FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Assisted and Incremental Refactoring Towards a Microservice Architecture

Rita Matos Maranhão Peixoto

Master in Informatics and Computing Engineering

Supervisor: Prof. Filipe Figueiredo Correia

July 19, 2023

Assisted and Incremental Refactoring Towards a Microservice Architecture

Rita Matos Maranhão Peixoto

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Prof. Jácome Cunha
External Examiner: Prof. Florian Rademacher
Supervisor: Prof. Filipe Figueiredo Correia

July 19, 2023

Resumo

No rumo contemporâneo tem-se verificado uma crescente tendência de migrar sistemas monolíticos existentes para microserviços, em buscar de maior disponibilidade, integração e escalabilidade com tecnologias da *cloud*, desenvolvimento e manutenção modulares, facilitação da colaboração e simplificação do *deployment*. No entanto, o processo de migração de monólitos para microsserviços é feito predominantemente de forma manual, com pouco suporte de ferramentas.

A falta de orientação para os desenvolvedores realizarem uma migração e o entedimento limitado das práticas da indústria em relação à migração de microsserviços destacam a necessidade de mais pesquisas e estudos empíricos para avaliar as abordagens e técnicas propostas na literatura.

A partir da revisão da literatura, concluímos que há muitos esforços na decomposição do sistema em serviços, mas pouco no *refactoring* em larga escala necessária para implementá-lo. Além disso, há falta de ferramentas automatizadas para facilitar o processo de migração, com algumas ferramentas exigindo *input* construído manualmente. Como resultado, os desenvolvedores acham difícil aceitar essas ferramentas para executar essas tarefas complexas, principalmente devido à sua compreensão limitada do potencial impacto no sistema. Para avaliar como é realizado o processo de *refactoring* para microsserviços, as ferramentas e os métodos de avaliação utilizados atualmente na indústria, realizamos um estudo entre empresas de software e profissionais da área, com foco em como as equipas selecionam e aplicam os *refactorings*. A pesquisa revela que a indústria realiza a migração intercalada com a evolução do produto, utilizando como principal técnica de *refactoring* o *Strangler Fig*. Contudo, necessita de ferramentas que sejam fáceis de usar e que forneçam múltiplas alternativas de decomposições com alguma visualização da hipótese.

Esta dissertação tem como objetivo demonstrar a viabilidade de geração automática de sequências viáveis de *refactorings* para a migração incremental em direção a uma arquitetura de microsserviços. Ao formalizar e sistematizar o processo de decomposição de um sistema em microsserviços, propomos uma abordagem que sugere uma sequência de *refactorings* com base num catálogo derivado da literatura existente e dos resultados da pesquisa. Adicionalmente, desenhamos e implementamos uma ferramenta baseada nesta abordagem, que denominamos MicroOnion. Uma vez dada uma decomposição pretendida para uma determinada base de código, esta ferramenta recomenda uma sequência de *refactorings* que podem ser usadas, de forma segura, para realizar essa decomposição.

Fomos mais longe e desenvolvemos um estudo empírico, realizando entrevistas semiestruturadas com profissionais de software para aferir a abordagem e o protótipo construído. O estudo avaliou em que medida as sugestões da ferramenta funcionariam na decomposição desse sistema e contribuiu para determinar se elas oferecem sequências ótimas. Os entrevistados expressaram confiança nas recomendações da ferramenta e expressaram interesse em analisá-la mais profundamente em seus projetos.

Abstract

There has been an increased tendency to migrate existing monolith systems to microservices in search of higher availability, integration and scalability with cloud technologies, modular development and maintenance, facilitation of collaboration and the simplification of deployments. Nevertheless, the process of migrating monoliths to microservices is predominantly performed manually, with little tool support.

Current approaches to decomposing monoliths into microservices primarily focus on architectural guidance, overlooking the practical considerations and consequences that arise when making code-level changes. This lack of guidance for developers and a limited understanding of industry practices regarding microservice migration highlight the need for further research and empirical studies to evaluate the proposed techniques in the literature.

From the literature review, we concluded that there are a lot of efforts in decomposing the system into services but little about the large-scale refactoring needed to implement it. Additionally, there is a lack of automated tools to facilitate the migration process, with some tools requiring manual built input. As a result, developers find it challenging to accept these tools for executing these complex tasks, primarily due to their limited understanding of the potential impact on the system.

To assess how the process of refactoring to microservices is undertaken, the tools and the evaluation methods used currently in the industry, we conduct a survey among software companies and professionals from this field, focusing on how teams select and apply refactorings. The survey uncovers that the industry performs the migration interspersed with the product evolution, using as the primary refactoring technique the Strangler Fig. Besides, they need tools that are easy to use and provide multiple decompositions alternatives with some visualization of the hypothesis.

This dissertation aims to demonstrate the feasibility of automatically generating viable sequences of refactorings for incremental migration to a microservice architecture. By formalizing and systematizing the process of breaking a system into microservices, we propose an approach that suggests a sequence of refactorings based on a catalogue derived from existing literature and survey findings.

Additionally, we design and implement a tool based on this approach, which we name MicroOnion. Once given an intended decomposition for a given code base, this tool recommends a sequence of refactorings that could be used to carry out that decomposition.

Furthermore, we performed an empirical study, conducting semi-structured interviews with software professionals to evaluate the approach and evaluate the prototype developed. The study assessed to what extent the suggestions the tool gave work in the decomposition of that system and determined if they represent optimal sequences. Respondents expressed trust in the tool's recommendations and expressed interest in further analyzing it within their familiar projects.

Keywords: Monolith migration, Microservice migration, Refactoring, Software Architecture, MicroOnion

Acknowledgements

I want to begin by expressing my gratitude to my supervisor, whose guidance, expertise, support, and commitment have been instrumental throughout this last year. I truly appreciate the availability, feedback, and knowledge shared during the development of this project.

I am immensely thankful to my incredible family for your unwavering love, encouragement, patience, and support that has carried me through every stage of my life. You have been my rock, providing the foundation and means necessary for me to complete this degree. Without you, none of this would have been possible. You are the pillars of my strength.

A special thanks go to my boyfriend. Your presence and belief in me have been my constant inspiration. Thank you for encouraging me during the highs and lows throughout these years and for all the help.

To the fantastic friends I have made while pursuing this degree, you have been a source of inspiration, motivation, and joy throughout my academic journey. These years at FEUP have been filled with unforgettable memories, and I am profoundly grateful for the individual and collective support you have provided me with.

To my friends from all corners of my life who have stood by my side during these university years, thank you for your support and for being the escape I occasionally needed from reality.

Last but not least, a heartfelt thanks to everyone I've crossed paths with along this adventure—colleagues, professors, lecturers, and all the amazing staff. Your assistance, support, and knowledge sharing have shaped me into the person I am today. I'm forever grateful for your impact on my academic journey.

Rita Peixoto

"I've been blessed to find people who are smarter than I am,
and they help me to execute the vision I have"

Russell Simmons

Contents

List of Figures

List of Tables

Listings

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CD	Continuous Delivery
CI	Continuous Integration
CIO	Chief Information Officer
CQRS	Command and Query Responsibility Segregation
CTO	Chief Technology Officer
DevOps	Development and Operation
DTO	Data Transfer Object
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
QA	Quality Assurance
OOP	Object-oriented Programming
ORM	Object Relational Mapping
REST	Representational State Transfer
RPC	Remote Procedure Call
RQ	Research Question
SQL	Structured Query Language
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UX	User Experience

Chapter 1

Introduction

This chapter provides a context for this dissertation. We contextualise the topic in the first section (Section 1.1). Then, we present the motivation (Section 1.2) followed by the problem (Section 1.3). After that, we present the hypothesis of this thesis (Section 1.4), lay out the objectives of this work (Section 1.5), and describe the document structure (Section 1.6).

1.1 Context

The topic of software evolution has gained popularity over the years. However, when dealing with highly complex systems, the process of evolution becomes more challenging as companies constantly strive to adapt and keep up with the competition. Every year, the number of projects and software developers grows significantly. Companies evolve the projects to the point where their infrastructure may be unable to meet the requirements of the systems.

Moreover, it is at this point that difficult choices must be made. "How should the infrastructure of our system evolve? What are our alternatives? What efforts do we need to make?"

We have noticed an increasing tendency for practitioners to solve this by changing their architecture to accommodate the scale of their evolution; as they gain more users, their code base grows larger, and their vision becomes more complex [111]. Microservices architecture is a popular architecture pattern to adopt in these situations, considered a trend in the industry, with its numerous advantages of scalability, flexibility and higher development speed. Besides, it also enables team scalability by allowing independent development, deployment and scaling of services, fostering agility and flexibility within the organisation.

Using the cloud to assist the system's growth is a good way to support it. However, in order to make the most of it and its elasticity, systems must be created differently than what we are used to with monoliths. Microservices are an excellent solution to address this. As a result, this trend might be due to the evolution of cloud computing and the effectiveness of a microservice design in cloud computation. Another noteworthy factor supporting the teams' decision to embrace this architecture is the substantial buzz surrounding it, reinforced by compelling experience reports

from industry giants like Amazon and Netflix, who have successfully adopted this architecture to scale their systems.

Cloud computing provides high elasticity to projects; systems scale, change, and are resilient whenever needed. As appealing as it may sound, migrating legacy systems to this architectural pattern is complex, challenging and time-consuming, requiring cautious planning and execution.

1.2 Motivation

As we saw before, changing our system's architecture to this new architectural paradigm with all its promises can be very attractive. However, putting this into practice requires demanding planning.

The migration from one architecture to another is frequently observed because, although a system may eventually require such a transition, it is generally considered preferable and a well-established best practice to initially build a monolithic system with good modularity [34]. As the system's requirements evolve and the need for microservices arises, organisations often opt for migration rather than building systems directly as microservices from the beginning. As this action is prevalent, we need to pay close attention to how to facilitate it.

Migration involves breaking a tight-coupled system into small, independent, loose-coupled services that can be deployed independently. This migration may take more work and effort, depending on how the developers built the monolith. The entanglement between entities and the quality of the current code base significantly impacts the effort required to perform the migration.

Besides, to divide the previous monolith code base into services, we must establish communication protocols, plan the database partition, deployment files updated, etc. As one can imagine, this migration does not happen all at once but is instead a continuous evolution and improvement of the architecture.

There are multiple things to consider before migrating, and one important to start is whether to change to microservices or keep the monolithic architecture. Moreover, this is a very error-prone task that requires significant human effort and expertise.

1.3 Problem

Numerous attempts have been made to automate the migration to a microservice architecture in order to make this process easier for developers. However, the vast majority of them are simply focused on one of the migration tasks, such as defining service boundaries and finding microservice candidates.

We rely on the developers' skills and knowledge to manually do the code refactoring to microservices because we have no aid in implementing the modifications in the code base to apply the migration.

As previously said, implementing these refactorings is hard, and because there is little assistance in the literature, it depends on intuition. Because the project's code base is vast and

complex, it is not uncommon that migration takes longer than expected and may not always be optimum [26].

In light of this, there is a clear need to systematise this process and automate the refactoring activity, which is what we intend to investigate in this dissertation.

1.4 Hypothesis

We presented the motivation and problem guiding this dissertation; hence, we propose the following hypothesis:

> "*It is possible to automatically generate viable sequences of refactorings that assist an incremental migration towards a microservice architecture.*"

A single refactor usually does not allow us to reach the goal of that refactoring. Refactoring sequences will be much more meaningful to the objectives of the refactorings. This way, by suggesting the sequence of refactorings, we will provide a higher level of assistance.

1.5 Objectives

To address the mentioned problems, this dissertation focuses on the concept of *assisted* and *incremental* refactoring towards a microservice architecture. The main objectives of this dissertation are to systematise the process of refactoring a monolith system to a microservice architecture and automatically suggest a sequence of refactorings given the microservices boundaries, so we can aid the developers to efficiently and effectively refactor the monolithic system into microservices. This approach aims to reduce human effort and minimise the risk associated with this task while allowing developers to take advantage of microservices architecture.

We want to analyse a system and suggest small and reversible refactoring steps based on best practices and architectural patterns. Rather than attempting to complete the migration all at once, we want to refactor the system allowing continuous testing and validation incrementally. This way, we can have early feedback and understand if something was not correctly performed.

We aim to give confidence and assist developers in their migration by addressing the current State of the Art gaps. To achieve these objectives, we must first understand what the literature says about this process, exploring the existing tool, methodologies and challenges and how practitioners in the industry carry it out since this dissertation intends to assist and support the developers in this challenging task. As a result, the emphasis should be on providing assistance tailored to the industry's needs so we can facilitate the successful adoption of microservices in real-world scenarios.

1.6 Document Structure

In this chapter, Chapter 1, we introduced the topics of this dissertation, transmitting our motivation, the problem it aims to solve, our hypothesis and its objectives.

Chapter 2 establishes the context for the underlying research.

Chapter 3 presents the State of the Art of the migration from monoliths to microservices. Here we perform both a literature review and an industry survey in order to answer our research questions for the State of the Art, leading us to conclude the current research gaps of this topic.

Chapter 4 exposes the research strategy of this dissertation, defines the problem, lays out our research goals and details the proposed methodology.

In Chapter 5, we introduce a refactorings catalogue to serve both as a resource for developers to perform the migration to microservices and as a guideline to our tool development.

Chapter 6 explains the proposed approach to assisted refactoring towards a microservice architecture in detail.

In Chapter 7, we describe the implementation of our tool, MicroOnion, that assists developers in the refactoring process to a microservices architecture.

Chapter 8 is dedicated to the empirical study we perform to evaluate our methodologies and assess the perception of developers on our tool.

The final chapter, Chapter 9, summarises the contributions of this dissertation and the main conclusions of this study in response to our research questions, identifying possible future work.

Chapter 2

Background

In this chapter, we provide background on the concepts referred to throughout this dissertation to familiarise readers with the prerequisites to understand it, facilitating their reading. It goes through the core notions of the two architecture types mentioned in this thesis, the migration to microservices and the notions of software patterns and refactoring.

2.1 Monoliths

A monolithic architecture is often seen as a unified model, as everything is encapsulated into a self-contained service independent from other applications. As it is a single unit, all the code base is in the same place; any time we change this system, the whole system must be built and deployed again [16, 73]. Usually has a single database and contains all the application subdomains, so all operations are local [83].

A common practice that is being increasingly spread is the Monolith First concept. It was proposed by Martin Fowler, that says that even if we know a project is going to grow to the point we will need to migrate it to microservices, we should not start to implement a project in a microservices architecture because it is only beneficial when a system is complex [34]. Besides, creating a project using a monolithic architecture is faster to develop and easy to maintain until our project grows significantly. It also allows everything to be deployed at the same time.

2.2 Microservices

Microservices are defined as a set of loosely coupled services that can be developed, deployed and maintained independently, organised around business capabilities and owned by a small team. Each service takes the Single Responsibility Principle as its motto, having things that change together stay together, and something that changes for different reasons should be kept independent of these. This way, the complex application is divided into small independent services. Services communicate with each other through API service calls, where they usually expose their functionalities using lightweight protocols such as HTTP or messaging systems.

Microservices can be developed and deployed in multiple technologies. As they are independent of each other, the choice of technology can care only about that microservice goal because the interactions shall be technology agnostic, allowing teams to work independently on different services without affecting one another.

There is no standard for each microservice size; the decomposition's granularity depends on its goals. Usually, each microservice has its own database to ensure its loose coupling. They are very prone to implementing continuous integration and continuous deployment because they offer loose coupling and independent deployment, leading to faster iterations and scalability. Microservices worry a lot about resilience because they must not depend on the other services, and if a service fails, it should not affect the others. Regarding scaling, microservices allow for horizontal scaling only of the needed parts and not the entire system like monoliths [104].

Martin Fowler and James Lewis define the microservices architecture as a set of seven principles [62]: componentisation via services, organised around business capabilities, products not projects, smart endpoints and dumb pipes, decentralised governance, decentralised data management, infrastructure automation, design for failure and evolutionary design.

Furthermore, quite a few patterns have been described for building microservice or cloud-native systems that can provide additional guidance. Richardson's book [84] and website[1] introduces some patterns, Brown's website includes a set of patterns currently under active development [14], and other pattern languages have been published as peer-reviewed articles [8, 65, 92, 93, 91, 90, 89, 4, 3, 25].

2.3 Monoliths vs Microservices

As we have seen, monoliths and microservices are two very different architectures. However, they are both useful depending on the characteristics of the systems and the trade-offs we are willing to accept. Figure 2.1 shows an example of the architectural differences between monoliths and microservices.

Monolith Advantages

The monolith advantages are that it is simple to develop from scratch, test and deploy, which is why it is recommendable to go monolith first. This way, we can launch a project faster. Scaling is simple because it only allows horizontal scaling, where we can run multiple instances with a load balancer. It has a low overhead on the calls because they are mainly local so it can offer better performance. In terms of development, it facilitates collaboration, testing and sharing code, as everyone is working on the same code base. Lastly, it is easy to maintain in small projects [112, 104].

[1] Richardson's website is available at `https://microservices.io`.

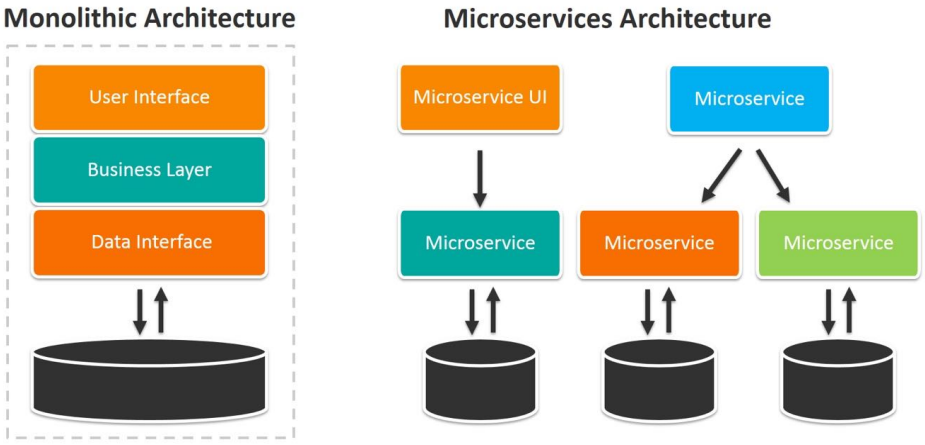


Figure 2.1: Architecture differences between traditional monolithic applications and microservices Source: What is "Microservice Architecture? Microservices Explained" [53]

Monolith Disadvantages

In terms of disadvantages, monoliths suffer a lot from maintainability when it comes to large and complex applications. In monoliths, the same technology stack and programming language must be used during the system's lifetime because changing is very costly as it means changing the entire application. In terms of scalability, it offers horizontal scalability. Still, the whole system has to be replicated, which is not ideal as some functionalities have low usage and are being replicated either way. When a change occurs, the entire system needs to be redeployed. Also, every time we change something, we must ensure we are not ruining something because of dependencies, as a simple bug in one module can shut down the entire process. When it gets too big, it might become hard to introduce new people to the project and too dependent on the people who built it [112, 104].

Microservices Advantages

Microservices enable continuous integration and continuous delivery of large complex applications. Because of this, they allow better testing and deployability. Each microservice should be relatively small so it is comfortable for all developers to understand. Teams work independently and own the services, so they can develop, test, deploy and scale their services as they wish. Due to this, they can also choose what technologies to use in their service. This way, teams become more flexible and agile [62].

Microservices are easier to scale because services can scale independently according to demand. They also do not have to worry about a failure on one service bringing down the entire system [104].

Microservices Disadvantages

Even though we might be painting it as the holy grail, it has its own drawbacks. Microservices are a distributed system which brings different challenges increasing the complexity, for example, implementing transactions, queries and more that refer to multiple services. Testing and debugging are harder because testing end-to-end touches on many services, and the same goes for finding out where the error is. Overall, microservices have an operational overhead regarding deployment, monitoring and management [104].

2.4 Software Patterns vs Software Refactoring

Software Patterns are *"descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context"* [40].

Patterns can be categorised into many different types. In this dissertation, we are more focused on microservices migration patterns and architectural patterns. The use of patterns has benefits like improving code maintainability, code reusability, and standardised communication, among others.

Fowler defined software refactoring as being *"[...] the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure"* [39, 33].

Software refactoring aims to improve code readability, maintainability, and performance while preserving the same functionalities. It is typically done in small steps and well-tested to minimise the risk of introducing bugs and breaking the functionalities. Overall it aims to improve the code quality and prevent bugs.

Software patterns and refactorings are closely related. Both contribute to better software quality, maintainability and readability but focus on different aspects. We often see software refactorings being used to evolve a system towards particular software patterns [56]to apply the software patterns. In these cases, the refactorings change the code to align with the pattern, improving the code quality and maintainability or other quality attributes.

The same happens the other way around. Software patterns can guide refactorings because they are common solutions to typical problems. This way, they provide a target for restructuring the code based on proven practices.

Both contribute to better software quality, maintainability and readability. While most refactorings provide a solution to address a particular design problem, some microservice and cloud-native patterns exist that focus on the process of migrating to microservices. However, these patterns are usually not as detailed on how to evolve the system as refactoring commonly is, as they do not go into the step-by-step process of changing the code. This distinction can be crucial, as this level of detail can allow for systematizing and automating the refactoring process.

2.5 Microservices Migration

The microservices architecture tries to overcome the issues with the monolithic architecture by decomposing it into small services that communicate with each other with lightweight mechanisms [16].

As systems become harder to maintain and understand due to becoming large and complex, we tend to see the migration to a microservices architecture. It is usually due to the fact that the disadvantages of using a monolithic architecture outweigh the advantages [16].

However, the suitability of this architecture is highly dependent on the specific requirements, constraints and needs of the systems. We have seen the pros and cons of each architecture, so organisations have to carefully analyse the benefits and trade-offs before choosing to migrate their system into a microservices architecture. Teams should consider factors like scalability needs, team expertise, and operational capabilities, among others, to see if the challenges do not outweigh the potential gains for a particular system.

Addressing migration challenges towards a microservices architecture requires careful planning and a well-defined strategy. It is essential to involve cross-functional teams, establish clear communication channels, and prioritise gradual and incremental changes to minimise disruptions during the migration process. There is not a single and perfect way to perform the migration. The developers can migrate everything at once or incrementally migrate in parts. Deciding how to decompose the system is a hard decision, but so is how to implement the decomposition and assess it [106].

If the monolith is small and has little business functionality, the migration to microservices may not suit this system. In this case, upgrading the technologies and performing code optimisations might be the best solution [106].

Conversely, migrating to a microservices architecture is a good alternative if a system is becoming hard to maintain, with changes implementation taking too long. Rewriting the entire system is a possibility, but it is an alternative that takes too much effort and time, and maybe when it is finished, a lot has changed since the beginning of it [106].

Chapter 3

From monoliths to microservices

This chapter analyses the state of the art of migration from monoliths to microservices, which provides a context for the upcoming work. The focus of this state of the art is on *Migration Automation*, *Large Scale Refactorings*, *Challenges* and *Evaluation*.

It offers an understanding of what other researchers have investigated and tried out. As a result, by being aware of what is already known, we may move forward and make the breakthroughs and discoveries necessary to accomplish the goal of our research and avoid duplicating earlier work when it is unnecessary.

Additionally, with this research, it is easier to recognise the areas that require additional research and development as well as potential roadblocks.

This chapter provides an overview of what is known about the scope of this dissertation, the refactoring of monoliths to microservices and an analysis of what areas should be explored further. It has different sections that go from the overview of the topic (Section 3.1), followed by the goals of this study (Section 3.2) and present the conclusions of the analysis, made into different sections: what the literature says (Section 3.3) and what the industry tells us (Section 3.4), in the last section (Section 3.5), we discuss the findings of state of the art.

3.1 Breaking the Monolith

To start, an important question is why organisations need to change their systems from a monolith to microservices. Taibi et al. identified that organisations have many factors that lead to this decision, and the main reasons are improving scalability, reducing the technical debt in the long run, independent development and deployment, maintainability and delegation of team responsibilities [95].

