a pattern, we could benefit from faster database access and database operations, when the requested information spans multiple services, and system's performance improvement in general [115]. If we use the ERP system as an example again, imagine a user makes a request to check his/hers todos. In order to get that, he will need to access the Todos service. However, as each todo is connected to a task, the system needs to create a join between the two tables and return the information. By having separate databases, we will have increased number of API calls to create this join. Instead, if using the Shared DB pattern, we will reduce the API calls and will perform the join directly. Essentially, if we decompose the monolith using a shared database, we would not need to make any changes to the database logic and to the data access layer of our services. This will significantly speed up the process of migration, as the only changes, that have to be made will be to the source code. These are some of the arguments, that support the choice of this pattern, but the majority of software developers prefers the Database per Service pattern [30]. The biggest issue with the shared database, is considered the security [116] - a service can access the data of another service, without its knowledge. This can result in performance-related spikes, because of the added queries, which is the better alternative. A far worse result, which may happen, is that one of the services changes a data type of a column, which the other service cannot process [116]. This will lead to failed operations and exceptions returned to the user. Furthermore, one of the biggest aims of the microservices is to have small, easily understandable components, which is completely in contrast with the shared database, which can be comprised of hundreds or even thousands of tables. Therefore, by weighing up all the pros and cons, we have selected the database per service pattern for our migration proposal.

A third solution, that we examined, is to introduce the microservices patterns on the first place and then carry on with the decomposition, like the authors of [6] proposed. By doing so, we are will be able to benefit from the complete functionality of the microservices architecture even before the introduction of any services. We will have the entire infrastructure ready to support the traffic and help us deal with the increased system complexity. In contrast, our approach focuses more on understanding the current system as much as possible on the first place and then continuing with the decomposition. Both of the approaches have their advantages and drawbacks. This alternative approach will save the the team responsible for the migration some time, as they won't need to go back and apply the patterns to the first separated service. Furthermore, the system will be more scalable and reliable when it comes to distributed API calls. However, we think, that there is no need to rush with the implementation of such a complex infrastructure, which won't have much to do until few services have been separated from the monolith and there is a proper distributed system present. The presence of one peripheral service, which won't have the microservices patterns available to use, won't be a problem for the performance of the entire system, as the service calls will be quite few, compared to all the possible operations. The existence of a simple API will be able to serve the purpose of inter-service communication and the user is not expected to feel the difference in operation. However, the benefits will be felt by the team, responsible for the decomposition, which will gain valuable knowledge by decomposing the first service and making test changes to the API and the database, without direct interaction with the end-user. Again, by evaluating the different options, we have decided to focus more on the improved system understandability of our approach, instead of the early infrastructure, provided by the other.

5.2 Analysis of the approach

In this section, we will provide two lists one of the limitations, which come with the implementation of our approach and one with the advantages. There are some approach-specific entries and there will be some more generic ones, which are relevant for all of the migration approaches available.

5.2.1 Advantages

- The first thing we would like to mention here is the size of the company's workforce. If there are not enough people to form the appropriate number of teams (separate team for each service) - the adoption of microservices wouldn't be nearly as effective, compared to the situation where separate teams are available for each service. One of the main aims of the microservices architecture is the precise distribution of tasks and the strictly defined boundaries between teams' expertise. By not using a separate team for each service, the teams will be involved in broader selection of tasks, which means, that they won't reach the levels of proficiency they could, if they were focused only on one service.

- A limitation of our approach is the use of peripheral services, which would work better if applied on larger software systems. If no such subdomains are identified in the system (interacting with only one other component), the selection criteria for the first service will become a bit harder. We have to identify which of the component groupings is least important to the usage of the system and decompose it first.

- Another approach-specific limitation, that has been discussed already is the need to apply the patterns after the decomposition of the first service, which would mean to go back create a container, CI/CD pipeline and rewrite some of the functions.