As Kuryazov et al. proposed, migrating monoliths to microservices usually revolves around analysis, extraction, refactoring and orchestration [60]. A common way to see this is by analysing the systems to identify the microservice candidates, the migration application, and the result's deployment [60]. Nevertheless, there are many different approaches possible to perform these

generic steps. One can decide to rewrite or rebuild the entire system, refactor it or keep the legacy/monolith system as it is and implement new features using a microservice architecture.

In addition to being unquestionably challenging, many studies mention that transitioning a monolith system to microservices is still primarily carried out manually [97, 67, 36]. It is due to the fact that there has yet to be identified a way to carry out this process that works for every case, as it is often performed in an unsystematic way based on intuition. Therefore, it is harder to automate and takes much time and effort to plan the migration in time and tasks.

This dissertation will focus on refactoring the monolith, what we can also call breaking the monolith. As we will see in the following sections, some works are published on conceptually breaking the monolith, but only some efforts on performing the activities that implement it.

3.2 Goals

To review the current state of the art in what concerns the migration of monolith to microservices, we take the following questions as starting points:

- **RQ1.** What is the process of breaking monolithic systems into microservices?
  - How do professionals do it?
  - What do researchers propose?
  - How do researchers and professionals evaluate the result from decomposition?
  - Which metrics and criteria are used by them to evaluate decompositions?
- **RQ2.** To what extent can this process be fully automated?
  - What tools already exist that try to automate this process?
  - How further from fully automated are we and why?

For this review, we decided to take different approaches to overview what researchers proposed and investigated and what is currently happening in the industry. To address the existing research, we conducted a literature review categorising the articles found and analysing the different scopes presented in the literature (Section 3.3). To support the analysis of how it is currently performed in the industry, we designed a survey and searched for similar surveys with published results (Section 3.4).

3.3 Literature Review

In this section, we will describe the methodology used in the literature review and analyse the literature found, divided into the main topics of this dissertation domain.

3.3.1 Methodology

Our literature review is methodologically similar to the one performed by Velepucha and Flores [106] that follows the guidelines given by Kitchenham et al. [57]: we identify the research questions and, from them, obtain the search terms, define the search system, determine the inclusion and exclusion criteria, search in the literature for the information to answer our research questions, review the articles that tackle the interesting topics, classify the papers and extract the information relevant to the search.

We pre-tested the search terms using *scoping searches* [11]. The search was conducted mainly on Google Scholar, which provides access to many relevant databases and to a vast number of papers and based it on the following search terms:

- `refactoring to microservices`
- `automated refactoring to microservices`
- `(monolith or microservices) and refactoring`
- `monolithic enterprise applications "microservice architecture"`
- `(monolith) and (service-oriented architecture )`
- `large scale refactoring`
- `refactoring`
- `architectural refactoring`

We kept focusing on books, conferences and journal papers from the retrieved articles. Then, these were filtered based on reading the title and abstract. In addition to this search, the supervisor recommended some articles, investigated some specific authors and followed references between papers. The articles recommended by the supervisor may not have appeared before because they were all from 2022, and the specific authors that were investigated also had very recent publications on this topic. The remaining articles went through an overview reading that categorised them into four interest levels. The highest interest level corresponds to the green colour and are the papers that are exactly the information we were looking for to perform our review. The second highest interest level corresponds to papers that complement some missing parts from the first level. The other two levels correspond to the papers found in our research but may potentially have some complementary information for our literature review (orange) or a really small possible input(red).

Figure 3.1 show the search strategy used for the review in a schema, where we can also see the categorisation of the articles in the four interest levels distinguished by colour, where green is the highest and red is the lowest.

There were 33 papers on the highest interest level, and these were all thoroughly analysed. The second highest interest level had 36 articles, and about half were thoroughly investigated to

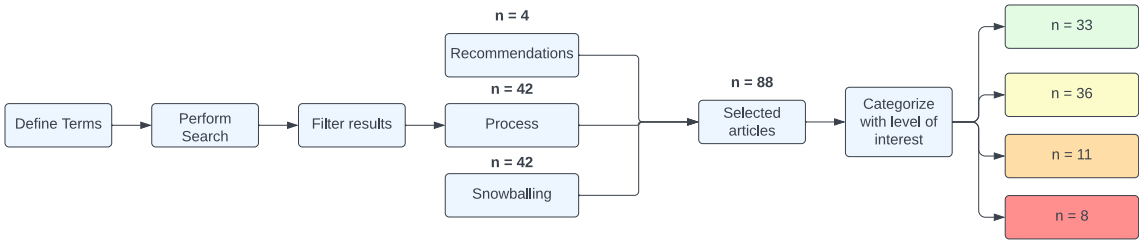


Figure 3.1: Search strategy used for the review

complement the conclusions in this chapter. The criteria used to decide which of the second highest level to analyse was to examine the ones from the domain we were missing relevant information.

In the domain of this dissertation, different issues need to be addressed and, consequently, various topics to explore. Ultimately, the goal is to suggest a sequence of refactorings to apply to the code to migrate the project from a monolith to microservices. We identified three subtopics of the suggestion of refactorings for migrating from a monolith to microservices: service decomposition, large-scale refactoring and the suggestion of a sequence of refactorings.

As previously seen, these topics were included in the search terms to give context and identify the gaps in the literature on those topics.

We divided the analysed articles into categories that address various topics regarding migrating monoliths to microservices. The categories used to separate the analysis performed were: *service decomposition*, *large scale refactoring*, *refactoring automation*, *migration patterns*, *challenges*, *automation of microservices migration* and *evaluation*. In the following sections, we analyse each category considering the automation of this process.

3.3.2 Service Decomposition

Under the concept of service decomposition, we can find different strategies with different goals. These strategies can extract services from monoliths, identify the boundaries of the services, or identify microservices candidates. The extraction of microservices is the extraction of the code that belongs to one single responsibility to create a new service, the identification of boundaries is identifying the files that represent different microservices, and the identification of microservices candidates is the identification of entities and operations that could potentially be fit to the same microservice.

According to Kuryazov et al., the migration process goes through four important steps: analysis to estimate the effort; extraction of metadata and business logic; refactoring, conversion of the monolith's source code into microservices; and orchestration, coordination of the communication between services using data and control flow [60].

We researched the approaches to decomposing a monolith system into microservices. This research focused on automation (or semi-automation) approaches that have yet to create a prototype, as we will later consider these tools. These approaches can automate a single decomposition task or the whole process.

Chen et al. proposed a dataflow-driven decomposition algorithm. They start by creating a purified DFD manually to illustrate the data flow according to the business logic. Then they combine the same operations with the same type of output data to improve the reusability of the potential microservices to be identified later. In the last step, the algorithm identifies microservices candidates by extracting *"individual modules of operation and its output data"* [16].

Al Debagy and Martinek proposed an approach that extracted operations and their parameters from an OpenAPI specification file. The focus of this approach is on the use of a vector representation using a fastText model. They use this representation to cluster similar operations to provide microservices candidates. Lastly, they evaluate the candidates using two metrics, LCOM (lack of cohesion metrics) and NOO (number of operations). The value of both metrics must be within a defined threshold for the decomposition to be presented, or the clustering process must be repeated [2].

A different approach was proposed by Mazlami et al., which used the source code and version control history to create a graph representation of the monolith. They offer three different extraction strategies based on different hypotheses. They stand that class files that change together should belong to the same microservice, what they call the *logical* coupling strategy. They also examine how much two files are related in terms of concepts or "things" present in the code, code of the same domain should be kept together, *semantic* coupling strategy. The *contributor* coupling strategy analyses team factors to cluster class files; the code of the same author should be kept together. These team factors can be team structure and communication patterns extracted from the version control systems [69]. With this approach, they were able to reduce the microservice team size to a quarter or less of the monolith's team size.

Nunes and Rito Silva proposed a decomposition based on transactional contexts assuming an architecture style of Model-View-Controller (MVC), where the controllers represent the business transactions. They assume that *"in the monolith, the execution of a controller corresponds to the transactional execution of a request"*. Therefore, a controller should be implemented by a single microservice with its database. The domain entities are extracted using static code analysis and represented in a call graph. Then clusters are generated from the dendrogram created from the call graph using a hierarchical clustering algorithm. The fact that several controllers access several domain entities makes them create a similarity metric that gives higher values to pairs of domain entities that the same controllers access so that they can be in the same cluster. An interesting fact about this approach is that they provide a visualisation step, where the architect can visualise the clusters of the entities and how the controllers access them, the accesses pattern of controllers on clusters, and the impact of domain entities data on controllers executing in other clusters. The architect can manipulate all these views, and the weights are recalculated whenever a change is made [76].

Another approach uses the use case diagram as the primary input for the decomposition. Tyszberowicz et al. proposal is based on functional decomposition. They create a system model consisting of the system's operations (public methods) and state space (system variables with information about the system's writes and reads). Then they create an operation/relation table, which

is then migrated to a graph format where the nodes are the states and operations, and the edges are the dependencies between an operation and a variable. The decomposition is performed by partitioning the state space using clustering so that each microservice has its own state space and procedures, so each microservice has high cohesion and low coupling. This clustering is based on the weights of the system model [103].

Table 3.1 compares specific characteristics of the different decomposition approaches, namely, their level of automation, type of input and output, if they need any manually built input, if the decomposition is validated and if so, how.

Table 3.1: Comparing characteristics of different decomposition approaches

Approach	**Level of Automation**	**Input**	**Manually Built Input**	**Output**	**Validation**
[16]	Semi	9 different artefacts	✓	Suggestion of services with the operations and the data structure that those operations output	
[2]	Fully	Open API specification file		Set of clustered operation that represent the microservice candidates	Metric-based
[69]	Fully	Source code and version control history		Connected component on a graph representing the microservice candidates	Case study with open source projects
[76]	Fully	Source code static analysis		Set of clusters that represent the microservices on a dendrogram	Metric based and Case study with open source projects
[103]	Semi	Use case diagram	✓	Clusters with the corresponding information about the operations and system state variables	

By analysing the table, we can see that the approaches addressed in this section have different automation levels. The semi-automated strategies present in this table need manually built inputs. Besides needing more effort from developers, it creates an even bigger disadvantage to add to the low level of automation, the lack of systematisation. The input of each approach has a lot of diversity, but we can summarise it in source code, artefacts and version control history. The

level of detail in the output also varies significantly between approaches, as some deliver only the clusters they create with the information in the representation format they used. At the same time, others also output operations and system information. In terms of validation, semi-automated approaches have no validation phase, while automated processes validate the result using metrics. In addition, some automated approaches performed case studies with open-source projects to verify the potential of their approach [51].

There are possibly other automated/semi-automated approaches to service decomposition. Still, we decided to compare the ones we found with especially interesting approaches to what input to use and how to use it. In all approaches analysed, we can point out the three main activities: extracting the architecture in some meaningful format, analysing the dependencies and then applying the decomposing strategy. The output can vary from service candidates to service boundaries, but these core steps remain the same.

3.3.3 Large Scale Refactoring

We have seen the definition of refactoring in the previous chapter. Refactoring is a recurrent task in a developer's daily work, especially in agile environments.

Large-scale refactoring, sometimes called architectural refactoring, is a refactoring that implements more substantial changes to a system and is very time-consuming for developers [48], and usually requires a high level of abstraction to understand the system. An architectural refactoring may change the internal part of the system but does not change its external behaviour [110]. Some even say that *"architecture refactorings can be considered as the first step in the quest of maintaining system quality during evolution"* [49]. While small-scale refactoring has local consequences, large-scale refactoring has global consequences, which increases the risk as there is a higher possibility of things going wrong. A large scale also needs more artefacts than just the code representation [110].

Understandably, a refactoring of this kind takes a lot of effort and multiple teams of developers. One example of large-scale refactoring is the scope of this dissertation, the refactoring of a monolithic system to a microservices architecture. The common reasons behind large-scale refactoring are reducing the time to deliver new features and versions, reducing the cost of software changes or reducing reliance on unsupported or outdated technology, as concluded by Ivers et al. In terms of technical reasons are the improvement of code understandability, migration to a new architecture and the improved use of automation [47].

The current challenges of large-scale refactoring are the "pre-conditions" for refactoring, understanding the code and the availability of tests, tools for supporting the refactoring and the non-technical concerns that worsen in scale, like team coordination and scheduling activities. In addition, it is hard to get approval for large-scale refactoring. Due to its high risk and costs, we must justify it well, demonstrating that it will lead to increased value and that it is essential [47].

Issues arise when performing refactoring in large code bases. For instance, performance issues, namespace confusion, breaking the code, introducing errors, coding style guidelines, etc. Therefore, developers appreciate when the suggestions are more visible and in-time, so when they

are doing the code, they can immediately refactor it and see how the refactoring will impact their code [41].

Some authors have been trying to identify "architecture smells" to make the parallelism with code smells [1] but for the architecture [110, 85, 88]. Many also believe that every architectural refactoring has a quality attribute goal as the focus [110, 85].

Other than surveys and articles on architectural refactoring, few works on large-scale refactoring explore the challenges of refactoring on a large scale, indicating that this topic requires more research.

3.3.4 Refactoring Automation

As seen in the previous section, refactoring has its share of complexity. Hence, several pieces of research attempted to automate it to simplify it.

We see many types of data being used to support refactoring decisions. From source code to version control history, logs, developers' insights or data retrieved from static or dynamic analysis tools that produce metrics.

A significant part of the approaches for automating refactoring use either graphs or vector-based representations to store information about the system [10].

Sometimes, to reach the goal of the refactoring, more than a single refactoring is needed, taking several chained refactorings to complete that action. For that reason, it is essential to understand how to create and predict these sequences of refactorings. Identifying refactoring opportunities is the start of this prediction. Concerning this, we also need to address if the order in which these refactorings are performed influences the quality of the result. This way, refactoring sequencing becomes an optimisation problem to try to obtain the result with the highest quality that will always depend on the quality attributes defined by the developers.

Mens and Torwé [71] identified the activities of the refactoring process as consisting of the following:

- Identify opportunities for refactoring.
- Determine which refactorings should be applied in each opportunity.
- Verify that the refactorings being used are preserving behaviour.
- Apply the refactoring.
- Analyse how the refactoring affected the program or process's quality attributes.
- Keep the refactored program code and associated software artefacts consistent.

Different tools and techniques can assist each of these activities.

[1] Code smell: *"A code smell is a surface indication that usually corresponds to a deeper problem in the system."* (Fowler, 2006) [31]

Many approaches have been published on refactoring sequencing. As mentioned before, they usually transform it into an optimisation problem for which they define the values of the metrics related to the quality attributes they want to emphasise. In terms of algorithms used in these publications, we have seen primarily search-based algorithms, for instance, greedy algorithms, A*, NSGA-III and hill-climbing [98, 18, 72, 43]. As for the quality metrics used for the objective function, we have maintainability, size of the refactoring sequence, number of modified system elements, amount of detected code smells, reusability, flexibility, understandability, functionality, extendibility, effectiveness, number of refactoring operations, design coherence, coupling between objects, the standard deviation of methods per class, etc. [70, 72, 43, 10].

Artificial intelligence can also be employed to automate the refactoring process and provide a faster and more optimal solution in many cases. However, it raises new concerns: what is considered a good refactoring candidate and how to construct good datasets to feed the algorithms. From the works analysed, we don't believe there is enough information available to allow machine learning algorithms to provide us with their benefits.

It is also recurrent that this task is interleaved with other maintenance and development tasks. In a survey conducted by Golubev et al., they concluded that many respondents use IDE features to perform their refactorings. However, they need to be more confident in how automated refactoring works [41].

There are already many tools to assist the refactoring process. Refactoring tools also have a duality of quickly identifying small refactors and having difficulty implementing large refactors. This poses a big con for developers because they believe small refactors can be implemented manually, and the interest in automated refactoring is in complex cases. If tools do not have this ability, then the learning curve of understanding these tools is not worth it [41].

The three main weaknesses of the current tools for refactoring, pointed out by Ivers et al. respondents, are usability, the ability to modify and planning what to refactor. Developers use more tools to perform large-scale refactoring than the ones that implement refactoring in code. In this study, one surprising conclusion is that respondents think testing tools would improve large-scale refactoring with much higher importance than tools that recommend specific changes, which is a critical conclusion for the scope of this dissertation. However, this study pointed out that it can be due to the need for more trust in the potential of these tools and their ability to make suitable recommendations [47].

The conclusions of these studies pose a vital aspect to keep in mind when developing an automated refactoring tool. The tool should be developed for developers, focusing on making it simple and intuitive to gain developers' trust. Often, this can be easily implemented by introducing some visualisation preview in the tool [41].

3.3.5 Migration Patterns

At the beginning of the research about migrating monoliths to microservices and refactoring, we expected to encounter good practices and some catalogued common refactorings. However, the

reality is that many good practices are expressed in the form of *migration patterns*, and we only found a few examples of catalogued *refactorings*.

Identifying migration patterns is a crucial step in refactoring microservices. To make the task of the engineers easier, various authors have pursued the goal of detecting migration patterns developers can resort to support the migration.

We can use migration patterns to plan how to lay out the migration process. The order in which we apply the patterns first depends on the requirements of each pattern because patterns can demand a specific context to be used. Therefore the choice of suitable patterns depends from project to project as it is based on the project specificities and priorities [9].

The literature contained a wide variety of migration patterns. We note that what differentiates a *migration pattern* from other microservice-related patterns is not always clear and objective, as some of these patterns are not specific for migration. They are rather patterns for constructing a system architecture based on microservices. As there is a fine line between these two situations, we make explicit that we are roughly using this term.

The migration patterns discovered in this literature revision are presented in Table 3.2. This table contains only the unique patterns, as we found many identical patterns across the literature and removed the duplicates for this table.

Table 3.2: Migration Patterns from Monoliths to Microservices

Pattern Name	**Category**	**Reference**
Enable continuous integration	Deployment	
Recover the current architecture	Migration plan's initial state	
Decompose the monolith	Decomposition, Modifiability	
Decompose the monolith based on data ownership	Decomposition, Modifiability	
Change code dependency to service call	Decomposition, Modifiability	
Introduce service registry	Scalability	
Introduce service registry client	Scalability	
Introduce internal load balancer	Scalability	[9]
Introduce external load balancer	Scalability	
Introduce circuit breaker	Fault tolerance	
Introduce configuration server	Modifiability, Deployment	
Introduce edge server	Modifiability	
Containerize the services	Deployment	
Deploy into a cluster and orchestrate containers	Deployment	
Monitor the system and provide feedback	Modifiability, Deployment	
Domain-driven decomposition	Boundary Definition	
Vertical Slicing	Boundary Definition	
Data ownership decomposition	Boundary Definition	
Code dependency to service call	Boundary Definition	

Continued on next page

Table 3.2 – continued from previous page

Pattern Name	**Category**	**Reference**
Greenfield Pattern	Functionality decomposition	
Common functionality library pattern	Functionality decomposition	[112]
CQRS Pattern	Database decomposition	
Recover current architecture	Auxiliary	
Service registry	Auxiliary	
Anti-corruption Layer	Auxiliary	
API Gateway	Auxiliary	
Tolerant reader	Auxiliary	
Cluster	Auxiliary	
Strangler Fig Application	Splitting the monolith	
UI Composition	Splitting the monolith	
Branch by Abstraction	Splitting the monolith	
Parallel Run	Splitting the monolith	
Decorating Collaborator	Splitting the monolith	
Change Data Capture	Splitting the monolith	
The Shared Database	Database decomposition	
Database view	Database decomposition	
Database Wrapping service	Database decomposition	[75]
Database-as-a-Service Interface	Database decomposition	
Aggregate Exposing the Monolith	Database decomposition	
Change Data Ownership	Database decomposition	
Synchronize Data in Application	Database decomposition	
Tracer Writer	Database decomposition	
Split Table	Database decomposition	
Move Foreign-Key Relationship to code	Database decomposition	
SAGA Pattern	-	[5]
The Client-side Discovery Pattern	Orchestration and Coordination	
The Server-side Discovery Pattern	Orchestration and Coordination	
The Hybrid Pattern	Orchestration and Coordination	[96]
The Multiple Service per Host Pattern	Deployment Strategies and Patterns	
The Database-per-Service Pattern	Data Storage Patterns	
The Database Cluster Pattern	Data Storage Patterns	
Gateway Routing	-	
External Configuration Store	-	
Static Content Hosting	-	
Pipes and Filters	-	
Command Query Responsibility Segregation	-	

Continued on next page

Table 3.2 – continued from previous page

Pattern Name	Category	Reference
Gateway Offloading	-	[105]
Backends for frontends	-	
Compute resource consolidation	-	
Sidecar	-	
Anti-corruption Layer	-	
Ambassador	-	
Gateway Aggregation	-	
Leader Election	-	
Data Transfer Object	-	
Move Foreign-key relationship to code	-	[36]
Database Wrapping Code	-	

From the patterns present in Table 3.2, some appear more frequently in the literature; these are Circuit Breaker, API Gateway, Strangler Fig Application, Move Foreign-key relationship to code, Split table, Database view, Database wrapping service and The shared database[2].

Some of the migration patterns discovered in this literature review were used in the context of the Industry Survey that we describe in the next chapter. For that purpose, we characterised them using a specification template. These descriptions are available in Appendix A.

3.3.6 Challenges

Some examined papers outline the current challenges of moving from monolith to microservice architecture. Since we want to propose an approach to perform the refactoring stage, we must be aware of this domain's current challenges. There are still no universally satisfactory approaches to address these challenges because they depend greatly on the project circumstances. The challenges can be split into *technical* and *organisational* challenges.

The technical challenges most present in the literature are:

- **Identifying services and defining their granularity**, also called splitting the monolith. This is usually referred to as one of the most complex challenges because it can significantly impact the quality of the result of the decomposition. This is a big decision because if it becomes too fine-grained, it can lead to performance losses from the communication overhead [55, 106, 95].

[2]Circuit breaker, Strangler Fig Application, Split table, Database view, Database wrapping service and Shared database are also present in [112]. Additionally, API Gateway is also present in [96], Strangler Fig Application is also present in [105], Move Foreign-key relationship to code is also present in [36], Shared database is also present in [112, 96].

- **Interaction between microservices**. Related to the last challenge of identifying the services is defining the interaction between microservices. This challenge can be worsened if the evolution mechanisms aren't well thought out, leading to redeploying multiple services due to changes in one service, which should be avoided in a microservice architecture. This concept is also known as evolutionary design. This interaction should not be restricted to technology to keep teams' independence [55, 62].

- **Finding the right tools and practices to support migration**. Most developers are unaware of the more complex tools being developed and successfully tested and mainly use static analysis tools. Besides knowing about these tools, it takes a lot of work to train developers to use them correctly [106].

- **Building a plan and deciding the priorities of the migration**. It is not always easy to build a plan that is effectively going to lead the migration, mainly because, most of the time, some features are more important than others, so they rise in priority when migrating [106].

- **Implementing new functionalities**. It is not advisable to do it in the monolith, but to do it in the microservice architecture during the architecture development takes more time [55].

- **Database management**. We need to maintain data consistency because data can be spread across multiple services. Therefore, decisions regarding this pose a complicated challenge as the database migration and data splitting depend on the system's requirements and ambitions [55, 106].

- **Security**. Ensuring communication between microservices is demanding because of the numerous potential vulnerability points. Also, as most microservice architectures assume they can trust other components, there is a higher risk of attack propagation. In addition to that, when there is data sharding, the provenance of data also poses a security threat [29].

- **Testing**. Because it may take several services to complete an action. It becomes more manageable to test each service separately, and, in contrast, it becomes harder to test the interactions between them when we cross the barrier of the individual service. Additionally, a good automatic test coverage facilitates the refactoring to ensure no new bugs are being introduced with the refactoring performed [74, 55].

- **Continuous integration and continuous delivery practices**. When developing microservices, there is a need to have these tasks automated because it becomes unfeasible to do this manually with the imposed demand by short development cycles and small releases. Deployment is more sophisticated and complex due to the fact we are deploying multiple services at different times. Nevertheless, these tasks are crucial as they enable a fast, consistent, safe way to deliver software [55].

- **Manage infrastructure**. The way the infrastructure is managed has a big role in the robustness of the microservices because having more services increases the points of failure,

network partitions and service outages, and they must be handled without the user being aware of it [55].

- **Adequate logging and monitoring**. This is crucial with this new complex infrastructure, and it poses a challenge because it becomes harder when having multiple services to monitor instead of one plus the several interactions between them, which also complicates debugging [55].
- **Lack of systematization of the process**. There isn't still a detailed systematisation of this process that allows us to try and propose ways of automating it [67].
- **Reliance on developers knowledge**. As the migration is mainly performed manually, it leaves the decisions for the developers, who have to make them based on their intuition and knowledge [36, 79].

Organisations may need to change to take full advantage of the microservice architecture. That is why we often see Conway's Law by Melvin Conway associated with microservices. It states the following:

> *"Any organisation that designs a system (defined broadly) will produce a design whose structure is a copy of the organisation's communication structure."* [21]

Therefore, the big challenge is to adapt the teams' structure to fit the needs of the new architecture and make the most of its potential. The effort required for this adaptation depends on the previous structure of the organisation, so it can go through a tough learning curve if the organisation isn't aligned with the software architecture being developed.

In terms of organisational challenges, we rely on the work of Kalske et al., who identified the following [55]:

- **Independence**. Teams are responsible for their service, have ownership over the code and work independently. It becomes easier to achieve independence when aiming for single responsibility for every service.
- **Communication**. Even though teams are less dependent on other teams, requires communication to be more effective between teams to establish the interaction between the services.
- **Teams' composition**. The composition of the teams will begin to incorporate professionals of different scopes besides development, like QA and DevOps, so that the team can achieve the independence mentioned even in testing and deployment. It affects mainly the developers as the development is closer to production than before.
- **Team Coordination**. Multiple teams working on different matters become harder to manage and coordinate.
- **Learning curve**. Inevitably, not all members of the teams will be familiar with microservices, and both development and operational teams will go through a learning curve.

When trying to bring a new approach to refactoring a monolith to microservices, we need to be aware of the presented challenges to try and mitigate them.

3.3.7 Automation of Microservices Migration

"Automation is a term for technology applications where human input is minimised" (IBM) [45]. It is also known as *"the process of creating software and systems to replace repeatable processes and reduce manual intervention"* (VMware) [107].

The trend of migrating systems to microservices usually happens when they start to get too large and complex [34]. With it happening recurrently in the industry [2], we realise that the task is complex and demanding. Hence, the need to systematise this process arises, reducing reliance on the developer's intuition. This way, efforts can be made to automate this task, thereby making the developer's job easier. In the process of migrating a system to microservices, we can identify the common activities performed: data gathering, model building, boundaries identification, refactoring and assessment of the result [60]. They usually occur in the order we can see in Figure 3.2, even though some approaches may go back and forward in these activities.

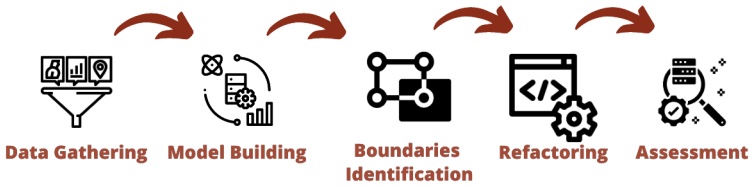


Figure 3.2: Migration activities

The data gathering corresponds to gathering information about the system, and the model building creates an internal representation of the system. The boundaries identification corresponds to defining which files belong to which proposed microservice and what these microservices' boundaries are. The refactoring is the code transformation to accommodate this partition in the identified microservices, but it includes making all these microservices independent and following the microservices principles. Lastly, it is usual to access the migration result according to the reasons that lead to the migration (this can be performance, for instance).

We looked at the tools available for migrating from monolith to microservices and discovered that the number of tools that work on automating this process is very limited. We considered tools, the publications that implemented a prototype to validate their proposed approach. Twelve tools that aided in migrating legacy systems to microservices were discovered while researching the literature.

The tools discovered are described according to their action domain in the following list:

1. Log2Ms: *"Tool which supports automatically microservices identification and MSA models generation,[...] using execution logs only"* (Liu et al., 2022) [64]

2. MonoBreaker: *"Tool for refactoring systems to microservice architectures that uses static analysis to determine the system's structure and dynamic analysis to understand its actual*

behavior. [...] MonoBreaker provides an overview of the decomposition.[...] These can be used by the developers to guide the refactoring process." (Matias et.al, 2020) [67]

3. Mono2Micro by Kalia et al.: *"AI-based toolchain that provides recommendations for decomposing legacy web applications into microservice partitions"* (Kalia et al., 2020) [54]

4. Mono2Micro by Rito Silva et al.: *"[...] software system that implements tools to support software architects on the task of migrating codebases from a monolithic to a microservices architecture"* (António Rito Silva and José Correia, 2022) [22]

5. FoSci: *"Framework that identifies service candidates from a monolithic system"* (Jin et al., 2021) [52]

6. Service Cutter: *"[...] approach to service decomposition based on 16 coupling criteria distilled from the literature and industry experience."* (Gysel et al., 2016) [42] through graph cutting.

7. GranMicro: *"Black box based approach to support the decision of service granularity"* (Mustafa et al., 2017) [77]

8. Pangaea: *"Semi-automatic tool to decompose a software system into microservices"* (Staffa et al. 2021) [94]

9. Measurement and Decomposition: *"A Microservice measurement framework to objectively evaluate and compare the quality of microservices-based system. A decomposition system based on business process mining"* (Taibi and Systä, 2020) [97]

10. AMI: *"Automated microservices identification approach that extracts microservices from execution and performance logs without providing documentation, models or source codes, while taking both functional and non-functional metrics into consideration"* (Zhang et al., 2020) [109]

11. Unsupervised learning: *"Based on a black-box approach that uses the application access logs and an unsupervised machine-learning method to auto-decompose the application into microservices mapped to URL partitions having similar performance and resource requirements."* (Abdullah et al., 2019) [1]

12. MicroRefact: *"Receives the Java code and a proposition of microservices and refactors the original classes to make each microservice independent.[...] Creates an API for each method call to classes that are in other services. The database entities are also refactored to be included in the corresponding service."*(Freitas et al., 2021) [37, 36]

13. MonoSplitter: *"A tool capable of decomposing Java projects"*. (Jesus, 2021) [51]

Despite having found twelve tools, only one performs the refactoring, and none supports the entire migration process. Table 3.3 shows which of the migration activities

Table 3.3: Migration activities performed by each tool

Tool	Data Gathering	Model Building	Boundaries Identification	Refactoring	Assessment
Log2Ms	●	●	●	○	○
MonoBreaker	●	●	●	○	○
Mono2Micro by Kalia et al.	●	●	●	○	○
Mono2Micro by Rito Silva et al.	●	●	●	○	○
FoSci	●	○	●	○	●
Service Cutter	●	●	●	○	○
GranMicro	●	●	●	○	○
Pangaea	●	●	●	○	○
Measurement and Decomposition	●	○	●	○	●
AMI	●	●	●	○	○
Unsupervised learning	●	○	●	○	○
MicroRefact	○	○	○	●	○
MonoSplitter	●	●	●	○	○

The majority of the proposed tools in the literature only work on one aspect of the process: service identification and decomposition. Although, as previously stated, they are associated with a complex and important task, they only represent one stage of the process. It lacks the actions on the code required to convert the project to microservices, forcing this task to be completed manually and with the developer's knowledge, acting only as a guide. Furthermore, different tools provide the same functionality, indicating no agreement on which should be widely accepted, as most claim to perform the best. As a result, the developers' challenge of selecting the appropriate tool remains unsolved.

Additionally, we found another tool that expected a definition of the microservices modelled in Jolie and generated the Jolie microservice files. This approach has *"[...] a set of generators that make the migration approach automatic and with less manual intervention by developers"* [15].

Aside from this type of tool, we discovered one other that worked on automating this process. MicroRefact, proposed by Freitas et al., receives the code and the microservices decomposition and implements the refactoring of the code into the microservices.

Automated tools can use different types of information as input about the system. This information can be extracted from the system or manually built by the system experts. It is trivial that the level of effort required to use the tool is important when choosing what tool to use. Some common types of information are execution logs (for example, calls of methods, information returned in the requests, etc.), the source code, artefacts (architectural model, use cases, systems model, etc.) and access logs (for instance, web access logs), and version control history (information from the GitHub repository). From the ones mentioned, artefacts are usually required to

be manually built, and even though there are already tools available to do this, it is only partially automated. The different types of information also lead to system views; therefore, the approaches are also diverse depending on the information they are fed.

Table 3.4 shows which type of information these tools use to perform their work.

Table 3.4: Type of information used by each tool

Tool	Execution Logs	Source code	Artifacts	Access Logs	Manually Built Input
Log2Ms	●	○	●	○	
MonoBreaker	○	●	●	●	
Mono2Micro by Kalia et al.	○	●	○	○	
Mono2Micro by Rito Silva et al.	○	●	○	○	
FoSci	●	●	○	○	
Service Cutter	○	○	●	○	✓
GranMicro	○	○	○	●	
Pangaea	○	○	●	○	✓
Measurement and Decomposition	●	○	○	○	
AMI	●	○	○	●	
Unsupervised learning	○	○	○	●	
MicroRefact	○	●	●	○	
MonoSplitter	○	●	○	○	

Log2Ms uses execution logs as input that are extracted using Kieker, an open-source tool requiring no manually built input. They extract the functional units and create a matrix of relations between classes. Then they perform a clustering algorithm (based on NSGA II) to identify abstract microservices. Then it automatically generates the models of the microservice. Its output is Papyrus files with the microservices models [64].

MonoBreaker's input is the project's directory, which is used to perform static analysis of the source code and dynamic analysis of the system to identify the structure of the project and the strength of the dependencies. Therefore, no manually built input is necessary. It creates a weighted graph from the analysis performed and creates clusters. It outputs a list of a set of lists, each list representing the classes a potential service will need [67].

Mono2Micro by Kalia et al. uses as input the source code, so no manually built input is required. It generates partitions with a temporospatial clustering on execution traces and clusters them using similarity. If two partitions have an association relation, these are merged [54].

Mono2Micro by Rito Silva et al. takes as input the source code and performs static and dynamic analysis that generates a call graph; no manually built input is needed. Then it clusters domain entities accessed by the same functionalities, and the architect manually performs the cut. It provides a way of visualising complexity metrics and experimenting with the clusters [22].

FoSci takes as input the source code and extracts the execution logs. It identifies the entities in the execution traces and creates groups of functional atoms for the class entities representing

each service candidate. The output is a list where each element is a tuple with the class entities, the interface classes and the operations [52].

Service Cutter requires all information to be extracted by the user. It will extract the coupling criteria from the artefacts imported by the user and let them prioritise the criteria. The output is the clusters in a graphical way that represent the entities of each service as well as the dependencies between services. It is possible to redo the prioritisation process of the coupling criteria and algorithm parameters if the suggestion does not meet the user needs [42].

GranMicro extracts the weblogs from the server using the workload pattern and clusters them to generate peaks. Then classifies web page categories by loads, and the ones with the highest loaded pages can be assigned the highest priority for splitting when splitting the monolith in microservices. The output of GranMicro is the service model diagram. To validate their approach, they performed load tests on a microservice design without GranMicro and a microservice design applying GranMicro [77].

Pangaea receives as input a system model, which defines the data entities and operations of the application, which needs to be manually built and a set of input parameters. Given this, it translates the model into an optimisation problem and provides the solution and a visual representation of the decomposition. Developers evaluate to accept or reject the solution. The output is a possible allocation of the data entities and operations onto microservices [94].

Measurement and Decomposition extract the execution logs of the system, so it requires no manually built input. This tool is semiautomated as it uses an algorithm to identify cycles on the call paths extracted from the logs and needs engineers to decide where and how to break the dependencies. So there is no clear identification of the output format. In the end, engineers assess the quality of the decomposition with the measurement framework also proposed in this work that uses four metrics, coupling, number of classes per suggested service, number of duplicated classes and frequency of external calls [97].

AMI uses the monolith system's executable as input, so it does not require manually built input. It extracts logs in the form of MessageRecord (object level) and uses this to distinguish Controller Objects and Subordinate objects. It applies the decomposition by functional behaviour, CPU and memory consumption. After, it creates a matrix for object metrics to evaluate the relationships between them. At the same time, it also gets the performance logs in the form of PerformanceRecord (class level) and creates a matrix for class metrics. With this, it performs the partition in microservices using a genetic algorithm. The output is the set of microservice candidates in the form of controller classes and subordinate classes that belong to it [109].

The unsupervised learning approach extracts the access logs from the web applications, requiring no manual input. They base their approach on clustering URIs, as it represents partitions with similar resource consumption characteristics. The output of this tool is a set of microservices with the corresponding URI of the resources belonging to that microservice [1]. This approach also automatically deploys the microservices but does not give insights into how the refactoring is performed.

MicroRefact receives the source code and a set of microservices, each described by the classes composing it. As it needs to be specified how this set is created, it can either be manually built or not, but they propose to use a tool for that and present one possible. It uses the structural information and the microservices proposal to identify the dependencies between them. Then, with the structural information and the dependencies, it identifies the entities and relationships between entities to refactor those relationships in the database. Lastly, with all the information gathered, it analyses *"the class variables and the dependencies between classes, to identify and refactor the classes that have dependencies with classes belonging to different microservices"* [36]. However, it has its limitations, for instance, the assumptions made that already receive a JSON file with a good list of the microservices and their composition and the restriction to object-oriented language Java and the framework Spring and the fact that the use a very limited number of refactoring options, providing no way of visualising the actions nor the system's evolution [36].

MonoSplitter receives a jar file with the system's code as input and a link to the GitHub repository of the project if the user considers it a reliable source and, with that performs a static analysis creating a representation of the system. After they extract all dependencies and consequent information by applying an algorithm, they define. Then the user defines some systems' properties for the algorithm to consider. The system's model is then created. The user defines the number of clusters, applying the decomposition algorithm. They present the output both in a textual and visual form [51].

It is important to add that from the tools present in this section, the ones we found that the source code is freely accessible are MonoBreaker, both Mono2Micro, Service Cutter and MicroRefact. As far as we could find, only FoSci, Mono2Micro by Rito Silva et al. and Service Cutter have something similar to replication packages.

From all the tools analysed, MicroRefact is the only one similar to what we are trying to achieve, so we will use it as a good example to iterate upon. However, we should keep in mind its limitations and work on them.

After analysing the previous works, the conclusion is that there is an increasing effort in trying to automate this process and design tools to assist it. Nevertheless, most of these approaches are oriented by the developer's experience and work under specific circumstances and assumptions [15].

Besides that, some tools still require manually built input, which is not ideal when the goal is to reduce the effort required by engineers in performing these tasks. In addition, it is also introducing subjectivity, which can reduce the potential of the results and make the results of this approach less reproducible. The quality of the output is another concern. As the level of detail varies a lot from tool to tool, a tool that only performs the service decomposition should support the next steps more and give more meaningful information besides the classes or files each microservice should contain.

Furthermore, as was concluded in the study performed by Ivers et al. in 2022, there need to be more tools for large-scale refactoring [48], which was already a suspicion that we had when

starting this research. These tools must also be easy to use to facilitate their introduction to the migration process as most professionals are not sufficiently educated on this type of tools [41].

The tool support falls short of the current needs of the industry and lacks continuous iterations to answer the migration's automation prevailing requirements. Despite its limitations, we conclude that only one tool works on refactoring from monoliths to microservices. However, there are not enough tools trying to automate the process of changing the architecture from a monolith to microservices.

3.3.8 Evaluation

Every migration must include an evaluation of the resulting architecture. The architectural goals that motivated the migration will always determine how the resulting decomposition is evaluated. However, achieving these goals should consider the preservation of the system's important quality attributes[3]. Although this evaluation can refer to the final outcome, it is not required to do so. If we are talking about a complex migration, this should be a recurring step to ensure the right path is taken and nothing is ruined in the meantime.

As previously stated, boundary identification is a migration activity. Many studies that work on automating this task use quality metrics[4] to validate the decomposition result; if it does not comply with the designated values for this metric, the process must be repeated.

In *Software Architecture Metrics*, Ciceri et al. offer generic guidance on how to apply measurements in the projects [19]. This guidance includes starting small and early, measuring meaningful things, using the conclusions of the measurements to improve the system, making the measurements visible so everyone is aware of the results and making it continuous in the development cycle.

Despite this, many practitioners do not use metrics in the development of their systems. According to Ciceri et al., the main reasons for this are the lack of knowledge and understanding of the metrics, the tools to gather the metrics, how to make metric rules (rules for the range of values the metrics should be to pass quality tests) that do not annoy developers, the need for automation so these metric-based rules are more useful and some more [19, 30].

Vale et al. found that their survey participants do not measure the software qualities directly and are not motivated to discuss concrete metrics for evaluation. They look for metrics that are simple to check and have easy procedures to check it [105].

In the articles that propose new decomposition approaches, microservices are mostly evaluated through a case study. In a good evaluation case, these are compared with other controlled systems whose capabilities are known. The quality metrics used vary much from the study's goal but were

[3] Quality attributes: *"[...] measurable and testable properties of a system.[...] Quality attributes and how they satisfy the stakeholders of the system are critical, and software architecture plays a large role in ensuring that quality attributes are satisfied"*(Joseph Ingeno, 2018) [46].

[4] Quality metrics: *"[...] measure of software characteristics that are quantifiable or countable"* (Alexandra Altvater, 2017) [6].

seen metrics like performance, efficiency, complexity, similarity, average team size across microservices, average domain redundancy, total cost, number of classes, number of microservices, independence of functionality, modularity and independence of evolvability [69, 9, 7, 94, 36, 52].

Cojocaru et al. selected the minimum set of quality attributes, choosing granularity, cohesion, coupling, scalability, response time, security, health management, execution cost and reusability. They performed a validation interviewing an expert that mostly matched their approach [20].

There are also tools to support the evaluation of the microservices. For instance, a tool-supported evaluation method for microservices was proposed by Engel et al., where they defined the principles for microservice architecture and the corresponding metrics to evaluate them [27]. Nonetheless, there is no evidence of practitioners actively using such tools.

Automated testing tools like Pact, Selenium and WireMock are commonly mentioned. It is also stated that Sonar Qube is usually integrated into the CI/CD pipeline [12].

We came to a similar conclusion from the previous topic; automation is lacking in evaluation methods [27].

Software quality was reported as being measured using mainly well-known terms like scalability, performance and maintainability [105]. Each of these qualities is measured in its own way. For example, we can measure performance according to latency, the time it takes to complete a task or throughput, the number of functions completed in a particular time, or both. Also widespread in a microservices architecture is evaluating three specific criteria: cohesion, coupling and granularity. These are also evaluated in different forms, for instance, evaluating cohesion through the number of synchronous and asynchronous dependencies, evaluating coupling through the number of consumed services, the number of consumers and the number of pairwise dependencies, and evaluating granularity through the number of exposed interface operations [27].

3.4 Industry Survey

Our literature review reveals few research works delving into professional's perspectives. However, to support systematising and providing tools for conducting the migration to microservices, we believe it is crucial to understand how practitioners are doing such migrations, what tools they use, and why.

To address this, similarly to the literature review conducted in the preceding section, we performed a targeted search for industry surveys related to this concern and a survey-based empirical study to gather the perspective of professionals.

3.4.1 Related Work

Some similar works assessed how the industry perceives and works in the migration to microservices. We limited our search to primary studies surveying techniques and patterns for the migration of monoliths to microservices. Table 3.5 summarises the intention of the analysed works.

Balalaie et al. used a qualitative empirical research to assess the validity of the migration patterns they were proposing on three migration projects of the industry [9]. Similarly, in our

Table 3.5: Related work on empirical research on the migration to microservices

Authors	**Type**	**Object of study**	**Sample Size**	**Type of participants**
Balalaie et al.	Qualitative	Migration Patterns	3	Projects
Di Francesco et al.	Survey	Activities and challenges of the migration	18	Practitioners
Taibi et al.	Survey	Processes, motivations and issues for migrating	21	Practitioners
Fritzch et al.	Interview	Intention, Strategies and challenges in the context of migration	16	Practitioners
Vale et al.	Interview	Adopted software patterns and perceived architectural trade-offs	9	Practitioners

survey, we will assess the importance practitioners give to the migration patterns found in the literature. Nevertheless, they do not assess the strategies, challenges and tools used during the migration process and we intend to.

Di Francesco et al. conducted an empirical study on migration to microservices practices in the industry with 18 practitioners. Their target group was composed of practitioners involved in the migration process of their application, and their goal was to characterise the activities and challenges of the migration [24]. We want to evaluate if we can reach some of the same conclusions as Di Francesco et al., as well as understand the kind of assistance those professionals use and how they evaluate the resulting decomposition.

Taibi et al. analysed migration's motivations, issues and processes through interviews with 21 experienced practitioners [95]. Besides taking a second look at some of the same concerns already addressed in this study, like monolith decoupling, database migration and data splitting and communication among services being migration issues or the maintainability, scalability and ROI being the most important benefits, we want to understand how practitioners perform and evaluate the migration.

Fritzch et al. conducted a qualitative study on intentions, strategies and challenges in the context of the migration to microservices [38].

Vale et al. analysed the perception of the practitioners on the impact of 14 patterns (including the *Strangler Fig*) on seven quality attributes. They explored the knowledge and adoption of patterns, their trade-offs, and what metrics professionals use to measure quality attributes [105].

To conclude, although there is already some empirical research on migration to microservices, they assess the impact of the migration patterns, some activities/processes/strategies and migration challenges. However, they do not assess the perception of practitioners of the tools to assist the decomposition, how they evaluate the decomposition and the migration patterns used. They are

more interested in understanding the necessity of migrating and its motivations rather than fully understanding the migration process. There are more to it than simply strategies used, like on what

We will go through challenges, strategies and intentions to assess if they remain the same as the previous works state, but we want to go beyond that and understand the entire migration process. We want to understand how they decide to break the monolith into microservices, what information they use to sustain their decision, where they look for guidance to support them during the migration, how they plan the migration, what techniques they commonly use, what tools they use and how do they evaluate the resulting decomposition.

3.4.2 Research Questions

The previous works analysed focus on specific parts of the migration, like challenges, intentions, strategies, and patterns. Whilst with this survey, we aim to understand how practitioners perform the migration and everything around it. This goal is reflected in the research questions of this survey, which are the following:

- **RQ1.** What is the refactoring process that they follow?
- **RQ2.** What tools do they use?
- **RQ3.** How do they evaluate the result from the decomposition?

By answering these questions, we can obtain the industry's perspective on how the migration to microservices is currently carried out.

3.4.3 Methodology

A survey was designed and conducted to characterise how practitioners perform migrations. Using a survey was a way to achieve a good sample diversity as it is easier to spread it geographically and domain-wise and an acceptable analysis effort for its end goal.

To support the survey's design, we collaborated with a few researchers who gave feedback and helped design the questions. The survey underwent several iterations to remove ambiguities and achieve a clear structure.

Instrument Design

The survey was built using Google Forms and is organised into seven sections with a total of 31 questions: experience and background (8), strategies and processes (4), tools (5), refactoring techniques – splitting the monolith (3), refactoring techniques – decomposing the database (3), refactoring techniques (2) and evaluation (6). The first section is used to collect experience, background and demographic information. The second section goes about the strategies and processes of the migration. The third section explores the knowledge of tools to assist the migration. The fourth and fifth sections explore refactoring techniques for splitting the monolith and decomposing the database, respectively. Here, the questions were based on a Likert scale [50, 63], going from

Strongly Disagree to *Strongly Agree*. We then followed up with two open questions: a question about techniques they often use together and in what order they use them and a question about the challenges of applying those techniques. The sixth section is composed of questions about refactoring techniques in general, where we included two questions that asked for more techniques they used in the migration process that were not mentioned and about the most important challenges faced in the migration process. The last section questions how the practitioners evaluate the decomposition result.

The mandatory questions are almost all close-ended, except the ones regarding evaluation. The optional questions aim to explore further the close-ended questions. Therefore, their nature is more open-ended.

The answers to the survey were aggregated in a single spreadsheet that was used for the data analysis.