- Some more generic limitations are related with the time and the available hardware. The entire migration process is slow and needs to be carefully planned, with the required amount of people and resources set aside. Furthermore, the hardware of the current system will most likely be outdated and won't be able to handle the added complexity and network requests. Therefore, this needs to be addressed as well and must be calculated before starting the process.

5.2.2 Limitations

- As we have mentioned multiple times already, the biggest advantage of this approach, compared to others is the possibility to give the decomposition team the required amount of time to decompose the first service and gain better understanding of the entire system.

- By employing the database per service pattern, we increase the security and the reliability of each of the services and therefore, of the entire system.

- The remaining advantages, which come with the adoption of microservices, won't be listed here, as they were mentioned on a couple of occasions already and by correctly migrating from the monolith using this method, all the other generic advantages are applicable.

6 Conclusion and further research

In this thesis, we have proposed a step-by-step approach on how to decompose a monolithic software system and turn it into microservices. We have tried to include all of the different aspects of the process, which have to be taken into account, when such a migration is considered. First, the required background knowledge was provided. We also discussed the advantages and drawbacks of the microservices and monolith, which can help in the decision making, whether to decompose, or not. As we have already stated multiple times, the migration is not always necessary - it depends on the situation and the resources available. If, however, a decision has been taken to decompose the existing monolithic system, our approach can serve as a general guideline for decomposition. We have focused our ideas on reducing the risks for the entire system by separating the most peripheral services first and then moving forward with the critical ones. Furthermore, we wanted to give the decomposition team time to get acquainted with the system and the process. Therefore, we have decided not to make any changes to the infrastructure initially. The suggested approach can be applied to any software system in any domain. Nevertheless, in order to provide more concrete examples and improve the understandability of the thesis, we have used an example with a monolithic ERP system.

As a future research topic, we can suggest, that parts of the process become automated. For example, the code decomposition can be made automatically by defining the imported libraries, which need to be replaced with service calls. This will save a lot of manual work and the migration team will be able to focus on the more system-specific aspects (e.g., definition of domain boundaries). Nevertheless, we are satisfied with the current solution and recommend using it to create an effective and fully-functional microservices-based software system out of a monolithic one.

References

[1] Shivendra Odean. Software Architecture — The Monolithic Approach. https://medium.com/@shivendraodean/software-architecture-the-monolithic-approach-b948ded8c333.

[2] Jalal Kiswani, Sergiu Dascalu, and Frederick Harris. Cloud-ra: A reference architecture for cloud based information systems. pages 849–854, 01 2018.

[3] Brian Hayes. Cloud computing. *Communications of the ACM*, 51(7):9–11, 2008.

[4] Johannes Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.

[5] Patrick Nommensen of NGINX, Inc. It's Time to Move to a Four-Tier Application Architecture. https://www.nginx.com/blog/time-to-move-to-a-four-tier-application-architecture.

[6] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A Tamburri, and Theo Lynn. Microservices migration patterns. *Software: Practice and Experience*, 48(11):2019–2042, 2018.

[7] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. Challenges when moving from monolith to microservice architecture. In Irene Garrigós and Manuel Wimmer, editors, *Current Trends in Web Engineering*, pages 32–47, Cham, 2018. Springer International Publishing.

[8] Chris Richardson. OOP VS Procedural Programming. https://neonbrand.com/website-design/procedural-programming-vs-object-oriented-programming-a-review/.

[9] Bjarne Stroustrup. What is object-oriented programming? *IEEE software*, 5(3):10–20, 1988.

[10] James Le. NoSQL and Big Data. https://medium.com/cracking-the-data-science-interview/an-introduction-to-big-data-nosql-96b882f35e50.

[11] Rick Cattell. Scalable sql and nosql data stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

[12] Why Legacy Monolithic Architectures Won't Work For Digital Platforms. http://www.vamsitalkstech.com/?p=5617.