A concern regarding the survey was its length, as a few people piloted it, and the average response time was around thirty minutes. We knew it takes significant time to answer a survey voluntarily. However, to reach the end goal of this survey, there were no more questions we felt could be discarded, and we have yet to find a way to reduce the response time.

The survey is available as a part of a replication package available at:
https://github.com/RitaPeixoto/Migration-of-Monoliths-to-Microservices-Survey_replication_package.

Sampling

This survey's target was practitioners currently working in the industry and have been involved in at least one migration from monolith to microservices.

To construct the sample for the survey, the survey was disseminated via email to personal contacts that consequently shared it with their contacts in the industry. The survey was also shared via social networks, online communities and mailing lists. The recipients were also asked to share the survey with their acquaintances that fit the target group explicitly described in the information. Therefore we can say that our sample is a *convenience sample*, possibly with *referral-chain sampling* [81].

The survey became available on 2023-01-03 and closed on 2023-06-15, having a total of sixty-six (66) responses on this report's delivery day.

3.4.4 Data Preparation

Before analysing the data, there were some rules we needed to verify to reduce the noise of our study. First, we set a prerequisite of currently working in the industry. Therefore, respondents whose primary working areas were only scientific research, coaching or teaching do not fit our target group. So we chose to ignore the responses of these types of respondents; luckily, in this case, it was only one response.

Then we have to assess the experience in microservices. Another prerequisite we set is to have participated in at least one migration. However, we may find odd the number of years of experience

in microservices, whereas it is strange if it is fewer than one. For example, one respondent had 0 years of experience in microservices and performed one migration. We needed to see the rest of his responses to understand whether this would cause an outlier. Luckily, the responses seemed fit, so we kept this response in our analysis.

The "other" open option responses were uniform to the first letter being capitalised; all the typos were removed, and one answer with only one Portuguese word was translated by us, as it had no other possible meaning.

The countries' names were standardised, and invalid answers were handled. For example, we had to alter an answer that did not correspond to a country but to a company, and we added the company's headquarters country.

To reduce the granularity when not necessary, when there were too many different responses on the "other" field, we aggregated the answers to that questions that only appeared once or much less relative to the others under the "Others" name.

3.4.5 Data Analysis

Here we report our analysis and key insights divided by the sections of the form.

Experience and Background

In this section of the form, our aim was to characterise our sample in their experience, background and demography.

Demographics: Our respondents are primarily from Brazil (32%), but there is a good demographic distribution. We have five continents represented, even though South America and Europe account for the majority of the country's representation (Figure 3.3).

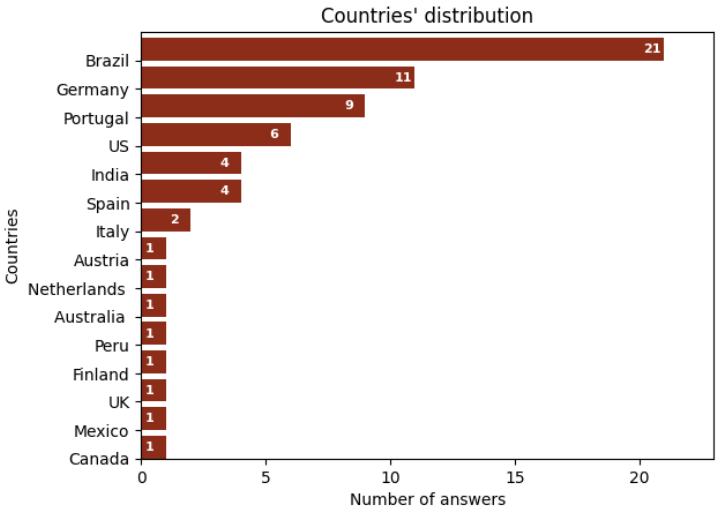


Figure 3.3: Countries representativeness

Background: Over the last five years, our respondents' primary work areas have been software development (92%) and software architecture (86%) (Appendix B.2.1). Our respondents' job titles

were diverse, with a slight emphasis on Architect, Senior Developer, and Software Engineer[5]. The number of years of professional experience in microservices ranges from 0 to 20. Almost 73.8% of the respondents had seven or fewer years of experience in microservices (Appendix B.2.2). Approximately 50 % were involved in 2 or fewer migration projects, which is also not surprising; given the topic's relative newness and how it can take several years to finish a migration project, we expected this number to be lower (Appendix B.2.3).

Experience: When it comes to the number of users these systems served when the migration was started, 32.3% of them served more than one million users, and only 9.2% served less than 100 users (Appendix B.2.4). As for the number of people working on these systems when the migration started, the biggest share (39.1%) of the projects had about 10 to 50 people working on it, and the maximum is 200 (Appendix B.2.5). The main domain areas of the projects migrated from our respondents in descending order are[5]: Finance (35.4%), Retail (27.7%), Education (15.4%) and Security (12.3%) (Figure 3.4).

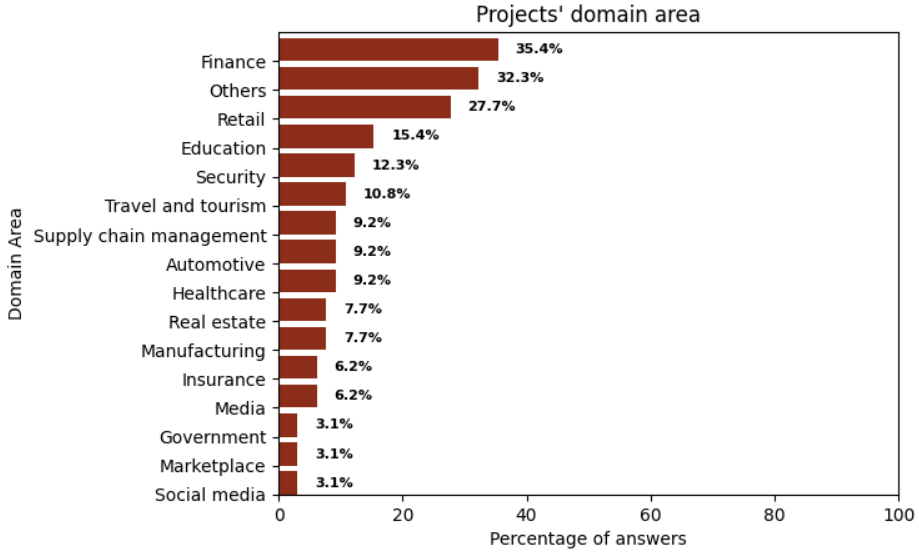


Figure 3.4: Domain areas of the migration projects

Strategies and Processes

In this section, we aimed to analyse the strategies pursued and the migration processes from a monolith to a microservices system.

Guidance: 84.6% of respondents look for guidance in web resources and blogs and 64.6% on other practitioners' experiences and books. Only 32.3% looked for guidance in scientific articles, even though 55.4% looked for it in conference presentations[5] (Appendix B.2.6).

Plan the migration: 73.8% respondents reported performing continuous refactoring interspersed with product evolution, while 36.9% rewrites or rebuilds the entire system from scratch

[5]This was a multi-select question, so these percentages overlap.

and 35.3% stop product evolution, refactors the system to microservices and then proceeds with the product evolution[5] (Appendix B.2.7).

Decomposing the monolith: When it comes to data sources considered when deciding how to decompose a monolith into services, respondents rely more on software documentation and development processes data and not so much on the output of static and dynamic analysis tools (Figure 3.5)[5]. Regarding the criteria used for deciding the new service boundaries when decomposing a monolith into microservices, 78.5% decompose by subdomain, 64.6% decomposes by business capability, 49.2% decomposes based on coupling and cohesion of the data owned by the monolith and 46.2% based on coupling and cohesion of the business logic of the monolith (Figure 3.6)[5].

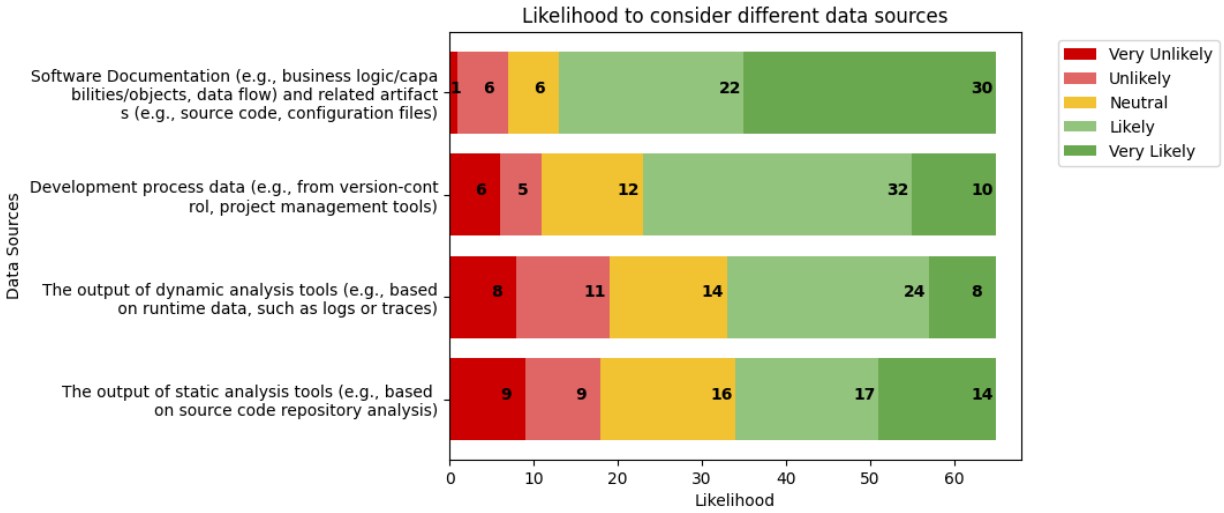


Figure 3.5: Likelihood of considering each data source when deciding how to decompose the monolith into services

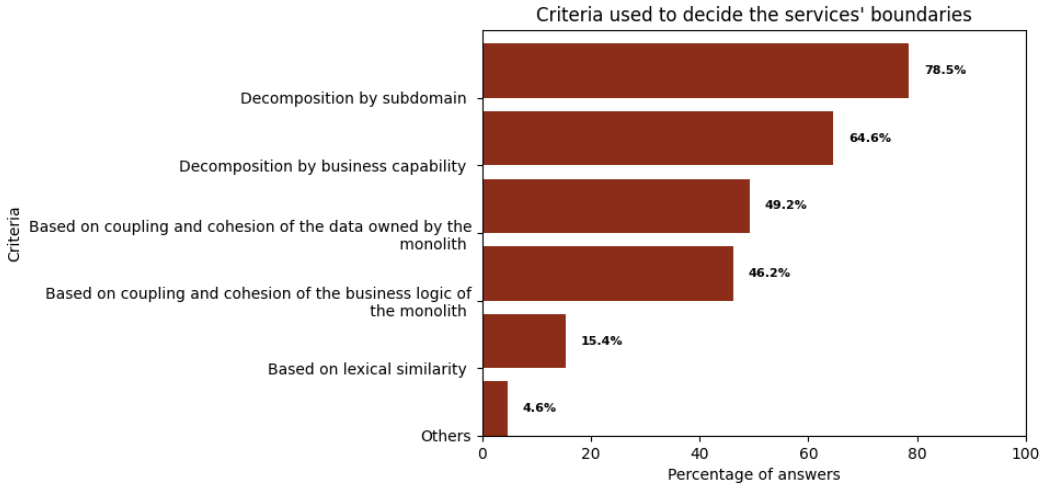


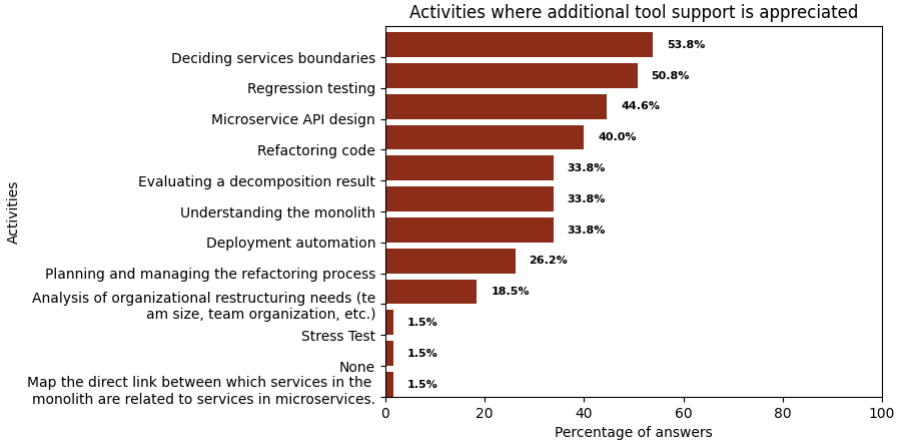
Figure 3.6: Criteria used to decide the new service boundaries when decomposing a monolith into different services

Tools

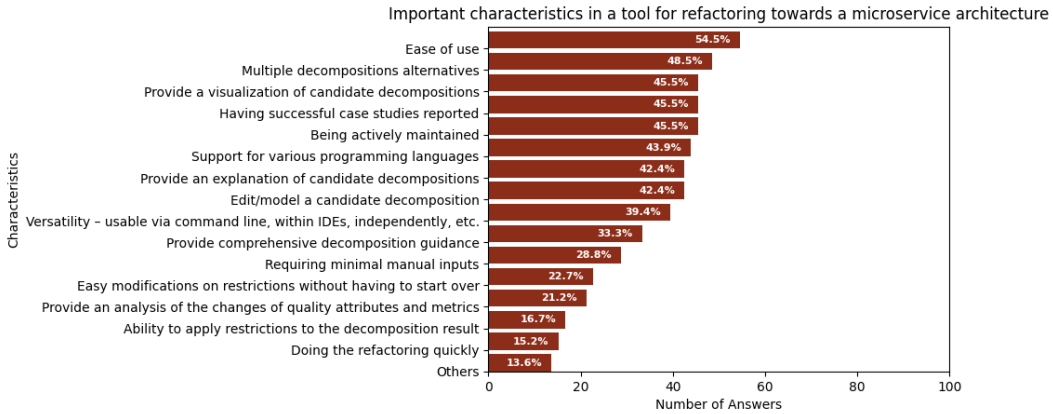
This section aims to understand how established the use of tools to assist the migration is, what tools are being used and what practitioners value in this type of tool.

Use of tools for automation: It is interesting to conclude that 61.5% of our respondents have never used any automated or semi-automated tools to assist their migration (Appendix B.2.8). The most mentioned tools used were Jenkins, toMicroservices, CI/CD pipelines and Terraform.

Additional tool support: In terms of where they would appreciate additional support is deciding services boundaries (53.8%), regression testing (50.8%), microservice API design (44.6%) and refactoring code (40%)[5]. As for the characteristics they find the most important in a tool to perform the refactoring towards a microservices architecture are: ease of use (53.8%), multiple decompositions alternatives (47.7%), providing a visualisation of candidate decompositions (46.2%), being actively maintained (46.2%) and having successful case studies reported (44.6%) (Figure 3.7)[5].



(a) Activities where practitioners would appreciate additional tool support



(b) Important characteristics in a tool for refactoring towards a microservice architecture

Figure 3.7: Additional tool support

Refactoring Techniques

In this section, we question the importance of specific refactoring techniques in the migration projects our respondents participated in.

Techniques: From the graphic showing the importance each technique had on the migration project(s) of the respondents (Figure 3.8), we can see that the *Strangler Fig* has the highest level of agreement from the refactoring techniques described to split the monolith. Also, to split the monolith, the *UI Composition*, *Parallel Run*, *Branch by abstraction* and *Change code dependency to a service call* were agreed to be relevant to at least 57% (more than 37) of the respondents. However, the *Decorating Collaborator* and the *Change Data Capture* are the least popular techniques, with a level of agreement below 50%.

Regarding refactoring techniques to decompose the database (Figure 3.8), *Change Data Ownership* is the most popular, the only one that most respondents agreed to be important, followed by *Database Wrapping Service* and *Database View*. While the *Split Table* and *Aggregate Exposing the Monolith* were the less agreed on.

Challenges of the migration process: The most important challenges faced in the migration process were found to be database migration and data store splitting (70.7%), dealing with data consistency (64.6%), ensuring reliability (52.3%) and communication among services (52.3%)[5] (Appendix B.2.9).

Evaluation

This section's purpose is to assess how the practitioners evaluate and validate the result of the decomposition.

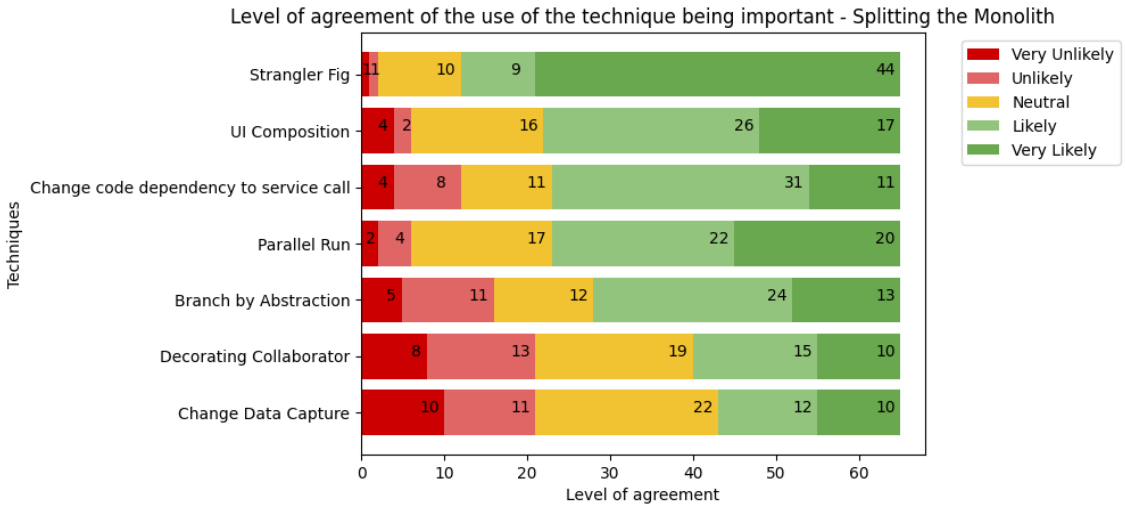
Quality attributes: The quality attributes most assessed by our respondents when evaluating the decomposition are performance/efficiency and scalability (61.5%), maintainability (58.5%) and availability (44.6%)[5] (Figure 3.9).

Environments: Most respondents evaluate the decomposition results in different environments, with 75.3% evaluating during development, 64.6% during staging and 58.5% during production[5] (Appendix B.2.10).

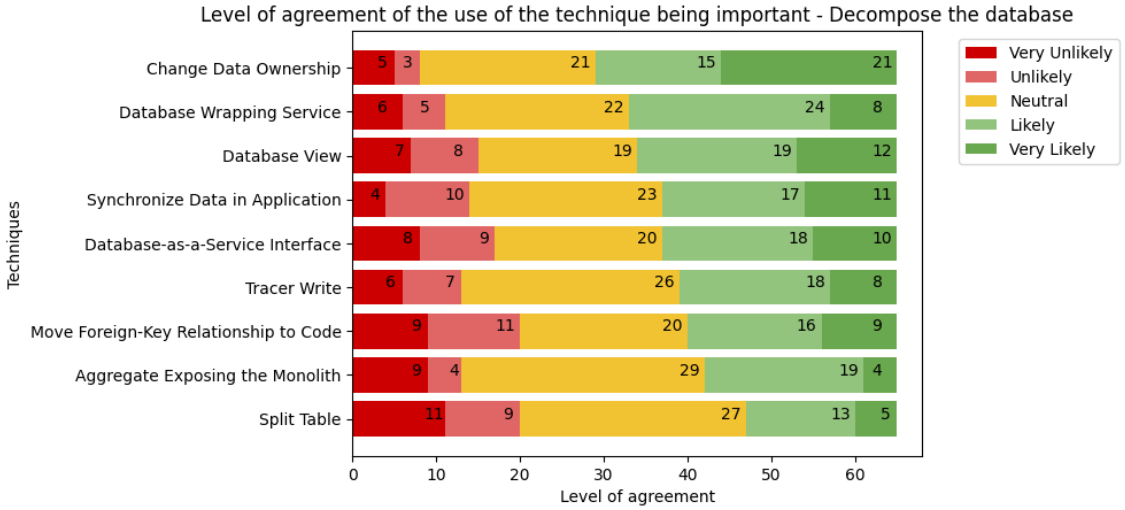
Inputs: 76.9% uses the functional tests as input to evaluate a decomposition result, 58.4% production inputs and 56.9% simulation inputs[5] (Appendix B.2.11).

3.4.6 Findings

To characterize our respondents in terms of migration experience, most of our respondents have seven or fewer years of experience in microservices, which did not take us by surprise as microservices' popularity has been increasing in the last few years, and before that, little was spoken about this topic. We can conclude this by comparing the number of papers published on this topic before and after 2016, which shows an astronomical difference. Also, half of the participants were involved in 2 or fewer migration projects, which is also not surprising; given the topic's relative



(a) Refactoring Techniques for splitting the monolith



(b) Refactoring Techniques to decompose the database

Figure 3.8: Refactoring techniques that were important to use in the migration project(s)

newness and how it can take several years to finish a migration project, we expected this number to be lower.

To discuss the findings of this survey, we answer the research questions elicited in Section 3.4.2:

RQ1. What is the refactoring process that they follow?

We can divide the process into three levels: the choice of how to do it, service decomposition and refactoring.

Concerning how to do it, practitioners plan the migration mostly to be interspersed with the product evolution. This allows different teams to work on different ways to evolve the system, and we believe they choose this approach to measure efforts and value. They mainly focus on

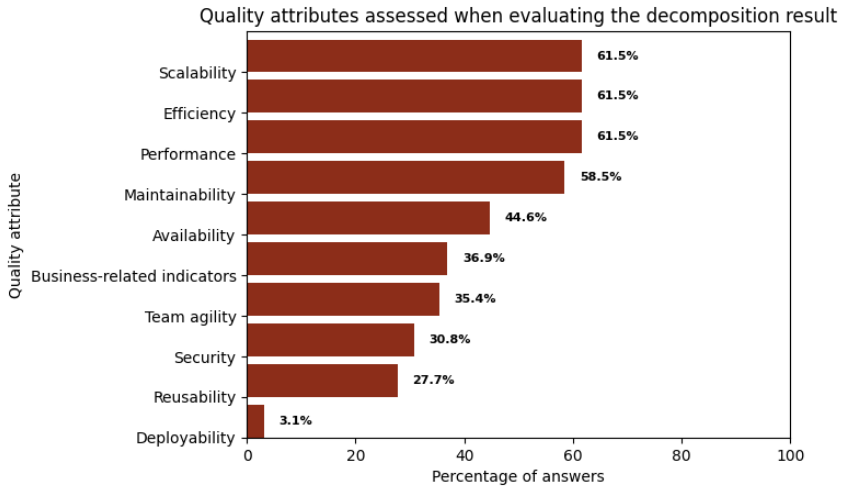


Figure 3.9: Quality attributes assessed when evaluating the decomposition result

web resources and other practitioners' experiences to guide their migration. Only a comparatively small share looks for scientific articles. Therefore, we believe that once we build a tool we can present to practitioners and it works according to their needs, other practitioners may start using it as well.

Regarding service decomposition, software documentation and development process data are used as data sources to decide how to decompose the monolith into microservices. In contrast, the output of analysis tools could be more emphasized. Here we expect to find the relationship between classes expressed in the software documentation, or else we can not understand the low level of usage of static and dynamic analysis tools. They mainly focus on decomposing by business capability or subdomain, which we believe is done because they want to divide into services that make sense and aggregate the functionalities that must be together in terms of business.