[13] 3-Tier Architecture: A Complete Overview. https://www.jinfonet.com/resources/bi-defined/3-tier-architecture-complete-overview/.

[14] Xue Liu, Jin Heo, and Lui Sha. Modeling 3-tiered web applications. In *13th IEEE international symposium on modeling, analysis, and simulation of computer and telecommunication systems*, pages 307–310. IEEE, 2005.

[15] Robert Annett. What is a Monolith? http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html.

[16] Ivan Montiel. Low Coupling, High Cohesion. https://medium.com/clarityhub/low-coupling-high-cohesion-3610e35ac4a6.

[17] Chris Richardson. *Microservices patterns*. Manning Publications Shelter Island, 2018.

[18] Chris Richardson. Pattern: Monolithic Architecture. https://microservices.io/patterns/monolithic.html.

[19] Microservices vs. Monolith Architecture. https://medium.com/pixelpoint/microservices-vs-monolith-architecture-c7e43455994f.

[20] Helmut Petritsch. Service-oriented architecture (soa) vs. component based architecture. *Vienna University of Technology, Vienna*, 2006.

[21] Hao He. What is service-oriented architecture. *Publicação eletrônica em*, 30:1–5, 2003.

[22] Service-Oriented Architecture (SOA) Definition. https://www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html.

[23] Service-Oriented Architecture. https://www.geeksforgeeks.org/service-oriented-architecture/.

[24] M-T Schmidt, Beth Hutchison, Peter Lambros, and Rob Phippen. The enterprise service bus: making service-oriented architecture real. *IBM Systems Journal*, 44(4):781–797, 2005.

[25] SOA - Enterprise Service Bus. https://www.tutorialspoint.com/soa/soa_enterprise_service_bus.htm.

[26] Saad Arshed. Monolithic vs SOA vs Microservices — How to Choose Your Application Architecture. `https://medium.com/@saad_66516/monolithic-vs-soa-vs-microservices-how-to-choose-your-application-architecture-1a33108d1469`.

[27] Kristijan Arsov. Microservices vs. SOA — Is There Any Difference at All? `https://medium.com/@kikchee/microservices-vs-soa-is-there-any-difference-at-all-2a1e3b66e1be`.

[28] Sam Newman. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[29] Eric Evans. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[30] Chris Richardson. Pattern: Database per service. `https://microservices.io/patterns/data/database-per-service.html`.

[31] Microservices: A new approach to building applications. `https://gautambiztalkblog.com/tag/microservices-approach-vs-traditional-approach/`.

[32] What is a CDN? `https://www.cloudflare.com/learning/cdn/what-is-a-cdn/`.

[33] Ted Schadler. Mobile Needs A Four-Tier Engagement Platform. `https://go.forrester.com/blogs/13-11-20-mobile_needs_a_four_tier_engagement_platform/`.

[34] Anton Kharenko. Monolithic vs. Microservices Architecture. `https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59`.

[35] Chris Richardson. Pattern: Microservice Architecture. `https://microservices.io/patterns/microservices.html`.

[36] Chris Richardson. Microservices Patterns. `https://microservices.io/patterns/`.

[37] Tony Mauro of NGINX, Inc. Adopting Microservices at Netflix: Lessons for Architectural Design. `https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/`.

[38] Chris Richardson. Pattern: Decompose by business capability. `https://microservices.io/patterns/decomposition/decompose-by-business-capability.html`.

[39] Chris Richardson. Pattern: Decompose by subdomain. https://microservices.io/patterns/decomposition/decompose-by-subdomain.html.

[40] Chris Richardson. Pattern: Shared Database. https://microservices.io/patterns/data/shared-database.html.

[41] Chris Richardson. Pattern: Saga. https://microservices.io/patterns/data/saga.html.

[42] Using API gateways in microservices. https://docs.microsoft.com/en-us/azure/architecture/microservices/design/gateway.