In terms of refactoring techniques used for splitting the monolith, Strangler Fig, UI Composition, Parallel Run, Branch by Abstraction, and Change code dependency to service call are the most used. Some of these techniques are implemented in specific parts of the migration, so it is common to see Strangler Fig being used together with Parallel Run, UI Composition, or Decorating Collaborator. Practitioners assume different roles for each pattern and use them that way in their migration. For instance, some use *Strangler Fig* during the implementation of the new services, *Branch by Abstraction* during implementation and *Change data Capture* during planning for migration, microservices development and implementation. When decomposing the database, Change Data Ownership is clearly more used than the other techniques. We see Synchronize Data in Application being used together with Tracer Writer, Change Data Ownership, or Database as a Service. Some identified Split table as a step of Change Data Ownership and Database Wrapping Service before Change Data Ownership. We concluded from the open question on how they used different techniques together that it varies greatly from experience to experience. Other techniques were identified apart from those presented, like service proxy between the monolith and the new services, extract class, replace inheritance with delegation, data mesh, database sharding and

replicas, clean architecture and different approaches to drive the design. The challenges identified in applying the mentioned techniques revolved around microservices boundaries (it is worth noting that a significant portion of the literature extensively explores this particular aspect), getting data ownership right, finding a source of truth and data synchronization, maintaining efficiency, understanding microservices might not be the solution, lack of time and knowledge, business, deployment, downtime and understanding how to apply these techniques. Challenges faced during the migration process are database migration and data store splitting, consistency, and ensuring reliability.

RQ2. What tools do they use?

A large share of practitioners does not use any tools to assist their migration. It may be justified by the fact that only 32% look for guidance to their migration in scientific articles, so many are unaware of the existing tools. There is limited availability of commercially-oriented tools developed for this specific purpose. The ones that do, find useful tools for automated testing, CI/CD, and container orchestration. More specifically, Jenkins, toMicroservices, Terraform, Dynatrace and SonarQube were pointed out. The majority of these tools do not seem to be directly related to the migration process itself but rather serve as utilities for working with microservices. They find that tools for deciding service boundaries, regression testing, microservice API design and refactoring code are needed. Refactoring tools should be easy to use, provide multiple decomposition alternatives and provide visualization.

RQ3. How do they evaluate the result from the decomposition?

Most of the respondents, when asked how they usually evaluate the result of the decomposition, mentioned that they did not do it, it was not worth the effort, or only sometimes, and some specifically said it is based on intuition. As to specific ways on how they evaluate were mentioned tests (automated, smoke, regression, performance, chaos and canary), checking the evolution of metrics and quality attributes (before and after) like maintainability, scalability, business-related indicators, memory usage latency, team related metrics, architecture fitness functions, maintenance effort, easiness of horizontal scaling and infrastructure costs, as well as some other non-functional requirements. Maintainability, Performance and Scalability are the quality attributes most assessed, which is understandable because they are usually the quality attributes related to the need to migrate to microservices. With the responses to our question of what metrics they use to evaluate these quality attributes, we conclude that, in the rare occasions in which the decomposition is evaluated, it is based on metrics associated with whatever aspect motivated the migration in the first place. Such metrics are not necessarily technical; they can also pertain to organizational challenges, for instance. The assessment is performed in multiple environments, mostly during development and using more functional tests than production input or simulation.

Comparison of results with the related work

Compared to the study performed by Di Francesco et al., our respondents identified the harder challenges related to data: database migration and data store splitting and dealing with data consistency(cf. Section 3.4.5). Regarding migration strategies, Di Francesco et al. identified more occurrences of implementing the new functionalities as microservices than refactoring the existing ones, although with slight differences. In our study, we obtained two out of sixty-five respondents, and they obtained 10 out of 18. It may be since we did not have this option, but it appeared to us as "Other" answers. Another similar conclusion is that the adoption of microservices is phased, not parallel or "Big Bang" (corresponding to rewrite or rebuild)(cf. Section 3.4.5). In this study, most people did not migrate the database, whereas, from the refactoring techniques presented in our survey, both migrating and not migrating are common [24].

We obtained the same results as Taibi et al. of the main issues of the migration being monolith decoupling, database migration and data splitting and communication among services (cf. Section 3.4.5) [95].

Fritzsch et al. concluded that the most mentioned strategy for the migration was rewriting the existing application combined with extending functionality, which is a combination of the most mentioned in our study (cf. Section 3.4.5). Strangler Patterns was also the most used pattern together with Parallel Run [38].

In the study performed by Fritzsch et al., the most mentioned strategy was Rewriting compared to Parallel Operation. In our study, the double of respondents performs continuous refactoring interspersed with product evolution which is the closest option to Parallel Operation (cf. Section 3.4.5). This can mean that in the few years between this survey, practitioners may have become more familiar with the downsides and increased effort of rewriting the entire system.

One unexpected result was the low percentage of respondents that used tools to assist the migration. More than 60% represents a big share of respondents who were never assisted by any tool (cf. Section 3.4.5). Among the ones that were used, the types of tools most mentioned were: testing and quality assurance tools, CI/CD tools, APM (Dynatrace), toMicroservices and Terraform. These are curiously useful in the context of microservices but do not support the process of applying the refactoring. For example, toMicroservices identifies the boundaries between the services. Interestingly, a few respondents stated they developed their own tools.

3.4.7 Threats to Validity

As in any empirical research, this survey is subject to validity threats.

Construct Validity

This is related to interpreting the survey questions. Some questions may have issues in their formulation that lead to inconsistencies. These may be due to participants having different mother tongues, some of the vocabulary can be less established or have a double meaning. We mitigated this by having feedback from multiple researchers and performing a pilot survey.

Representativeness of the Sample

This poses a threat because of the generalisation of the results. We asked the roles, domains and countries to understand our sample distribution to mitigate this. As our sample was constructed by network contacts and shared on social media, we focused on achieving a good distribution across the background and domains. The geographic location became a limitation as Brazil may be overrepresented in our sample. We tried to remove the responses from Brazil to understand the level of influence it had in our conclusions, and the end results remained sensibly the same with no major change besides the fact that we no longer had such a big representation of people that work in the Finance domain. We also have many participants that work on projects with a high number of users but with a small development team, which may also lead our results towards this type of project with a lower representation of the others. Moreover, the same happens for the domain of the projects, as a high representativity of the Finance domain area. We performed a similar experiment and removed these answers to analyse the impact, and once again, we did not find that the representativity of the domain was affecting our results. However, this study required a particular target group which is challenging to find. Given this and that the survey was considerably long (estimated 30 min), we consider our sample size a reasonable one for our research goals.

Voluntary Participants

A survey with voluntary participants will also be subject to the threat of self-selected bias, which means that some people are interested in the topic and, therefore, participate. However, the people who do not participate voluntarily will not be represented. On the other hand, if we were making people do the survey or rewarding them for participating, we would have to deal with the fact that their motivation is not to contribute to the research, which might also bias the results.

Conclusion Validity

This is related to the extracted data and the obtained results. It can lead us to infer an incorrect conclusion about relationships in the observations. One example is self-reporting bias, where participants may answer according to their beliefs rather than their experiences, and we specifically intend to know their experiences. To mitigate this, we formulated the questions expressing clearly what we wanted as an answer, and we analysed further the open questions to understand if there was some outlier of this kind.

Reliability

We have not found evidence of clear disagreement between answers, so the respondents consistently respond to similar questions. None of the responses raised any doubts regarding their reliability.

3.5 Discussion

From the state-of-the-art review performed in this chapter, there are many insights to take as a guide to the work of this dissertation.

Large-scale refactoring research is at a different level than we would like it to be. Most of the publications on this are surveys that try to understand it, but we need concrete conclusions on how to perform it and the implications to consider.

The level of systematisation of the migration process is still relatively low. In the literature review, we identified five migration activities (data gathering, model building, boundaries identification, refactoring, and assessment). However, explicit information on the migration's activities needs to be provided. Besides that, the way these activities are performed and their input and output still vary a lot from approach to approach of the migration monolith to microservices, and many methods are still being studied in this matter.

A good part of the work in microservices research is only theoretical, and the review reveals a higher number of use case studies than technology solutions and tool support studies [69].

If we analyse the challenges identified in the survey (cf. Section 3.4.2), they are similar to the ones found in the literature. Both state the communication and interaction between services, database management and decoupling services from the monolith. In addition, both analyses conclude that tool adoption during migration is relatively low and that the process is primarily based on intuition.

Most existing tools that try to automate the migration process focus on identifying microservices. One notable concern observed in the tools within this domain is the requirement for a more comprehensive understanding of the system's behaviour and changes. Practitioners only accept the output of the tools if they better understand their impact on their system.

In addition to this, a portion of the analysed tools needs manually built input, which makes it difficult to systematise the process as it introduces subjectivity. Most of these tools use a representation of the system as a graph and analyse it to decide how to break it into microservices.

Assessing the result produced by the tool is not a common practice, which means that teams fail to evaluate the system's evolution.

The evaluation also came as a surprise not to be a universal practice. In fact, it's very far from that. Notwithstanding, most of the metrics/quality attributes used to evaluate the systems are the drivers of the migration.

We found one work that proposes a way of refactoring monoliths to microservices, MicroRefact. MicroRefact takes a project, creates its representation, and generates the proposed division into microservices using a pre-existing tool. It then applies that refactoring to the system to achieve the microservices goal. It systematises part of the migration process by implementing the refactoring, which shows the potential of automating some of these refactorings. On the other hand, it only applies a limited number of refactorings, proposing a *"Big Bang"* approach, refactoring the entire system at once, providing only the result. However, instead of proposing the refactoring implementation on the code, we aim to suggest a sequence of refactoring to perform a step-by-step

migration of the system, providing visualisation to assist the comprehension and mitigate doubts that can emerge in developers about how the refactoring is performed. This means we will need to explore further the advantages of software visualisation techniques.

We concluded that we are still a bit far from full automation, but integrating the existing tools with the approach we are trying to propose will take a step further in the automation of this process.

Chapter 4

Research strategy

The purpose of this chapter is to define and plan how to address our research challenges. Considering the state of the art stated in the preceding chapter, we begin by outlining our research problem (Section 4.1) and identifying the dissertation's goals (Section 4.2). Following that, we present the methodology to address these research objectives (Section 4.3).

4.1 Problem Statement

The trend of migrating legacy software to a microservices architecture has been visible in the industry in the last few years due to the high demand for scalable and flexible systems caused by the fast evolution of software. There are multiple strategies to implement this migration, and the most common are to rewrite, rebuild or refactor the system to a microservices architecture.

Most of the time that a system is migrated to a microservices architecture, it is due to the fact that it has grown over the years, becoming large, complex and hard to maintain [38]. Given such, it is clear that this is a time- and effort-consuming process that is challenging and complex. Migrating a system to a microservices architecture has several challenges, namely refactoring, which is a crucial step to achieve this architecture but often implies a manual job performed by the system's experts. This migration is usually performed differently for each case, and no systematisation exists.

In Chapter 3, we analysed the current state of many topics related to the refactoring of monoliths towards a microservices architecture. Based on the analysis performed, we can immediately identify specific unresolved concerns with the current proposals in this domain.

Large scale refactoring, there is still not enough information on how to perform refactoring on a large scale, as most works in this domain are surveys. On top of that, large-scale refactoring still presents many challenges regarding its pre-conditions.

Systematization, even though it is pretty popular, the migration process has yet to be systematised, so developers still rely heavily on their intuition to perform the refactoring. There are steps that occur in many different cases that can be systematised and contribute to the automation of the process. However, systematisation currently focuses a lot on the pattern level, not the refactoring

level, which does not provide much assistance to the developer because it focuses more on what behaviour it wants to achieve and not on how to get there. It would reduce the time and costs to complete the migration and guide the developer through the process. We must start by identifying recurrent refactorings and describing them.

Automation, there is a lack of automated tools to assist the migration of monoliths to microservices besides tools to detect service boundaries, and we look far from fully automating this process. In addition, developers are very sceptical when accepting the output of automated tools because they are aware that refactoring is complex, and it is even harder to identify where to apply each refactoring and how to do it in different contexts. Herefore, to accept that a tool can do all this that they consider complex takes much trust. However, it is currently necessary that this process becomes more automatic and less expensive at all levels.

Manually Built Input, many of the existing tools still require some manually built input. As expected, this is not ideal for developers because the cost of using a tool may surpass the cost of creating the input, depending on how hard it is to build it.

Sequences of refactoring have been explored in small-scale refactoring. However, to guide the refactoring to microservices, we must adapt it to large-scale refactoring and understand how to suggest a sequence of large-scale refactoring to change architecture.

Visualisation, current tools do not provide sufficient visualisation to the user of what is happening in the organisation and code of the system, resulting in a lack of trust.

Evaluation of the resulting decomposition is still not a universal practice, so there is not much to evaluate on the system to guarantee the refactoring was correctly performed. Anyhow, only a few approaches assess the result of the decomposition and mainly focus on the quality attributes that lead to the migration.

There is much work on assisting the decisions of what service to extract, as we saw in the previous chapter, but not on how to really extract them.

4.2 Research Goals

This dissertation aims to explore automatically suggesting a refactoring sequence to assist developers in migrating monoliths to microservices. After reflecting on the open issues of this domain stated in the previous section, we decided to focus the research and experiments of this dissertation on the following matters: large-scale refactoring, systematisation, automation, sequences of refactoring and visualisation.

By addressing these issues, we expect to contribute to the software maintainability and evolution field by facilitating the adoption of a microservices architecture in existing software systems, enabling teams to achieve agility and scalability in response to the evolving business requirements and industry needs.

With the hypothesis (defined in Chapter 1) and focus specified, we determined the following research questions we expect to answer:

RQ1. *To which extent can known microservice migration techniques be described as sequences of smaller-scale refactorings?*

By smaller-scale refactorings, we mean that to apply a "bigger" refactoring, we have to do it by applying a sequence of "smaller" refactorings than the techniques identified in the book *Monolith to microservices: Evolutionary patterns to transform your monolith* (Newman, 2020) [75]. For example, when we apply a Strangler Fig, what smaller-scale refactorings are inside this high-level refactoring? One could say, for example, *Extract Service* or *Move Foreign-key relationship to code*, considering that these refactoring examples are at different depth levels. No research shows that refactoring can be fully automated without being limited to the system being in a specific framework or programming language, limiting the spectrum of systems it can serve.

RQ2. *To what extent can we systematically generate a refactoring-based plan to migrate to a microservices architecture given a desired decomposition?*

We have no evidence that the refactorings suggestions generated will assist the developers in refactoring towards a desired decomposition. We need to understand how we can assist in planning the migration with refactorings and how to generate the refactorings. We want to systematise both the possible refactorings and how to choose the refactorings we need to perform the migration. It is also critical that we do so in a reasonable amount of time and with minimal developer effort to tackle some of the previously mentioned issues.

RQ3. *Is it possible to assess the impact of a refactoring sequence on the system's evolution step by step?*

With this research question, we want to support assessing the system's evolution step by step to provide more understanding through the suggested refactoring since the early stages of the refactoring. However, we want to do this without applying the refactorings.

Moreover, we aim to assess the viability of employing incremental refactoring in conjunction with continuous integration and continuous deployment as a suitable approach for migration while ensuring that ongoing migration activities do not pose any risk of system malfunction.

4.3 Methodology

In order to meet these research goals and validate our hypothesis, we expect to bring the following contributions:

- creation of a catalogue of refactorings;
- design and prototyping of a tool to support the refactoring of a monolithic architecture to microservices systematically by suggesting refactoring sequences;
- empirical study with software professionals to evaluate the approach using the prototype developed.

We will create a catalogue of refactorings based on the literature and the industry survey conducted to analyse state of the art, identifying refactorings common in migration. We are building a catalogue of refactorings instead of patterns because a catalogue of patterns needs to give clear insights on how to perform the refactoring necessary to implement that pattern, and that is an issue we want to mitigate. Each large-scale refactoring should describe the sequence of small-scale refactorings that compose it. As a result, the catalogue will aid in creating refactoring sequences and serve as a resource for developers. This will be a very manual and time-consuming task because these refactorings are frequently disguised as migration patterns or common refactorings that are not necessarily architectural.

The tool we aim to create will not start the migration process from scratch. We assume we can use one of the previously analysed tools to perform the prior migration steps: data gathering, model building and boundaries identification(cf. 3.3.7). This way, we will focus only on the refactoring activity. The service boundaries suggested by our chosen tool will be our tool's primary input. One of the goals of this tool is to avoid the need for manually built input and to systematise the choice of refactorings for each specific system. As we have seen in the state of the art, the tool must be developed human-centred, having a simple and intuitive interface. Besides that, to gain the developer's trust, we want to create a visualisation step, so the developer can see how the refactoring will be performed and its impact on the system. Ideally, we should also assess how the quality metrics of the system are evolving. The developers should choose the quality metrics relevant to the system.

Lastly, refactoring should not be implemented in one single step. Instead, it should be implemented in sequences of refactorings that provide enough information to the user on how to perform the refactoring step by step, being able to divert from it if they believe there is a better alternative. These are the type of sequences we want to suggest.

We will evaluate the catalogue by conducting a study to individually validate the refactorings and other assumptions we make throughout the tool development.

We also intend to conduct the empirical study with software professionals because, rather than understanding if our tool works, which can be accomplished through case studies on open-source projects, we want to see if our tool meets the industry's current needs. This dissertation intends to aid developers in migrating the monolith; understanding how well our tool does so is critical. The target group of this study will be software professionals on our networks with considerable experience in migrating monoliths to microservices. We will interview this target group individually to ask them about their perception of the sequence of refactorings generated by our tool on the open source projects chosen, given a go through these projects and context explanation. These interviews aim to understand their perspectives on our tool and jointly understand the future work required.

We do not aim to evaluate the coupling, cohesion or even modularity of the code produced by this tool because the service decomposition significantly impacts it. We would instead assess how it facilitates the work of developers and provides guidance. That is why evaluating this tool will focus on professionals' opinions.

Chapter 5

Catalogue of Refactorings

This chapter shows the Catalogue of Refactorings to migrate a monolith system towards a microservices architecture developed in the scope of this dissertation. We aim to give developers a guide to assist their migration by cataloguing the refactorings found in the literature. We lack systematisation of how to change the code to convert monolithic systems to a microservices architecture. By systematising this as a catalogue of refactoring, we are contributing to automating this process.

The catalogue is available in full in Appendix C. It contains large-scale refactorings, the smaller-scale refactorings that compose it, and the steps to implement the smaller-scale ones, like a recipe to follow, touching very different topics of the migration process across eight different chapters:

- Chapter 1 (Appendix C.1) describes refactoring sequences for breaking existing dependencies between microservices.
- Chapter 2 (Appendix C.2) outlines infrastructure changes needed for a microservices design.
- Chapter 3 (Appendix C.3) explores microservice deployment and orchestration strategies.
- Chapter 4 (Appendix C.4) describes the properties of microservices and how the refactorings in this catalogue help to achieve them.
- Chapter 5 (Appendix C.5) outlines different refactoring techniques that lead to intermediate states of refactoring when we have constraints and limitations during our refactoring process.
- Chapter 6 (Appendix C.6) contains sequences of common refactorings.
- Chapter 7 (Appendix C.7) tackles additional challenges when transferring a system to a microservices design.
- Chapter 8 (Appendix C.8) discusses how to prepare for the migration process.

Any time in this chapter that we will mention the chapters of the complete catalogue, there will be a connection to the referring part on the version in Appendix C.

5.1 Methodology

This catalogue was built based on the literature (cf. Section 3.3) and the industry survey (cf. Section 3.4), and we leaned on some *grey literature* to improve the descriptions of the refactorings.

We started by identifying the higher-scale refactorings we know from the literature review, and from there, we identified the smaller-scale refactorings that compose them. To do this, we tried to understand what is needed in each situation to solve the present issue and then break it into smaller steps to be more actionable.

We thoroughly analysed the *Monolith to microservices: Evolutionary patterns to transform your monolith* (Newman, 2020) [75] book patterns and the survey and concluded that most corresponded to intermediate refactorings as they did not lead to a final state of microservices architecture. Then we combined all papers and articles analysed in our literature review and identified all patterns and refactorings mentioned. With this, we wrote all these in the format we decided to follow and started identifying what we thought was missing in the smaller-scale refactorings. These were the refactorings we designated for "breaking dependencies"; by this, we mean breaking the functional dependencies between services, this can mean changing a local dependency to a remote dependency or changing a local dependency to be a local dependency still, but that facilitates in the future the transition to a remote dependency. We spent the most time in this section, which we explored in Chapter 1 of the catalogue(Appendix C.1).

In addition, we examined the characteristics of a microservices architecture proposed by James Lewis and Martin Fowler [62]. We created our description of them, adding the refactorings in the catalogue that contribute to achieving these characteristics.

Lastly, we decided not to investigate how to achieve specific quality attributes or other challenges that may occur when describing the refactorings. Because of that, we decided to create two additional chapters, one to talk about how to deal with these additional challenges, giving good resources to understand more about it and one to help prepare the migration process, where we provide some tips on how to do it.

Throughout the catalogue, we assumed that further mechanisms to improve performance, guarantee data consistency and fault tolerance, etc., would need to be put in place to make the system perform as intended, even if the refactoring did not address them directly. In other words, these are not mandatory to achieve the purpose of that specific refactoring but may be needed to transform the system into a microservices architecture truly. Hence, the system makes the most of its architecture's benefits and handles its drawbacks. All the examples in the catalogue use Java[1] as the programming language. We also used REST[2] and Apache Kafka[3] in our examples.

[1] Java.com. Available at: https://www.java.com/en/ (Accessed: Jun. 12 2023).
[2] Ravan et al., "What is rest," REST API Tutorial, https://restfulapi.net/ (accessed Jun. 21, 2023).
[3] "Apache kafka," Apache Kafka, https://kafka.apache.org/ (accessed Jun. 21, 2023).

5.2 Content

We ended up measuring efforts and dedicating more time to the descriptions of the refactorings to break the functional dependencies. As we wanted to create a tool to assist the refactorings, we were more interested in the most easily actionable refactorings, and it became our focus.

Still, we wanted to deliver a catalogue that covered the different topics of the migration, although we delved deeper in detail in the refactoring to "break" dependencies. Therefore, the other chapters are still considered essential for migration, even if we include examples of applications for only some of them.

In this chapter, we only present the important refactorings for our tool development. This includes all the refactorings from Chapter 1 (Appendix C.1) and three from Chapter 6 (Appendix C.6), Extract Service, Strangler Fig and Change Data Ownership. The complete catalogue can be found in Appendix C.

Table 5.1 shows the refactorings in the catalogue and the main references we used.

Table 5.1: Refactorings references

Refactoring Name	**Chapter**	**References**
Change local method call dependency to a service call		[9], [112], [79]
Move Foreign-key relationship to code		[37], [75], [79]
Replicate Data Across Microservices		[35]
Split Database Across Microservices	1	[112], [75]
Create Data Transfer Object		[37]
Break data type dependency		[37]
Duplicate file Across Microservices		-
Introduce the circuit breaker		[112], [79], [82]
Introduce service registry		[9]
Introduce internalexternal load balancer		[9]
Introduce configuration server	2	[9]
Introduce edge server or API Gateway		[9]
Configure service discovery		[79]
Configure health-check		[79]
Enable continuous integration		[9]
Containerize services		[79]
Orchestrate service		[79]
Deploy into a cluster and orchestrate containers		[9]
Centralize logging	3	[79]
Centralize Configuration		[79]
Shared Database		[75]
Database Wrapping Service		[75]

Continued on next page

Table 5.1 – continued from previous page

Refactoring Name	**Chapter**	**References**
Database-as-a-Service		[75]
Database view		[75]
Synchronize data in the application		[75]
Tracer writer		[75]
Separating libraries from their dependents		[75, 9]
UI Composition		[75]
Branch by Abstraction		[75]
Parallel Run		[75]
Decorating Collaborator		[75]
Change Data Capture		[75]
Aggregate Exposing the Monolith		[75]
Extract Service		[108], [79]
Strangler Fig		[75], [108]
Typical functionality library pattern		[112]
API Composition		[68]
SAGA		[68]
Change Data Ownership		[75]
Tolerant Reader	6	[112], [32]
Anti-corruption Layer		[112], [58], [23]
Adapt Service Interface		[61], [59], [100]

The refactorings are presented using a structure similar to the one proposed by Martin Fowler and Kent Beck in their 2nd edition of the book *"Refactoring: Improving the Design of Existing Code"* [33]. They are composed of four sections:

- **Title** corresponds to the name of the refactoring. This name is designed to be as self-explanatory as in any code refactoring naming.
- **Motivation** describes a scenario, the set of observed conditions that may lead to the need to apply the refactoring. We can see it as the driving cause.
- **Mechanics**, presents a step-by-step guide of the refactoring implementation.
- **Example** exemplifies the refactoring application, either by using some code snippets, descriptions or schemas.

The following sections correspond to the parts of the catalogue that are crucial for comprehending the tool developed.

5.3 Breaking Dependencies

To separate the system into microservices, we need to identify the dependencies between the clusters of classes that will make our intended microservices and "break" (cf. Section 5.1) the dependencies between those clusters so that the microservices can function independently.

We can consider multiple types of dependencies. This catalogue chapter (Appendix C.1) describes how to break the different dependencies and how the code should be refactored to achieve that. It is common to find these dependencies types together, so some refactorings described below may link to others and form a sequence of refactorings to solve the dependencies between microservices.

5.3.1 Change Local Method Call Dependency to a Service Call

Motivation

When splitting the monolith into microservices, it is prevalent to have code dependencies in the form of a method invocation between components. As the classes in question will become part of different services, method invocations often have to be refactored to service calls in the microservices architecture. This service call should be made with a protocol that can be synchronous or asynchronous, depending on the requirements and goals.

When we do not want a service waiting for the other to respond and an instant response is not required because we do not need that information immediately, asynchronous calls are usually a good option. This is the case when we accept having eventual consistency when it comes to data operations. It is especially common when a service call triggers other service calls, and we do not want the first service to be busy waiting for these calls to complete. It helps in scalability and availability. It is common to implement it through an asynchronous remote procedure call or messaging with a publisher/subscriber protocol, where services publish messages to a message broker, and the subscriber consumes the messages when available to process them.

Asynchronous calls are not the best solution to ensure we are reading accurate data and need consistency in the moment of the action, as consistency in asynchronous calls is eventual; in this case, a synchronous request/response protocol is usually preferred.

Making this solution asynchronous allows for better scalability and fault tolerance as the services become decoupled.

Mechanics

1. Decide the communication strategy and make the initial configurations to use it.
    (a) Synchronous (using strategies like REST or RPC, for example)
        i. Store the necessary information (e.g. URL) to make the remote calls to the microservice.

   (b) Asynchronous (with technologies like Mosquitto, RabbitMQ, Apache Kafka, etc.): using strategies like Event Sourcing or some form of asynchronous RPC.
      i. Set up a message broker/event bus.
      ii. Create a topic.

2. Configure the microservice that makes the invocation to use the communication strategy.
   (a) Synchronous - Change the method calls to and from local components to be remote calls to a different service:
      i. Create an interface with the declaration of the identified methods.
      ii. Create a class that implements that interface and makes the service calls, a Request Class.
   (b) Asynchronous
      i. This microservice will act like a subscriber, so make it subscribe to the topic you have created.
3. Configure the microservice that has the method to use the communication strategy.
   (a) Synchronous - arrange the microservice owning the method to respond to this communication protocol, creating an API to respond to the service calls. Example:
      i. Create a class that defines the resource paths for the requests and processes them producing a response.
      ii. Add methods to the class to perform the actions required by the service calls.
   (b) Asynchronous
      i. This microservice will act like a publisher, so you make it push the messages it wants to communicate in the topic created.

Note: Make sure to guarantee fault tolerance (check **Chapter 7 (Appendix C.7)**).

Example

Let us imagine we have in the monolith a set of classes for managing orders (candidate to become a new *OrderManagement* microservice) and another set of classes for managing inventory (candidate to become a new *Inventory* microservice). The former includes a class *OrderProcessor* that makes a local method call to the *updateInventory* method defined by the class *InventoryService* of the latter, as shown in Listing 5.1.

```
1 public class OrderProcessor {
2     private final InventoryService inventoryService;
3
4     public OrderProcessor(InventoryService inventoryService) {
```

```
5           this.inventoryService = inventoryService;
6       }
7
8       public void processOrder(Order order) {
9           inventoryService.updateInventory(order);
10          // Other processing logic
11      }
12  }
13
14  public class InventoryService {
15      private final InventoryManager inventoryManager;
16
17      public InventoryService(InventoryManager inventoryManager) {
18          this.inventoryManager = inventoryManager;
19      }
20
21      public void updateInventory(Order order) {
22          inventoryManager.updateInventory(order);
23      }
24  }
```

Listing 5.1: The *OrderProcessor* class from proposed *OrderManagement* microservice makes a local method call to the method *updateInventory* of the *InventoryManager* call of the proposed *Inventory* microservice.

In Listing 5.1, we can see that the *OrderProcessor* class performs a local call to the method of *InventoryManager*, and as we want to create two separate microservice, we need to break this dependency. For that, the method calls should become remote service calls. An example for a synchronous solution using REST is shown in Listing 5.2.

```
1   //OrderManagement Microservice
2   public interface InventoryService {
3       void updateInventory(Order order);
4   }
5
6   @Service
7   public class RemoteInventoryService implements InventoryService {
8       private final RestTemplate restTemplate;
9       private final String inventoryServiceUrl;
10
11      public RemoteInventoryService(RestTemplate restTemplate, @Value("${inventory.
            service.url}") String inventoryServiceUrl) {
12          this.restTemplate = restTemplate;
13          this.inventoryServiceUrl = inventoryServiceUrl;
14      }
15
16      @Override
```

```
17      public void updateInventory(Order order) {
18          String url = inventoryServiceUrl + "/update";
19          restTemplate.postForObject(url, order, Void.class);
20      }
21  }
22
23  public class OrderProcessor {
24      private final InventoryService inventoryService;
25
26      public OrderProcessor(InventoryService inventoryService) {
27          this.inventoryService = inventoryService;
28      }
29
30      public void processOrder(Order order) {
31          inventoryService.updateInventory(order);
32          // Other processing logic
33      }
34  }
35
36  // Inventory Microservice
37  @RestController
38  @RequestMapping("/api/inventory")
39  public class InventoryController {
40      @PostMapping("/update")
41      public ResponseEntity<Void> updateInventory(@RequestBody Order order) {
42          // Update inventory logic
43          // ...
44          return ResponseEntity.ok().build();
45      }
46  }
```

Listing 5.2: *OrderProcessor* uses the *InventoryService* interface to make a synchronous service call to the *Inventory* service using REST.

In the *OrderManagement* microservice, we created the *InventoryService* interface that defined the *updateInventory* method and created the implementation of this interface, the *RemoteInventoryService* class. This class uses *RestTemplate* to send a POST request to the inventory service URL.

The *OrderProcessor* class remains the same, although now it depends on the *InventoryService* interface for updating the inventory.

In the *Inventory* microservice, we created the *InventoryController* class that handles the HTTP POST request for updating the inventory. The *updateInventory* method contains the logic to update the inventory based on the received order.

This refactoring changes the direct local method call dependency to a synchronous RESTful API call, allowing the *OrderManagement* microservice to communicate with the *Inventory* microservice via remote synchronous service calls using HTTP requests.

The asynchronous solution using *Apache Kafka* is shown in Listing 5.3.

```
// OrderManagement microservice
    public class OrderEvent {
    private Order order;

    // Constructors, getters, and setters
}

public class OrderUpdatedEvent {
    private Order order;

    // Constructors, getters, and setters
}


@Component
public class KafkaEventProducer {
    private final KafkaTemplate<String, OrderEvent> kafkaTemplate;
    private final String topic;

    public KafkaEventProducer(KafkaTemplate<String, OrderEvent> kafkaTemplate,
        @Value("${kafka.topic}") String topic) {
        this.kafkaTemplate = kafkaTemplate;
        this.topic = topic;
    }

    public void publishOrderEvent(OrderEvent orderEvent) {
        kafkaTemplate.send(topic, orderEvent);
    }
}

@Component
public class OrderEventListener {

    private final InventoryService inventoryService;

    public OrderEventListener(InventoryService inventoryService) {
        this.inventoryService = inventoryService;
    }

    @KafkaListener(topics = "${kafka.topic}")
    public void handleOrderEvent(OrderEvent orderEvent) {
        inventoryService.updateInventory(orderEvent.getOrder());
    }
}

public class OrderProcessor {
```

```
46      private final KafkaEventProducer kafkaEventProducer;
47
48      public OrderProcessor(KafkaEventProducer kafkaEventProducer) {
49          this.kafkaEventProducer = kafkaEventProducer;
50      }
51
52      public void processOrder(Order order) {
53          OrderEvent orderEvent = new OrderEvent(order);
54          kafkaEventProducer.publishOrderEvent(orderEvent);
55          // Other processing logic
56      }
57  }
58
59  // Inventory microservice
60  @Service
61  public class InventoryService {
62      public void updateInventory(Order order) {
63          // Update inventory logic
64          // ...
65      }
66  }
```

Listing 5.3: *OrderProcessor* uses the *InventoryService* interface to make an asynchronous service call to the *Inventory* service using *Apache Kafka* and Events.

We created the event class, *OrderEvent*. We then created a *Kafka* event producer, implemented an event listener to process the order events and updated the *OrderProcessor* to use the *Kafka* event producer.

The *Kafka Event Producer* publishes the *OrderEvent* to the *Kafka* option, while the *OrderEventListener* listens to the *Kafka* topic and invokes the *updateInventory* method of the *Inventory* service, to perform the actual update. This service contains the logic to update the inventory.

The *OrderProcessor* class starts using the *Kafka Event Producer* to publish the *OrderEvent* so the events can be handled asynchronously instead of directly calling the *Inventory* service.

By making this solution asynchronous, it allows for better scalability and fault tolerance as the services become decoupled.

5.3.2 Move Foreign-key relationship to code

Motivation

When two entities are related and dependent on one another, their relationship is frequently characterised by a foreign-key relationship between the database tables representing each entity. One-to-one, Many-to-One, and One-to-Many relationships are possible.

When we extract a service and realise that these entities should be in different microservices, the database tables that describe them should belong to different database schemas, as each microservice should have its database.

Because one service will own the table containing the foreign key and another will own the table from which the foreign key comes, we must break the database and eliminate the foreign-key relationship. Therefore, as the constraint is no longer in the database, we must move this relationship to the code itself.

Mechanics

The following steps can either be performed after breaking the code or with the code breakage in mind. Whenever services are mentioned, they are mentioned as what will be the end service division, but they do not have to be already implemented.

1. Remove the foreign-key constraint from the table that stores it.
2. In the class of entity (database table) that used to have the foreign-key constraint, create an instance variable that represents the other entity involved in the said relationship and create a column for that variable in this entity table. This variable will no longer be a foreign key but a filter of the select query to retrieve data.
3. Separate the tables into the databases of the different owners (at this moment, this might be more conceptual, but in the future, this will represent the databases of the different microservices).
4. Create an interface for each of these databases that implements the methods of data manipulation.
5. Identify the methods that use/manipulate data from different databases and change them to use the newly created interfaces.
6. When you separate the services, don't forget to use the previous refactoring to "Change local method call dependency to a service call"(5.3.1) to change these local methods to service calls using the primary key as a parameter.

Note: We may need to remove code annotations when using specific programming languages, frameworks or ORMs that use it. The way we join the information of these two entities is no longer through a join query, so data consistency has to be a concern. Do not forget to implement mechanisms to guarantee data integrity and consistency (check **Chapter 7 (Appendix C.7)**). Additionally, we should be aware that the latency of requests increases as we transform the database calls into service calls.

Example

Suppose we have two entities, *Order* and *Customer*, with a foreign-key relationship where an *Order* belongs to a *Customer*. The *Order* entity has a *ManyToOne* relationship with the entity *Customer*, using the *customer_id* foreign-key column for the association, as can be seen in Listing 5.4.

```java
@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "customer_id")
    private Customer customer;

    // Other properties, getters, and setters
}

@Entity
@Table(name = "customers")
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other properties, getters, and setters
}

@Service
public class OrderService {
    private final OrderRepository orderRepository;

    public OrderService(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }

    public void processOrder(Order order) {
        // Perform business logic

        Customer customer = order.getCustomer();
        // Use customer data for processing

        orderRepository.save(order);
    }
}

@Repository
public interface OrderRepository extends JpaRepository<Order, Long> {
    // Order-related methods for data manipulation
}
```

Listing 5.4: *Order* has a foreign-key constraint with *Customer*

The *OrderService* class depends on the *OrderRepository* class for data access and manipulation. In the *processOrder* method, the *Customer* entity is accessed directly through the *getCustomer* method.

Listing 5.5 shows the code after applying the refactoring.

```
@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private Long customerId;

    // Other properties, getters, and setters
}
```

Listing 5.5: The foreign key constraint between *Order* and *Customer* is now in the code.

We removed the foreign-key constraint from the *Order* table referencing the *Customer* table. In the *Order* class, we created an instance variable *customerId* to represent the association with the *Customer* entity. In the *Order* table, we added a column for the *customerId*. Each table goes to a separate database; the *Order* table goes to *orders_db*, and the *Customer* table goes to *customers_db*.

We created the interfaces for data manipulation. *OrderRepository* defines the methods for manipulating the *Order* entity in the *orders_db* database. *CustomerRepository* defines the methods for manipulating the *Customer* entity in the *customers_db* database. We identified that the method *processOrder* interacted with both entities, so we modified it to use the respective interfaces.

5.3.3 Replicate Data Across Microservices

Motivation

Sometimes, different microservices need the same data. Still, if they access the same data source, they will not be totally independent, as one microservice will also be managing the data of another.

To solve this, each service can have its own dedicated database with replication to the shared data source. Ideally, in a microservices architecture, this should occur with no distributed transaction, assuming eventual consistency; however, synchronous replication is also possible. One of the services is the data owner and source of truth. The other simply holds a copy of the data it needs to access and manipulate.

Mechanics

1. Split the database by deciding which service will be the owner of the shared data.

2. There are a few strategies to replicate the data:

   (a) Using mechanisms of the database engine, you can create one or more replication channels between it and the shared data source.

   (b) Using event sourcing, a method of storing (or communicating) data which facilitates data replication because events may be easily repeated. It is a way to keep eventual consistency. It holds events that are frequently objects, and because event sourcing does not need to know its consumers, other technologies can be utilised concurrently (for more on event sourcing, check Martin Fowler's article here).

   (c) Using "Change Data Capture" refactoring (check refactoring C.5.12)

Example

This example will show how to use Event Sourcing to replicate data across microservices.

We will use the example of the entity *Order* containing a relationship with the entity *User* of the type ManyToOne. These two entities will belong to different microservices. Since the entity *User* needs to know its order, one solution is to replicate the data of the entity *Order* for the *User* using Event Sourcing.

To apply event sourcing, we have to identify the entity for data replication, the *Order* in this case; implement event sourcing; publish events, subscribe to events and update the local data. Listing 5.6 shows the *OrderManagement* microservice code to implement event sourcing.

```
1  @Service
2  public class OrderService {
3      private final EventPublisher eventPublisher;
4      private final OrderRepository orderRepository;
5
6      public OrderService(EventPublisher eventPublisher, OrderRepository
           orderRepository) {
7          this.eventPublisher = eventPublisher;
8          this.orderRepository = orderRepository;
9      }
10
11     public Order createOrder(OrderDTO orderDTO) {
12         // Create the order entity
13         Order order = new Order(orderDTO);
14
15         // Save the order in the repository
16         Order savedOrder = orderRepository.save(order);
17
18         // Publish the order created event
19         OrderCreatedEvent event = new OrderCreatedEvent(savedOrder.getId());
20         eventPublisher.publish(event);
21
22         return savedOrder;
```

```
    }

    // Other methods for updating and deleting orders
}
```

Listing 5.6: The *OrderManagement* microservice publishes *OrderCreatedEvent* events every time a new order is created

The *OrderManagement* microservice is responsible for creating and managing orders, so every time a new order is created, it publishes an *OrderCreatedEvent* that contains the newly created order using the *EventPublisher*. The microservices interested in replicating the *Order* data can subscribe to the *OrderCreatedEvent* and update their local order records accordingly. Listing 5.7 shows the code in the *User* side, the microservice that wants to replicate the data.

```
@Service
public class UserService {
    private final EventSubscriber eventSubscriber;
    private final UserRepository userRepository;

    public UserService(EventSubscriber eventSubscriber, UserRepository
        userRepository) {
        this.eventSubscriber = eventSubscriber;
        this.userRepository = userRepository;
        eventSubscriber.subscribe(OrderCreatedEvent.class, this::
            handleOrderCreatedEvent);
    }

    private void handleOrderCreatedEvent(OrderCreatedEvent event) {
        // Retrieve the order details based on the event
        Order order = getOrderDetails(event.getOrderId());

        // Update the user associated with the order
        User user = userRepository.findById(order.getUserId()).orElse(null);
        if (user != null) {
            user.addOrder(order);
            userRepository.save(user);
        }
    }

    private Order getOrderDetails(String orderId) {
        // Retrieve the order details from the appropriate source (e.g., call the
            Order service)
        // ...
        return new Order(orderId, "User123", /* other order details */);
    }

    // Other methods for user management
```

```
31 }
```

Listing 5.7: The *User* microservice listens to *OrderCreatedEvent* events to update its own data storage

As the *User* microservice is interested in replicating *Order* data, it subscribes to the *OrderCreatedEvent* using *EventSubscriber* and the *handleOrderCreatedEvent* method is invoked whenever a new order is created.

Inside this method, the *Order* details are retrieved based on the event. The associated *User* is fetched, and if it exists, the *Order* is added to its list of orders, and this way, it keeps the user's order data updated.

This way, each microservice can independently handle the received events and update its own data storage, ensuring data consistency.

5.3.4 Split Database Across Microservices

Motivation

When extracting a service, we commonly find that some monoliths aggregate in the same database table data that will further need to be split across different microservices, as different microservices access the same database table. Therefore, splitting a monolithic database is not so trivial.

Mechanics

1. Separate into each microservice database only the tables that are only accessed/manipulated by that microservice.
2. Find the columns in the table used by the different newly defined microservices.
3. Analyse which is the case of each table and what solution better fits your system's requirements:
   (a) Two microservices access the same database table but do not update the same columns
      i. You can replicate the data for both microservices using **"Replicate Data Across Microservices"** (5.3.3) and use a data replication mechanism to keep it consistent or
      ii. Decide to which of these microservices each column belongs.
      iii. Split this table inside the monolith schema between the two tables belonging to the different components that will soon be microservices.
      iv. In each component, include the corresponding table and adapt the code to use that table.
      v. If the different microservices interact with what used to be foreign keys on the monolith schema, use the **"Move Foreign-key relationship to code"** refactoring (5.3.2).

(b) Two microservices use the same database table and update the same columns

i. You can replicate the data for both microservices using **"Replicate Data Across Microservices"** (5.3.3) and use a data replication mechanism to keep it consistent or

ii. Decide which microservice should own this data.

iii. Make the other microservice make a service call to update this column.

A. To perform this incrementally, we can first make the change in the monolith to the soon microservice that does not own the data, make the update through a method call, and then, when creating the microservices, use the refactoring **"Change local method call dependency to a service call"** (5.3.1).

Note: Guarantee data consistency (check **Chapter 7 (Appendix C.7)**)

Example

We will present an example for option (b) of the Mechanics without data replication.

Suppose we have two microservices, the *Inventory* microservice and the *OrderManagement* microservice and that both use the same database table called *Product*. If first need to identify the microservice that should own the data, and in this case, we believe the *Inventory* microservice is more suitable. Then we need to define a service API for the *Inventory* microservices with methods to update the specific columns related to inventory management. Then we must remove the direct database updates from *OrderManagement* microservice to the shared columns in the *Product* table and change them to make service calls to the API provided by *Inventory* microservice whenever updates to the shared columns related to inventory management, are required. Listing 5.8 shows the code on the *Inventory* microservice side.

```
1  @RestController
2  @RequestMapping("/api/inventory")
3  public class InventoryController {
4      private final InventoryService inventoryService;
5
6      public InventoryController(InventoryService inventoryService) {
7          this.inventoryService = inventoryService;
8      }
9
10     @PutMapping("/product/{productId}/stock")
11     public ResponseEntity<String> updateProductStock(@PathVariable("productId")
           Long productId, @RequestParam("stock") int stock) {
12         // Call the service method to update the stock of the product
13         inventoryService.updateProductStock(productId, stock);
14         return ResponseEntity.ok("Product stock updated successfully");
15     }
16 }
```

Listing 5.8: *Inventory* microservice code - the owner of the data)

Inventory microservice owns the shared data related to product inventory and exposes and API endpoint *'api/inventory/product/productId/stock'* to update the stock of a product.

Listing 5.9 shows the code on the *OrderManagement* microservice side.

```
@Service
public class OrderService {
    private final RestTemplate restTemplate;
    private final String inventoryServiceUrl = "http://localhost:8082/api/inventory
        /product";

    public OrderService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public void placeOrder(Order order) {
        // Perform order placement logic

        // Make a service call to the Inventory Service to update the product stock
        String updateUrl = String.format("%s/%s/stock?stock=%s",
            inventoryServiceUrl, order.getProductId(), order.getQuantity());
        restTemplate.put(updateUrl, null);

        // Continue with the remaining order placement logic
    }
}
```

Listing 5.9: *OrderManagement* microservice code - the service that uses data owned by Inventory microservice

The *OrderManagement* microservice is responsible for placing orders, and whenever it places an order, it makes a service call to the *Inventory* microservice API endpoint *'api/inventory/product/productId/stock'* to update the stock of the ordered product.

With this refactoring, the *Inventory* microservice has control over the inventory-related data, while the *OrderManagement* microservice interacts with the *Inventory* microservice through service calls to update the shared inventory columns. This way, each microservice focus on its specific responsibilities.

5.3.5 Create Data Transfer Object

Motivation

This refactoring is commonly necessary when we extract a service and there is a relationship between entities that will belong to different microservices. It suggests transferring more data in

each call through a data transfer object that will hold all the call's data. Hence, this object will contain all the data that must be shared between the microservices to reduce the number of service calls performed, as service calls are expensive regarding latency.

It decouples presentation from the service layer and the domain model.

Note: This data transfer object must be serialisable to be sent through the connection.

Mechanics

1. Create an entity (Data Transfer Object) to hold the data necessary in a call between those services.

Example

In this example, we must create a Data Transfer Object to hold the information about an *Order* to be transferred between microservices during order-related communication. An example of the *Order* object being sent in the communication can be seen in Listing 5.10.

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String customerName;

    // Other fields and relationships

    // Constructors, getters, and setters
}

@Service
public class OrderService {
    private final OrderRepository orderRepository;

    public OrderService(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }

    public Order getOrderDetails(Long orderId) {
        return orderRepository.findById(orderId);
    }
}
```

Listing 5.10: *Order* object is being sent through the communication channel.

In the *getOrderDetails* method from *Order* microservice class, an object of type Order is being sent through the communication. However, we want to create a Data Transfer Object that can hold the necessary data in a call to this method that contains more than information only present in the *Order* class. This way, the services will not have to share the same entity because we are encapsulating the specific data for communication, creating an abstraction. Listing 5.11 shows the creation of a DTO for *Order*.