[43] Chris Richardson. Pattern: API Gateway / Backends for Frontends. https://microservices.io/patterns/apigateway.html.

[44] Chris Richardson. Pattern: Client-side service discovery. https://microservices.io/patterns/client-side-discovery.html.

[45] Chris Richardson. Pattern: Service registry. https://microservices.io/patterns/service-registry.html.

[46] Chris Richardson. Pattern: Server-side service discovery. https://microservices.io/patterns/server-side-discovery.html.

[47] Chris Richardson. Pattern: Circuit Breaker. https://microservices.io/patterns/reliability/circuit-breaker.html.

[48] Chris Richardson. Pattern: Service instance per container. https://microservices.io/patterns/deployment/service-per-container.html.

[49] Chris Richardson. Inter-Process Communication in a Microservices Architecture. https://dzone.com/articles/building-microservices-inter-process-communication-2.

[50] Kubernetes. https://kubernetes.io/.

[51] Chris Richardson. Pattern: Service Instance per VM. https://microservices.io/patterns/deployment/service-per-vm.html.

[52] Chris Richardson. Pattern: Service Instance per Host. https://microservices.io/patterns/deployment/single-service-per-host.html.

[53] Chris Richardson. Pattern: Multiple Service Instances per Host. https://microservices.io/patterns/deployment/multiple-services-per-host.html.

[54] Sanjay Nair. What is CICD — Concepts in Continuous Integration and Deployment. https://medium.com/@nirespire/what-is-cicd-concepts-in-continuous-integration-and-deployment-4fe3f6625007.

[55] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 122:14, 2006.

[56] Continuous Integration. https://www.thoughtworks.com/continuous-integration.

[57] Christian Melendez. What Is CICD? What's Important and How to Get It Right. https://stackify.com/what-is-cicd-whats-important-and-how-to-get-it-right/.

[58] Trunk-Based Development: Introduction. https://trunkbaseddevelopment.com/.

[59] Architecting for Continuous Delivery. https://www.thoughtworks.com/insights/blog/architecting-continuous-delivery.

[60] Containers 101: What are containers? https://cloud.google.com/containers/.

[61] What is virtualization? https://opensource.com/resources/virtualization.

[62] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[63] Mathijs Jeroen Scheepers. Virtualization and containerization of application infrastructure: A comparison. In *21st Twente Student Conference on IT*, volume 1, pages 1–7, 2014.

[64] Steven J. Vaughan-Nichols. Doomsday Docker security hole uncovered. https://www.zdnet.com/article/doomsday-docker-security-hole-uncovered/.

[65] What is cloud computing? A beginner's guide. https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/.

[66] What is cloud computing? https://www.salesforce.com/products/platform/best-practices/cloud-computing/.

[67] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. Recommendations of the National Institute of Standards and Technology, 2011.

[68] Cloud Computing: A complete guide. https://www.ibm.com/cloud/learn/cloud-computing.

[69] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano Merino, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. 10 2015.

[70] Sarath Pillai. Difference Between Monolithic and Microservices based Architecture. https://www.slashroot.in/difference-between-monolithic-and-microservices-based-architecture.

[71] Siraj ul Haq. Introduction to Monolithic Architecture and MicroServices Architecture. https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63.

[72] D. Taibi, V. Lenarduzzi, and C. Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, Sep. 2017.

[73] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to cloud-native architectures using microservices: an experience report. In *European Conference on Service-Oriented and Cloud Computing*, pages 201–215. Springer, 2015.

[74] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly, 2012.

[75] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. In *CLOSER*, pages 221–232, 2018.

[76] Michael Feathers. Working effectively with legacy code, first edition. pages 221–232. Prentice Hall, 2018.

[77] Gert Glatz Nhiem Lu and Dennis Peuser. Moving mountains – practical approaches for moving monolithic applications to microservices. 2019.

[78] An Introduction to Low-Code Development. `https://www.mendix.com/low-code-guide/`.