```
public class OrderDTO {
    private Long orderId;
    private String customerName;
    private List<String> products;
    // Other fields as needed

    // Constructors, getters, and setters
}

@Service
public class OrderService {
    private final OrderRepository orderRepository;

    public OrderService(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }

    public OrderDTO getOrderDetails(Long orderId) {
        Order order = orderRepository.findById(orderId);
        // Transform Order entity into OrderDTO
        OrderDTO orderDTO = new OrderDTO();
        orderDTO.setOrderId(order.getId());
        orderDTO.setCustomerName(order.getCustomer().getName());
        orderDTO.setProducts(order.getProducts().stream().map(Product::getName)
            .collect(Collectors.toList()));
        // Set other fields as needed

        return orderDTO;
    }
}
```

Listing 5.11: Creation of a DTO to be sent through the communication

We defined a new class representing the DTO and declared the necessary fields to hold the data. In the future, more fields can be added to this DTO as they correspond to the transferred data. Then, we transform the data being transferred into the DTO. The *getOrderDetails* method in the *Order* microservice will no longer retrieve an *Order* object but a *OrderDTO* object.

The DTO provides a standard format for transferring the data of orders between services.

5.3.6 Break Data Type Dependency

Motivation

A data type dependency is common between different microservices. When we separate the microservices by business capabilities, some microservices might still need information about some entities in specific parts of their operations that are part of a different business capability. This dependency can appear on instance variables types, parameter types, return types and even method variables types.

We must identify where this data type is used and break this dependency to separate the microservices smoothly.

Mechanics

1. Identify where the data type is used (for example, the method invocations from the data type class, variable, parameters, or return types of the data type).
2. There are three ways of doing this:
   (a) Assuming it belongs only to the microservice where it was first defined:
      i. if there are method invocations:
         A. Create an interface with the same name as the data type that defines the methods invocations identified for use through the data transfer object to make service calls to the data owner.
         B. The method invocations will change from local calls to calls to the service that owns the data types and its methods, using the refactoring **"Change local method call dependency to a service call"** (5.3.1).
      ii. The return types, variables and parameters shall use a Data Transfer Object to create the data type in the microservice because it will be sent through the service calls. Use the refactoring **“Create Data Transfer Object”** (5.3.5).
      iii. Make the necessary changes in the code to use the new data type and the right interface for the method calls.
   (b) Keep it in both microservices
      i. Replicate the data type in both microservices and use event sourcing to ensure eventual consistency. Check the refactorings **"Change local method call dependency to a service call: asynchronous"** (5.3.1) and **"Replicate Data Across Microservices"** (5.3.3).
   (c) Keep it in both microservices, but one of them is a proxy.

Example

We will focus this example on the first way of solving this, assuming it belongs only to the microservice where it was first defined.

Imagine we have two microservices *OrderManagement* microservice and *Inventory* microservice, where the *OrderManagement* microservice depends on a data type called *Product*. We want to break the dependency of the *Product* data type in the *OrderManagement* microservice and let the data type be owned by *Inventory* microservice. The *OrderManagement* microservice before the refactoring using the *Product* data type as a variable type, as can be seen in Listing 5.12.

```
1  public class OrderService {
2      private ProductService productService;
3
4      public OrderService(ProductService productService) {
5          this.productService = productService;
6      }
7
8      public void createOrder(Order order) {
9          // Perform order creation logic
10
11         // Directly access the ProductService to get product information
12         Product product = productService.getProductById(order.getProductId());
13
14         // Use the product to complete the order creation process
15     }
16 }
```

Listing 5.12: *OrderManagement* microservice before the refactoring

To resolve it, we create a *ProductInterface* that defines the necessary methods invocations to interact with *Product* data in the *Inventory* microservice. The *ProductService* implements this interface and makes the requests to the Inventory microservice that owns the data type *Product*. This way, We then replace the local method invocations in the *Order* service that involves the *Product* data type with calls to the *ProductService* interface, which will make service calls to the *Inventory* microservice to retrieve or manipulate the *Product* data.

We create a *ProductDTO* to use for transferring the *Product* data between the microservices through service calls, and we modify the return types, variables and parameters in the service's communications to use the DTO. Lastly, we update the *Order* service to use the new data type and the *ProductService* interface for method invocations. All changes performed to the *Inventory* microservice can be found in Listing 5.13 and all changes performed to the *OrderManagement* microservice can be found in Listing 5.14.

```
1  @Service
2  public class InventoryService {
3      public ProductDto getProductById(Long productId) {
4          // Logic to fetch product information from the inventory database or any
               other source
5          // ...
```

```
6
7           // Assume product information is retrieved and stored in the 'product'
                variable
8           ProductDto product = new ProductDto();
9           product.setId(productId);
10          product.setName("Example Product");
11          product.setPrice(BigDecimal.valueOf(9.99));
12
13          return product;
14      }
15  }
16
17  public class ProductDto {
18      private Long id;
19      private String name;
20      private BigDecimal price;
21
22      // Getters and setters
23  }
```

Listing 5.13: *Inventory* microservice after the refactoring

```
1   public class ProductDto {
2       private Long id;
3       private String name;
4       private BigDecimal price;
5
6       // Getters and setters
7   }
8
9   public interface ProductInterface {
10      ProductDto getProductById(Long productId);
11  }
12  @Service
13  public class ProductService implements ProductInterface {
14      private final RestTemplate restTemplate; // or any HTTP client
15
16      public ProductService(RestTemplate restTemplate) {
17          this.restTemplate = restTemplate;
18      }
19
20      public ProductDto getProductById(Long productId) {
21          // Make an HTTP request to the InventoryService to fetch the product
22          String inventoryServiceUrl = "http://inventory-service/api/products/" +
                productId;
23          ResponseEntity<ProductDto> responseEntity = restTemplate.getForEntity(
                inventoryServiceUrl, ProductDto.class);
24          return responseEntity.getBody();
```

```
25      }
26  }
27
28  @Service
29  public class OrderService {
30      private final ProductService productService;
31
32      public OrderService(ProductService productService) {
33          this.productService = productService;
34      }
35
36      public void createOrder(OrderDto orderDto) {
37          // Process the order details
38          // ...
39
40          // Retrieve product information from the ProductService
41          Long productId = orderDto.getProductId();
42          ProductDto product = productService.getProductById(productId);
43
44          // Perform other order-related operations
45          // ...
46      }
47  }
```

Listing 5.14: *OrderManagement* microservice after the refactoring

5.3.7 Duplicate file Across Microservices

Motivation

When extracting a microservice from the monolith, it is common to find the need to have an interface or an abstract class in different microservices.

Mechanics

In this case, duplicating the file for each microservice is okay, as these classes do not handle business logic.

Example

Imagine you have a file called *Utils.java* that defines multiple functions useful for this domain but does not handle any business logic. If the microservice *Order* and the microservice *Inventory* both use multiple functions from that file, we can just simply add the *Utils.java* file to both microservices repositories.

Suppose the microservice *Order* and the microservice *Inventory* both use multiple functions from that file. In that case, we can just simply add the *Utils.java* file to both microservices repositories.

5.4 Common Sequences of Refactorings (partial)

This chapter (Appendix C.6) introduces common sequences of refactorings found when migrating from monoliths to microservices.

5.4.1 Extract Service

Motivation

We may need to extract a service from the rest for multiple reasons, like scalability, ease of deployment, etc. We have multiple functionalities when we have a monolith system, but not all are used as much as the others. As one functionality is used more frequently than the other, scalability can become an issue. This refactoring will transform one or more regular class(es) into a remote service.

Mechanics

1. Analyse all the dependencies of the functionality we will extract to a service. Both the dependencies to the rest of the monolith and that the monolith has with the service to be extracted.
2. Resolve these dependencies. This usually involves refactoring foreign keys and changing method calls to other files/classes/etc. to remote calls (synchronous or asynchronous). Appendix C.1 describes interesting refactorings to resolve these dependencies.
   (a) If external libraries are dependencies, they will later need to be added to the new service.
   (b) All its methods used by other components inside the monolith had to be ready to be called over service calls, like services communication, etc.
   (c) Its original clients have to access it through remote service calls.
3. Create a new project folder with the files identified to be owned by this functionality with the same characteristics and the changes made during the previous steps.
4. Make it capable of running independently, installing the necessary dependencies and preparing its production environment.

Note: Make sure to guarantee fault tolerance and data consistency and to solve performance issues (check **Chapter 7 (Appendix C.7)**).

Example

We have seen in the previous section (cf. Section C.1) many refactorings that lead to the extraction of the services *Inventory* and *Order*. To apply this refactoring, the same way we identified those dependencies, we identify all dependencies these microservices have between themselves and to the rest of the monolith and use similar refactorings to the ones identified in that section to extract them. When all dependencies are resolved, we create a new project for each of these microservices with the designated files that belong to them; we build them and deploy them after correctly preparing them with the necessary dependencies for the production environment.

5.4.2 Strangler Fig

Motivation

When we have decided to evolve a system to a microservices architecture, we want to take incremental steps toward the new architecture and ensure that each step is easily reversible, reducing risks.

Mechanics

1. You can start by deciding if you want to wrap the monolith with an API that allows us to access the new system in the old way. If so, perform the necessary implementations (Branch by Abstraction refactoring, **Chapter 5 (Appendix C.5)**).

2. Start small, with the macro, then micro mindset and identify the functionalities you want to extract and their boundaries.

3. Identify the order by which you want to extract the functionalities. And program how you want to do the extraction.

4. Gradually move functionality over to the new microservices architecture. This technique usually uses as the main refactoring of the extract service refactoring (5.4.1).

5. Reroute calls from the monolith over to the new microservice using the change local method call dependency to a service call (5.3.1).

6. If the new extracted functionality uses functionalities that remain inside the monolith, then the monolith can expose this functionality (C.5.13). Iteratively extract all functionalities.

7. Write new code as microservices.

   Note: Do this by replacing or rewriting existing features parallel to the old architecture, one at a time, until the old architecture has been entirely replaced. Usually, we must create a proxy or façade that provides a stable API for old clients throughout the migration.

Example

As an example, we are going to use the same example given by Sam Newman on *"Monolith to Microservices: Evolutionary Patterns to Transform your Monolith"* [75].

In Figure 5.1, we can see that the *InventoryManagement* functionality is self-contained and, therefore, has no dependencies. So, we can simply extract this service, rerouting existing calls regarding this functionality to this new service instead of to the monolith.

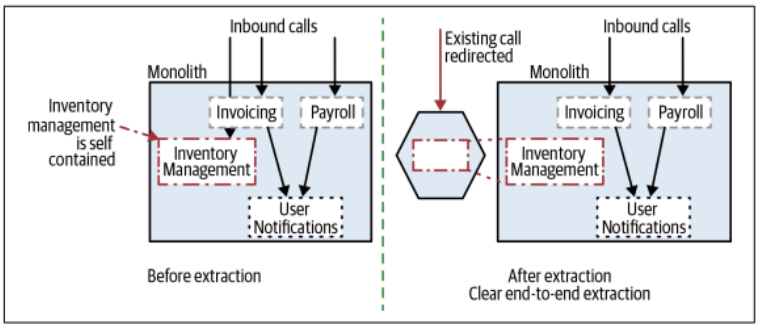


Figure 5.1: Example of application of the Strangler Fig refactoring to the InventoryManagement refactoring
Source: From S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly, 2019. [75]

We gradually plan the extraction of the functionalities to services according to an order we define and at our own pace, and all the new code is written as microservices. These changes can be rolled back whenever we find it is not suitable. If the functionality were to be used by the functionalities inside the monolith, we would have to redirect those too.

5.4.3 Change Data Ownership

Motivation

When extracting a new service that encapsulates the business logic of some data, that data should belong to that service and, therefore, should be moved into the new service. The new service has to be the new source of truth of that data.

Mechanics

To do this, we have to break the dependencies the monolith may have with this data through refactorings like: 5.3.2, 5.3.3, 5.3.4, C.5.2 and C.6.5.

Example

As an example, we are going to use the same example given by Sam Newman on *"Monolith to Microservices: Evolutionary Patterns to Transform your Monolith"* [75].

In Figure 5.2, we can see that we want to move invoice-related data into the new *Invoice* service. Therefore, we need to change the monolith to take the *Invoice* service as the source of truth for invoice-related data and change all calls to read or change the invoice-related data to be made directly to the *Invoice Service*. This can initiate other refactorings like "Move Foreign-key relationship to code" to make this work.

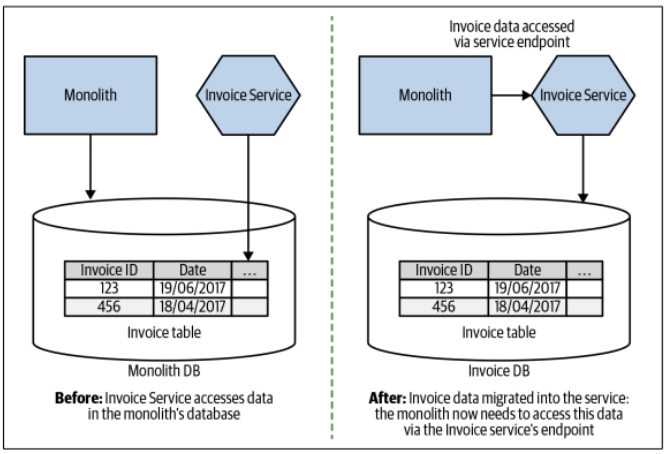


Figure 5.2: Example of application of the Change Data Ownership refactoring of the Invoice table to the Invoice Service
Source: From S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly, 2019. [75]

Chapter 6

An approach to assist the refactoring towards a microservice architecture

As a result of the review of the state of the art (cf. Chapter 3), we identified few efforts in trying to automate the process of refactoring monoliths towards a microservice architecture and some reluctance by professionals in accepting the tools being developed to perform architectural refactorings. Being such a complex task makes it harder for developers to trust the suggestions of a tool as they cannot see how it is affecting their system.

Hence, we want to create a systematic approach to refactoring that can support teams in choosing which refactorings to apply. We seek to support the creation of new refactoring tools that can provide the sequence of refactoring to incrementally extract services from the monolith and create a microservices architecture. We aim to assist the migration process by suggesting a sequence of actions and providing a visualisation of the system's evolution as the refactorings are applied.

Moreover, we want to avoid manually built input, striving for a highly systematic process to automate the migration, thereby reducing developers' efforts. This approach should be language agnostic, implying the necessary abstraction in its implementation.

The approach described in this chapter is the basis for the prototype tool developed in the scope of this dissertation (cf. Chapter 7).

Our review of the state of the art in what concerns existing tools (Chapter 3) also revealed MicroRefact [36] as the only tool that addresses the problem of implementing the refactoring towards a microservice architecture. However, it does not address some essential points identified in our literature review: visualisation to provide assurance and understanding of what is happening and limited systematisation, as they use a limited number of possible refactorings.

Therefore, the focus of our approach will be to provide a flow to the refactoring with visualisation of the changes in the system's overall organisation while helping the developers implement the refactorings. We believe the refactoring catalogue that we have developed (cf. Chapter 5) allows the developer to have a good understanding of the refactoring process.

Nevertheless, there are points in common between MicroRefact and our own work. The information extraction, as done by MicroRefact, obtains the same information we need, so we expect

that part of our approach in that stage to be alike. The refactoring part of their methodology does not suit our goals, as we do not intend to apply the refactorings directly on the code but to find ways of guiding the user through the entire migration process.

6.1 Approach Overview

As mentioned in the methodology topic in Chapter 4, we aim not to address the migration process from scratch but to assist in the refactoring stage. However, before implementing any refactorings, some prior decisions should occur, namely the decision of how many microservices we will break the monolith into and its boundaries, which significantly impacts the resulting system but is not the scope of our research. For that reason, we require as input the intended microservices decomposition, which should describe which files belong to each, noting that a file can only be present in one intended microservice. Needless to say, this input should not be produced manually, as many tools can provide us with a monolith-intended decomposition that can be easily translated into a standard schema. Different tools have different strategies to generate this; the user shall choose the ones that better fit their requirements.

In addition to the microservices identification, we also need a representation of the monolith (its source code and database model) so that it can be generalised for any programming language or framework. This representation should include any information required to understand what refactorings should be used to complete the migration and, once again, should be gathered using existing tools.

The flow of the assisted refactoring tool's approach is depicted in Figure 6.1 represented by the arrows. We can divide the approach into three (3) steps: the *Information Extraction*, the *Refactoring Suggestion* and the *Refactoring Application*. In the *Information Extraction* step, the structural information is extracted from the source code representation that, together with the intended microservices decomposition, allows the identification of the microservices dependencies. Then, all this information is stored in our system's internal representation (the conceptual model). In the *Refactoring Suggestion* step, the developer selects the service extraction order to say what services they give priority to extract first, and we analyse the system using the information from the conceptual model to suggest a refactoring sequence to extract all services in the order provided by the user. Having a sequence suggestion, we move to the last step, the *Refactoring Application*, where the refactorings suggested are applied to our internal representation of the system to obtain the microservice's final representation.

The following sections detail the steps of our approach.

6.2 Information Extraction

The input of this approach is a representation of the source code and a representation of the intended microservice decomposition.

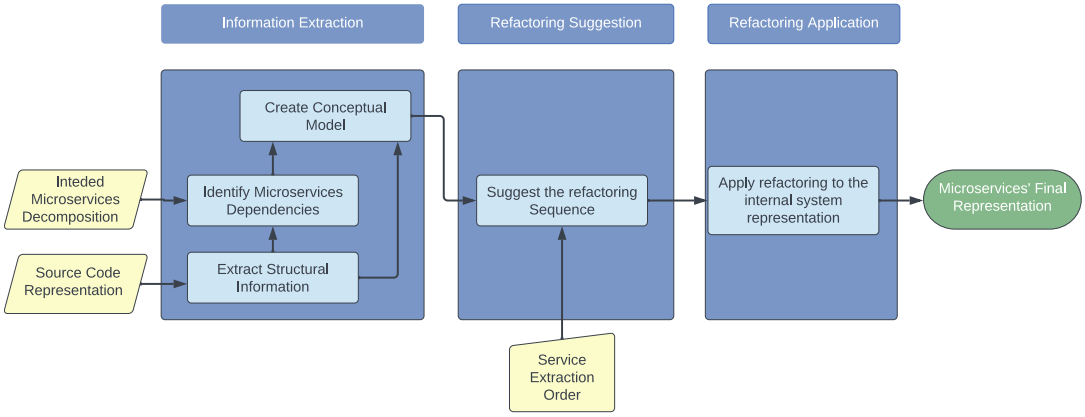


Figure 6.1: Overview of the approach

In refactoring, we want to extract each proposed microservice from the monolith. However, we need to understand that it is common for some files to interact with each other, translating into dependencies. Therefore, before simply extracting these services, we need to point out these dependencies and "break" them (refactor them) so the service can function independently from the monolith. To identify these dependencies, we need both the system's representation and the intended microservices decomposition to cross the information and realise what dependencies need to be refactored.

This phase is crucial, as we need to extract all the necessary information about the project and intended decomposition from the inputs and store it appropriately. Next, we detail how we acquire this information.

6.2.1 Extraction of Structural Information

The extraction of structural information uses the source code and database model representation it receives as input. This representation is expected to be similar to an AST, as it is critical to have all the information about the system's data structures and files to decide what refactorings to apply. This includes methods, variables, imports, return types, database schema, and much more; this input's information will resemble an AST due to that.

To obtain this information is very common to perform static analysis, the examination of the source code and other static artefacts of the system, like architecture and use case diagrams. This analysis can obtain the relationships between files, each file structure and other key points. However, a dynamic analysis is also important to give us additional insight as we evaluate the system's behaviour in run time. Dynamic analysis is especially interesting regarding performance analysis, allowing us to find bottlenecks, resource usage, inefficiency, etc. This data can help us decide whether a service call should be synchronous or asynchronous. We highlight that when it comes to architectural refactorings, that are more artefacts that may be useful for decision, but we need to find a way to obtain them and interpret them correctly, like version control history, architecture documentation, how the architecture manifests in the code and other runtime artefacts [110].

6.2.2 Microservices Dependencies Identification

After identifying the microservice boundaries and the system's structural information, we must identify each proposed microservice dependencies. This refers to the dependencies each file in a specific microservice has on files in another.

We define a dependency as something the file uses and cannot access if the other file is unreachable. We can identify different types of dependencies between files: the data type in a parameter, method variable or even return type; the method call; the database dependency; the import, extend or interface dependency. For instance, if a file "Client.java" creates a class "Client" with a name, email and fiscal number and a file "Billing.java" wants to access the Client's attribute fiscal number by doing "client.getFiscalNumber()", then the file "Billing.java" has a method call dependency to the "Client.java" file.

To identify the dependencies between microservices, we must check each microservice file's dependencies and check from these which ones do not belong to that microservice.

Listing 6.1 presents the algorithm for identifying microservices dependencies. To analyse the dependencies of each microservice, we iterate all microservices files, analyse the dependencies of that file, and store all of this microservice's files. Then we iterate this list where we stored all dependencies to check which of these dependencies are to files that do not belong to this microservice. The files that do not belong to this microservice are their actual dependencies.