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

[80] Elisabeth J Umble, Ronald R Haft, and M Michael Umble. Enterprise resource planning: Implementation procedures and critical success factors. *European journal of operational research*, 146(2):241–257, 2003.

[81] Amal Ganesh, K.N. Shanil, Sunitha C, and A.M. Midhundas. Openerp/odoo - an open source concept to erp solution. pages 112–116, 02 2016.

[82] Arun Devkota. Open erp odoo guidebook for small and medium enterprises. 2016.

[83] Odoo ERP. `https://www.odoo.com/`.

[84] Daniel Reis. Odoo development essentials. Packt Publishing, 2015.

[85] Odoo Architecture. `https://doc.odoo.com/trunk/server/02_architecture`.

[86] Odoo Architecture Updated. `https://doc.odoo.com/7.0/book/1/1_1_Inst_Config/1_1_Inst_Config_architecture/`.

[87] Elizabeth J O'Neil. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1351–1356. ACM, 2008.

[88] Object-relational Mappers (ORMs). `https://www.fullstackpython.com/object-relational-mappers-orms.html`.

[89] G. Moss. *Working with Odoo*. Packt Publishing, 2015.

[90] Odoo Community Modules. `https://apps.odoo.com/apps`.

[91] Odoo Inheritance. `https://www.odoo.com/documentation/11.0/howtos/backend.html#inheritance`.

[92] SchemaSpy. http://schemaspy.sourceforge.net/.

[93] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot. 2015.

[94] NDepend. https://blog.ndepend.com/identify-net-code-structure-patterns-with-no-effort/.

[95] Spring Framework 5. https://spring.io/.

[96] RESTful API creation with the Spring Framework. https://spring.io/guides/gs/rest-service/.

[97] Scott W Ambler and Pramod J Sadalage. *Refactoring Databases: Evolutionary Database Design.* Pearson Education, 2006.

[98] JW Walton. Should I Containerize My Database? https://www.credera.com/blog/technology-solutions/should-i-containerize-my-database/.

[99] Docker. https://www.docker.com/.

[100] Jenkins. https://jenkins.io/.

[101] Selenium. https://www.seleniumhq.org/docs/.

[102] Netflix. Eureka. https://github.com/Netflix/eureka.

[103] Python Service Registry. https://pypi.org/project/service-registry/.

[104] Hystrix Circuit Breaker. https://cloud.spring.io/spring-cloud-netflix/multi/multi__circuit_breaker_hystrix_clients.html.

[105] Circuit Breaker for Python. https://pypi.org/project/circuitbreaker/.

[106] NGINX API Gateway. https://www.nginx.com/blog/deploying-nginx-plus-as-an-api-gateway-part-1/.

[107] High availability and disaster recovery support in Lync Server 2013. https://docs.microsoft.com/en-us/lyncserver/lync-server-2013-high-availability-and-disaster-recovery-support.

[108] Jenkins Pipeline. https://jenkins.io/doc/book/pipeline/.

[109] NGINX. Service Discoveri in a Microservices Architecture. https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/.

[110] Riemann monitors distributed systems. http://riemann.io/.

[111] Vivek Madurai. Agile Squad for building Microservices. https://medium.com/@vivekmadurai/agile-squad-for-building-microservices-54a05469ece9.

[112] Lianping Chen. Microservices: architecting for continuous delivery and devops. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 39–397. IEEE, 2018.

[113] Zhamak Dehghani. Breaking the Monolith into Microservices. https://martinfowler.com/articles/break-monolith-into-microservices.html#GoMacroFirstThenMicro.

[114] Chris Richardson. Pattern: Shared Database. https://microservices.io/patterns/data/shared-database.html.

[115] Roman Krivtsov. Is a Shared Database in Microservices Actually an Anti-pattern? https://hackernoon.com/is-shared-database-in-microservices-actually-anti-pattern-8cc2536adfe4.

[116] Oren Eini. Shared Databases in Microservices Are a Problem. https://dzone.com/articles/shared-databases-in-microservices-is-a-poblem.

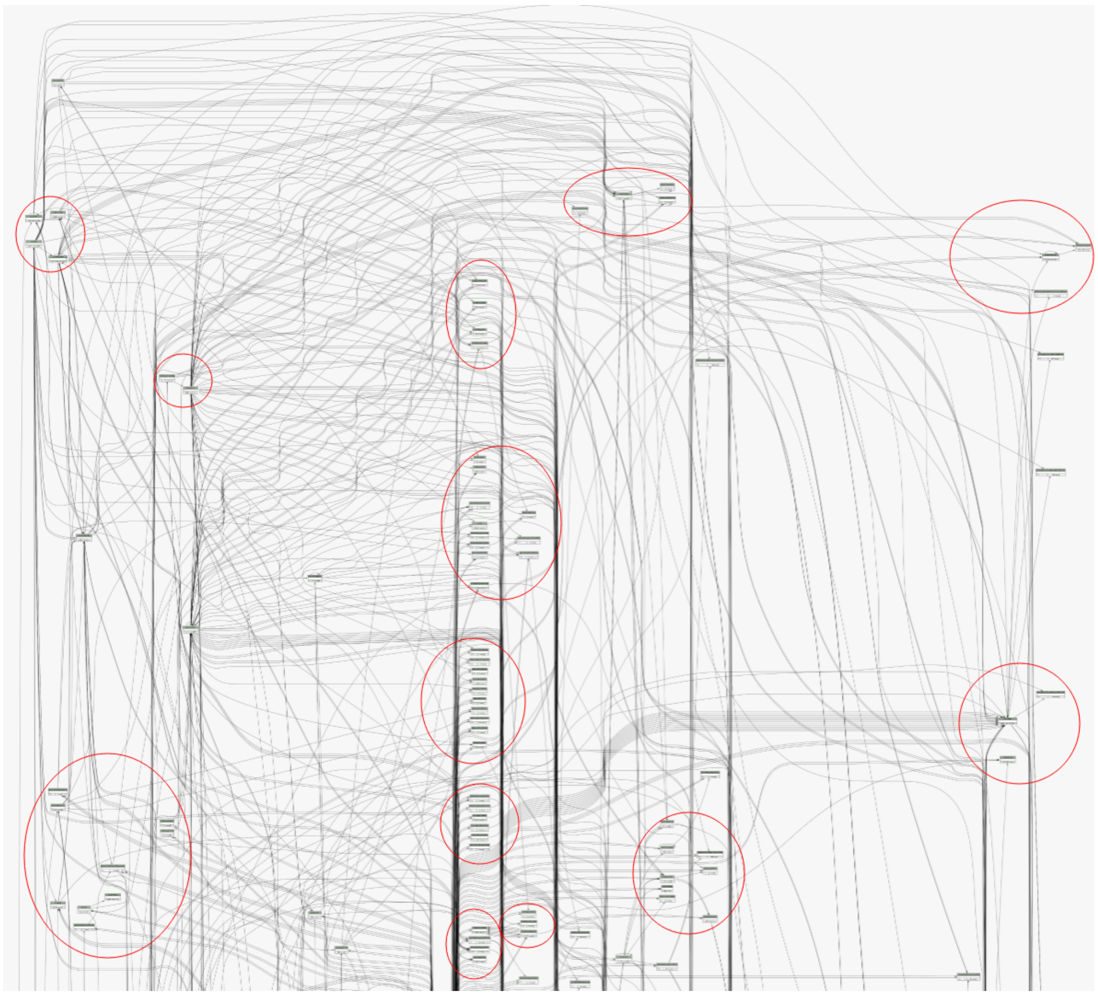

Figure 23: Part of the Database Schema of Odoo

Source: Produced by SchemaSpy