```
1  #Input: microservice > All the information we have about the microservice
2  #Output: microservice_dependencies > All the files this microservice depends on
3  def get_dependecies(microservice):
4      dependencies = []
5      for f in microservice.files:
6          dep = f.get_dependencies()
7          dependencies.append(dep)
8      microservice_dependencies = []
9      for dependency in dependencies:
10         for dep in dependency:
11             if dep not in microservice.files:
12                 microservice_dependencies.append(dep)
13     return microservice_dependencies
```

Listing 6.1: Microservice Dependencies Identification Algorithm

Let us use a hotel management system example and focus on determining the dependencies of the first proposed microservice (microservice 1). Table 6.1 shows the files of microservice one, what external dependencies they have, and to which microservices these dependencies belong. From the table, we can see that the output of the previous algorithm for microservice one would be `[2, 10]`, as it depends on files from microservices 2 and 10.

To identify the possible refactorings in the code in the scope of a tool, static analysis plays an essential role as it examines the information regarding all software files so that it can access their attributes together with the relationship between them.

Table 6.1: External Dependencies of microservice 1 of HotelManagementSystem's project

File	Dependens on
RoomOrder	Employee (2),
RoomOrderService	EmployeeDao (2)
RoomOrderController	SessionUtil (2), ExtAjaxResponse (2), ExtjsPageRequest (10)
IRoomOrderService	
RoomOrderDTO	
RoomOrderQueryDTO	
RoomOrderRepository	

In this section, we are only looking to determine which proposed microservices depend on other proposed microservices. However, eventually, we will want to store more information about these dependencies, for instance, which specific files rely on other microservices, what files they depend on or what type of dependency there is, and some other details to be able to suggest refactorings. We look further into it in the implementation.

6.2.3 Conceptual Model

After receiving the information needed, we need to create our own internal representation of the system to make it easier to analyse and suggest refactorings. The approach to creating these conceptual classes should be to only represent what we need to know and will use about these structures since there is typically much more information in a source code representation than what we will need.

Figure 6.2 shows the conceptual model with classes representing the information extracted and their relationship. The classes with grey backgrounds represent the classes necessary to represent the system, and those with white backgrounds are needed to implement the approach.

This conceptual model revolves around the *Service* class representing each intended service and its information. The determining factor indicating whether a service has been successfully extracted is its attribute *independent*. If this attribute is true, it signifies that the service has become autonomous and no longer remains a component within a monolith. Each service has files that have methods and service calls to other services. The database model is represented at the *File* level, for instance, with the database dependencies and types of variables.

A *Service* also needs refactorings to be extracted, stored in the class *Refactoring*, and these refactorings are present in the suggested *Refactoring Sequence*.

To keep track of the system evolution, we also need the *Snapshot* class that contains the various *Service*(s) either dependent from the monolith or independent and all its information after the application of the refactoring. So each snapshot shows how the application of the refactorings is changing the representation of the system.

Upon determining how to collect the required input for this tool, a representation of the system's code and database model and the intended microservices decomposition, we now narrow our focus on the refactoring assistance itself.

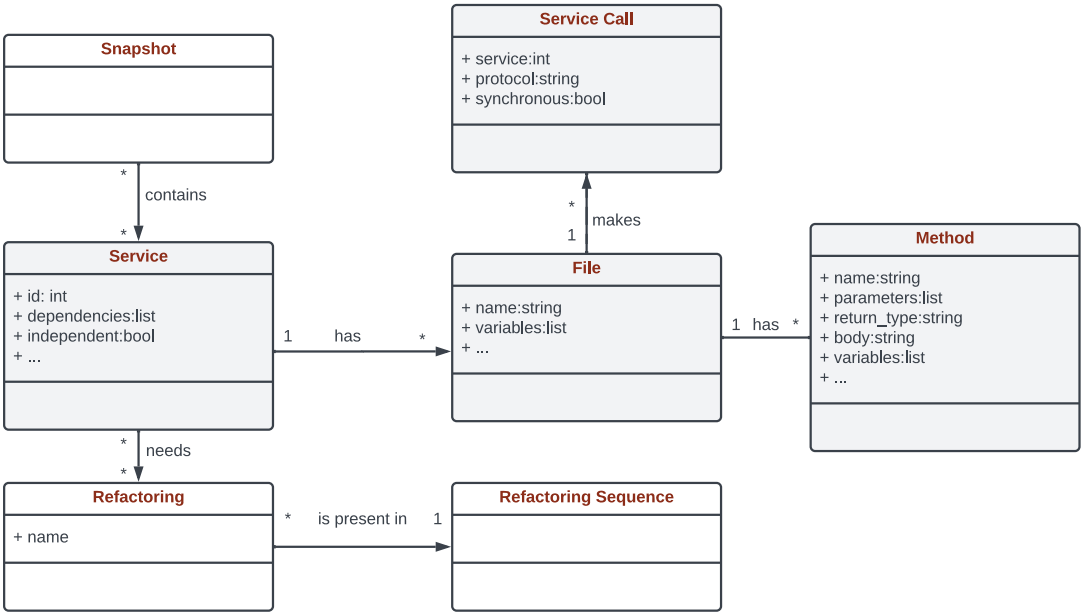


Figure 6.2: Conceptual Model of the approach

6.3 Refactoring Suggestion

To assist the developers in the refactoring process, we developed a strategy that, by analysing the conceptual classes detailed in the previous section and the dependencies between *Service*(s) and *File*(s), can suggest a sequence of refactorings the developer needs to apply to extract all services from the monolith.

This strategy uses as the highest level refactoring the *Strangler Fig*, owing to the fact that we want this refactoring process to be assisted and incremental, so each step is reversible, reducing the risks. We can think of each step of the Strangler Fig being an *Extract Service* refactoring. This way, we can start small and take it as an incremental process that should be adequately tested at every stage.

Some of the work necessary was to identify the sequence of smaller-scale refactorings present in higher-level refactorings and what steps were required to implement these smaller-scale ones. Therefore, we catalogued these refactorings (cf. Chapter 5). These catalogued refactorings are helpful so we can suggest the higher-level refactorings and smaller-level ones that compose them.

Furthermore, after knowing the dependencies between services, a critical decision is the order by which we perform their extraction. This should be the developer's decision as there are multiple strategies, depending on the developer's priorities. Some strategies could be to extract first the service with the lowest number of dependencies, extract first the service with the highest number of dependencies, extract first the one with the most database dependencies, perform a brute force algorithm trying all possible orders and choosing the one that leads to the least number of refactorings needed to extract all services, according to team availability and much more. In the given context, the selected strategy aims to order the extraction by the extractions that lead to more gain (in scalability either for the system or the team).

Thus, we identify the need for a specific refactoring to extract a service based on the dependency type and its presence in the system.

In an ideal scenario, multiple refactoring sequences should be proposed, all converging towards the same decomposed microservices. However, specific sequences may align better with the system's requirements and the trade-offs it is willing to make. These sequences should include multiple refactoring methods, including using intermediate refactoring states. At any given point, the user should be able to continue without assistance from that step on, and the system's state in that step should be readily available.

6.4 Refactoring Application

We implement the corresponding modifications in the system that are triggered by this refactoring, as described in the implementation details of each refactoring. These modifications manifest at multiple levels, including code (such as service calls and new methods), file organisation (introduction of new files), database schema, and queries, among others.

The process of performing these refactoring operations involves creating an abstract method that outlines the steps for each specific refactoring. These refactoring operations work with the abstract representation of the system, allowing for a higher-level view of the changes being made. When the refactoring to be applied is identified, the corresponding method is called and applies the transformation to the abstract representation of the system, gradually aligning it with a microservices architecture as each refactoring is applied.

6.5 Output

The final important part of this approach is to determine the output of this tool. The proposed output, to assist the developer in implementing a refactoring sequence, should be a representation of the monolith's dependencies to the intended microservices decomposition, a representation of the suggested refactoring sequence, and the evolutionary system representation. The evolutionary system representation would consist of various files with a system representation after applying a specific refactoring to understand its evolution better. Additionally, some visualisation of this output could significantly contribute to the success of this approach. In these situations, we expect visualisations like the big picture of the refactorings, where we can explore each step to understand how to implement the refactoring and how it affects our system. Other things that could be represented, such as dependency graphs, code metrics dashboards, change impact analysis, and much more, might be displayed. Still, we should concentrate on using visual aids to facilitate the migration process rather than providing too much information.

Chapter 7

MicroOnion: an Assisted and Incremental Refactoring Tool

This chapter describes in detail the tool developed as a prototype of the approach explained in Chapter 6. To assist developers in their refactoring process of migrating monoliths to microservices, we developed MicroOnion, a tool that provides assisted and incremental refactoring towards a microservice architecture.

With MicroOnion, we believe that, by providing assistance to the refactoring process incrementally, developers can be more confident in their migration and more productive and that we are contributing to the automation of the migration process without putting the system's quality at risk.

7.1 Scope

Monoliths are developed in the most various programming languages and frameworks, and many of the verifications we need to do to identify refactorings depend on the programming language. Therefore, we decided that we would have to focus on one programming language for this prototype and analyse how it could be generalised for all the possibilities.

As we mentioned before, we do not aim to start the migration from scratch but to focus on the refactoring step of the migration. Numerous works already propose decomposition strategies and identify the microservices boundaries, and we want to base our prototype on one.

When trying to figure out which tool to use to provide the input for our tool, we encountered a limitation of what projects to use to use as a test when building our prototype. Therefore, we resorted to using some of the same projects MicroRefact [36] collected through a search on GitHub. We did it because we were already familiar with MicroRefact and could access pre-generated proposed decompositions and source code representations. From all the projects MicroRefact retrieved, we focused on three with different levels of complexity, different number of files, and different number of proposed microservices: RestaurantServer, ProyectoUNAM and HotelManagementSystem.

As we chose those specific projects from MicroRefact, we narrowed the tool's scope to Java projects using the framework Spring and, consequently, an OOP paradigm. Additionally, we decided to focus on only one part of the refactoring process: breaking dependencies with the purpose of extracting services from a monolith. We decided to do so because we believe that in the first stage of migration, to extract services, it is necessary to break the dependencies of the component that will soon be a microservice to the monolith and the dependencies of the monolith to the component. Therefore, we reduced the scope of our tool to simply suggest the refactorings that allow the services to exist independently and then the developer shall implement other refactorings to make the most of the new architectural paradigm.

By breaking dependencies, we mean that, for instance, when Component A uses a method from Component B, we will break this dependency by changing the local method call to a service call, making B a remote service. The dependency is not broken but is refactored to a different type of dependency. These refactorings do not handle the additional problems of breaking this dependency, such as A's ability to function appropriately when B is unavailable.

Unlike Martin Fowler, that defines a component as *"a unit of software that is independently replaceable and upgradeable."* [62], we define a component here as a unit that has been recognised to become a microservice when the architecture is eventually decomposed to microservices but has yet to be extracted and so remains part of the monolith but will become independent of the others during the migration.

When comparing what we want to propose in contrast to MicroRefact, we suggest more vast refactoring types. Besides that, we want to build our tool to allow us to quickly add more refactoring types as soon as we can describe them correctly enough to understand how it affects the system. Even though MicroRefact works on actually implementing the refactoring, we believe our tool is valuable since the developer is being guided through the migration and can perform it whenever they wish and have the resources to. Additionally, the developer can migrate the system incrementally at their own pace and is guided step by step, where they can assess the system's evolution so far. And we believe this assistance accommodates the industry's current needs. As has been proven before, the developers need adaptation to the automation of the process of architectural refactoring. Therefore, our tool is a significant step towards this automation but straightforward enough for developers to adapt to this transition.

7.2 Overview

This tool is divided into two major parts: the refactoring sequence suggester and the visualisation. The first performs the approach mentioned in the previous chapter (Chapter 6), and the other visually represents the suggested refactoring sequence.

The refactoring suggester was developed using *Python*[1] and it only deals with *JSON*[2] files, either for input or output.

[1] Python.org, https://www.python.org/ (accessed Jun. 10, 2023).
[2] JSON, https://www.json.org/json-en.html (accessed Jun. 10, 2023).

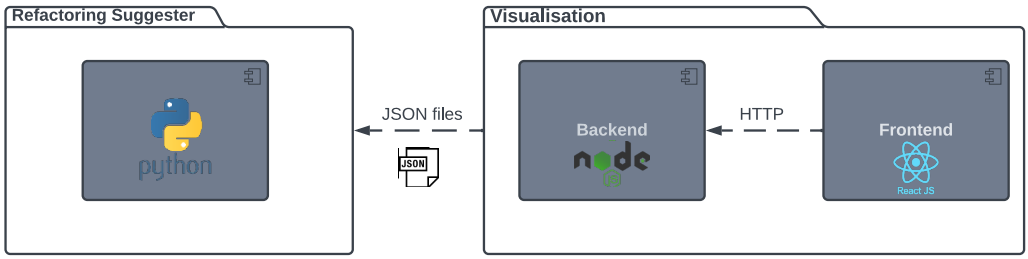


Figure 7.1: System's component diagram

The website visualization was created using *React*[3] for the frontend and *Node.js*[4]to the back-end. These two communicate via HTTP requests. Figure 7.1 represents the component diagram of our system.

The tool was designed to iteratively extract a service by breaking the component dependencies to the monolith and of the monolith to the component. With the two elements of the tool, the developer can access its output of the suggested sequence of refactorings to extract all services and have visual guidance on how to perform it. Therefore, we created both a command-line-based tool and a website tool.

Figure 7.2 shows the flow of the part of the refactoring sequence suggester.

Using some of the same projects as MicroRefact allowed us to access the input files mentioned in the previous chapter (cf. Section 6.1), which are necessary for the tool. The way they are obtained and their format is further explained in Section 7.3. Following the detailed approach, we determine each service's dependencies and gather all the received information in the data structures used for the system's internal representation.

After that, we order the services to extract from the least to the most coupled (from the least to the most number of dependencies), which is strategies chosen as default to decide this order.

Continuing the tool's flow, we enter the iterative and step-by-step stage of the tool. For each microservice to be extracted, we suggest the refactoring sequence to extract it and apply these refactorings to the internal system's representation. This is iterative because we are at each step only applying the changes necessary to extract that service, and the user may want to extract only the two first services before it has the means to continue with the rest of the migration. After extracting all services, the output files are generated, representing the refactoring sequence and the snapshots of the system's evolution. Even though we wish to deliver the best tool possible and automate the entire process of refactoring, the tool should be flexible enough to allow the user at any point to take the refactoring from there and perform it as they wish. Therefore, a new project version can be created and exported at each step of the refactoring sequence.

This tool is freely available to get suggestions and allow people to take our work as a reference for theirs.

We express further details on the tool's implementations in the following sections.

[3]React, https://react.dev/ (accessed Jun. 10, 2023).
[4]Node.js, https://nodejs.org/en (accessed Jun. 10, 2023).

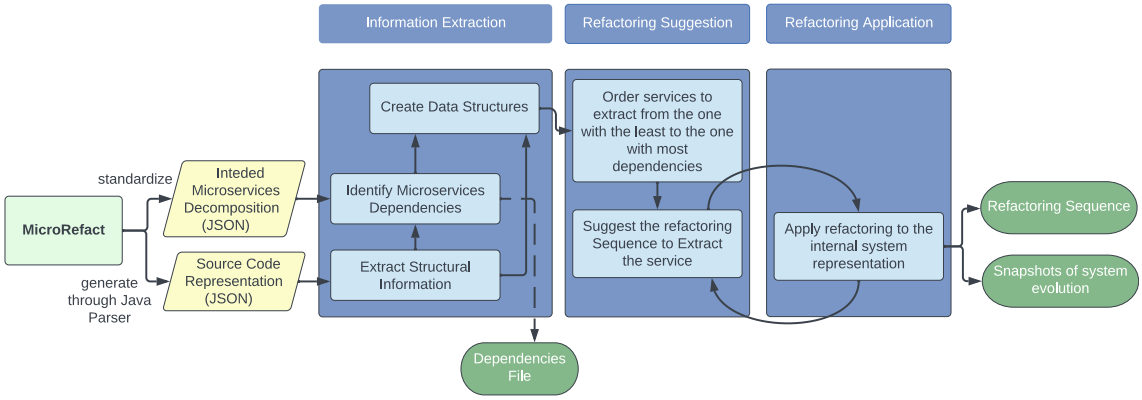


Figure 7.2: Flow of the tool

7.3 Information Extraction

We stated before that we used some projects collected by MicroRefact as the projects to guide the design of this tool. However, our input files differ from the input MicroRefact used and its output.

7.3.1 Input Files

To obtain the source code representation, we resourced to MicroRefact, which created an AST by implementing the Java Parser [5] library. With this library, it is possible to create visitors to each type of entity in Java and collect their information.

We did not collect the output directly from their implementation of the Java Parser. We also took advantage of some cleanup MicroRefact did when constructing their internal representation and worked with MicroRefact's code to dump a JSON file with the information held on each class. This way, we obtained the source code representation without unknown characters and unnecessary attributes. MicroRefact added its own attributes, but our tool later discarded them.

The expected format of the JSON file with source code representation's content is presented through an example in Appendix in Listing D.1 with only the relevant content for our tool. In this example, we can see all attributes we expect to know from a file, and therefore, our tool will work with any system that can represent its files in these attributes. These attributes are for each file the full name of the file, its constructor, the package it belongs to, its short name, annotations, instance variables, methods, classes it implements, classes it extends, other files it imports, its method invocations and if it is an interface. Each instance variable has its annotations, modifier, identifier, type and the variable name. Each method has the name, annotations, return data type, identifier, parameters data type, variables, body and method invocations. Each parameter data type has the type, the variable and the qualified type. Each method invocation has the scope name, the method name and the target class name.

MicroRefact received as an Inteded Microservices Decomposition, a file resulting from running the tool developed by Brito et al. [13] on the desired system. We accept that MicroRefact

[5] https://javaparser.org/

used this tool with the default parameters. However, as we wanted to standardise this input to be possible to produce by any tool that detects service boundaries and produces a proposal of division into microservices, we made some changes in the schema of the object being used.

The resulting file of the Brito et al. [13] tool for the RestaurantServer project can be seen in Appendix D.2.

As all we need from this file is the *clusterString* attribute, we standardised the input file of the intended microservices decomposition to contain only the microservice identifier and the files that belonged to that microservice. An example of part of that file for the RestaurantServer project can be seen in Listing 7.1.

```
{
    "0": [
        "pl.edu.wat.wcy.pz.restaurantServer.security.WebSecurityConfiguration",
        "pl.edu.wat.wcy.pz.restaurantServer.security.jwt.JwtAuthEntryPoint", ...
    ],
    "1": [...]
   ...
}
```

Listing 7.1: Standardized input of intended microservices decompositions exemplified for the ResturantServer project

7.3.2 Internal Representation

From the source code representation, we extracted the structural information of the system and started to create the data structures that we used as our internal representation of the system.

Figure 7.3 represents the class diagram used for the system's internal representation. In this class diagram, we see the objects created and their relationships. The main class is *Service*, which has many instances of *Class* and can make several *Service Call*(s). A *Class* has many *Method*(s).

7.3.3 Microservices Dependencies Identification

After extracting the structural information from the source code representation and the intended microservices decomposition, we can identify the microservices dependencies and create the previously mentioned data structures.

To analyse the dependencies of each service, we start by analysing the dependencies of each class of the service and storing them in a map of the form `{className: dependencies}`, where *className* is the name of the file of the class and *dependencies* is an array with the dependencies of that files where each element has the following structure: `[file, type1, ..., typeN]` where *file* is the name of the file of the class it has a dependent to and *type* 1 to N are the types of dependencies it has with each file.

After retrieving all dependencies of that service's files, we need to understand which dependencies are external (dependencies to another service). While we do it, we construct the object

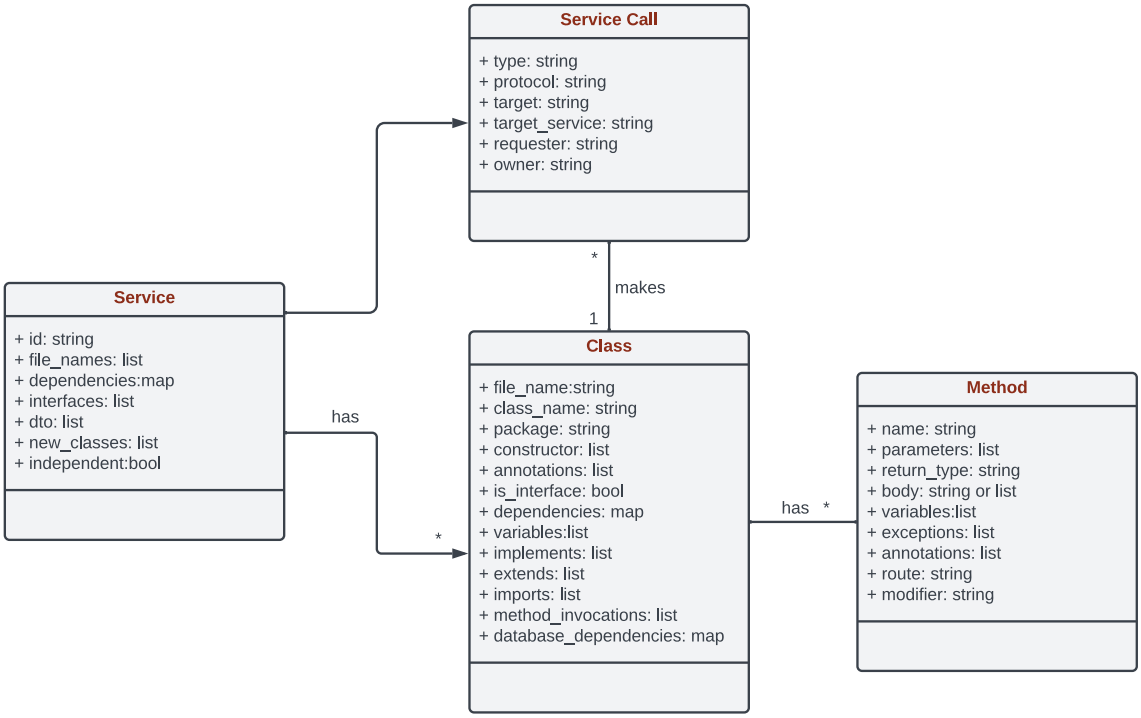


Figure 7.3: Class Diagram used for the internal representation of the system

representing the service dependencies. This object is supposed to store the dependencies of that service, including to which service, to which files of that service, which file from this service has dependencies to that file and what types of dependencies it has. Therefore, we expect that object to have the schema shown in Listing 7.2.

```
dependent_microservice: {
    file: [ [ dependent_file, dependency_types], ... ],
    ...
}
```

Listing 7.2: Service dependencies object

To obtain each class's dependencies, we check the files it extends and implements and add them all as dependencies. Then, we check all method invocations and add to dependencies all that belong to the system (whose *targetClassName* is one of the system's files). After that, we always add types of variables that are not primitive. Then, we iterate all methods and run a similar algorithm to see if their return type, variables or parameters are not primitive and if not, we add them as dependencies. Lastly, for the chance of not being caught in previous verifications, we add all files that are imported and are from the system but aren't yet on this class dependencies. These dependencies are also added to the database dependencies.

All these dependencies are added in the format mentioned, and the possible types are "implements", "extends", "methodInvocation", "variableType", "methodVariable", "imports", and "databaseDependency" by the order they were described in the previous paragraph.

The identification of database dependencies differs from what we have seen in the other cases. Annotations in the variables identify the database dependencies and are very dependent on the programming language and framework of the project we are using. We have four possible types of relationships between two database entities: *OneToMany*, *ManyToOne*, *ManyToMany* and *OneToOne*. These appear through different annotations before the variable declaration, as the following:

- *OneToMany*:
  - Owner of the relationship:
    `@ManyToOne`
    `@JoinColumn(name="book_id")`, where "book_id" is the foreign-key.
  - Parent of the relationship:
    `@OneToMany(mappedBY=" id")`, where "id" is the primary key of the parent.
- *ManyToOne*:
  `@ManyToOne`
  `@JoinColumn(name="book_id")`, where "book_id" is the foreign-key.
- *ManyToMany*:
  - Owner:
    `@ManyToMany()`
    `@JoinTable(name=, joinColumn=, inverseJoinColumn=),.`
  - Owned:
    `@ManyToMany(mappedBy= "id")`, where "id" is the primary key of the owner.
- *OneToOne*:
  - with foreign key:
    * Owner:
      `@OneToOne(...)`
      `@JoinColumn(name="", referencedColumnName="")`
    * Owned:
      `@OneToOne(mappedBy="")`
  - with shared primary key:
    * Owner:
      `@OneToOne(mappedBy="")`
      `@PrimaryKeyJoinColumn()`
    * Owned:
      `@OneToOne()`
      `@MapsId`
      `@JoinColumn(name="")`

Therefore, we created a representation of the database dependencies for each type of relationship:

- *OneToMany*: `[OneToMany, TableOwningTheRelationship, ForeignKey]`
- *ManyToOne*: `[ManyToOne, JoinColumn, OtherTableColumn]`
- *ManyToMany*:`[ManyToMany, OtherTable, JoinTable, ColumnInThisTable, ColumnInOtherTable]`
- *OneToOne*:`[OneToOne, OtherTable, ColumnInThisTable, ColumnInOtherTable]`

After determining the external dependencies of each microservice, these are written to a file, "dependencies.json", whose content is exemplified in Listing 7.3.

```
{
"1": {
        "5": {
            "User": [["pl.edu.wat.wcy.pz.restaurantServer.entity.Reservation", "
                methodVariable", "databaseDependency]],
            "UserService": [["pl.edu.wat.wcy.pz.restaurantServer.entity.Reservation", "
                methodVariable"]],
        },
    ...
    }
}
```

Listing 7.3: Dependencies file example

In this example, service 1 has a dependency to service 5 in the file *User* to the file *Reservation* of type *methodVariable* and *databaseDependency*.

With the dependencies determined, we can join this information to our internal representation and proceed to the refactoring suggestion.

7.4 Refactoring Suggestion and Application

As we can see in the tool flow presented at this chapter's beginning, this stage is circular because we propose performing iteratively. To follow the Strangler Fig approach of reversible steps to reduce the risks, we extract service by service, suggesting the refactoring sequence to extract a service and applying those refactorings to the system's representation.

Before suggesting the refactoring sequence, we need to order the services in the order we want to extract them. From all the possible strategies to decide this order, we concluded that ordering by the ones with the least dependencies made more sense to this Strangler Fig approach as, ideally, fewer dependencies needed to be taken care of, and fewer changes would be performed on the system.

To extract each service, we focus on breaking the component dependencies to the monolith and the dependencies from the monolith to the component. In this suggestion, we are logically only using the refactorings from the "Breaking Dependencies" Chapter (cf. Appendix C.1) of the Catalogue of Refactorings, plus the "Change Data Ownership" refactoring because it is very common when breaking dependencies.

The suggestion of the refactoring sequence to extract a service starts by analysing its dependencies. Inside the microservices dependencies, we perform another order, this time of the types of dependencies to treat first. This might seem odd at first, but if we remember the catalogue of refactorings, many of the refactorings are a consequence of prior refactoring and, therefore, at a lower level, so it is inside the sequence of the parent refactoring. We order database dependencies to be treated first, then variable types and method variables, method invocation, and interface implements, extends and imports. The database dependency can lead to a variable type or method variable dependency and, therefore, should be treated first to reduce the number of changes of doing it all the way around. The same occurs for the rest of the possibilities.

Once we have that order, we can easily map the dependencies to the refactorings:

- *databaseDependency*: the database dependency can have, at the highest level, a "Change Data Ownership" refactoring because if a join table was defined, the microservice is the data owner. Whether it is the case or not, the corresponding refactoring is "Move Foreign-key relationship to code".

- *methodInvocation*: is a "Change local method call dependency to a service call". This can be a second-level refactoring if variable types or method variable dependencies are on the same file.

- *variableType*, *methodVariable*: in these cases, we have a "Break data type dependency" refactoring that can very likely need a "Create DTO" refactoring if there is not a DTO for that data type yet and "Change local method call dependency to a service call" if there are method invocations of that data type.

- *import*, *extends* an *implements*: these correspond to a "Duplicate file Across Microservices" refactoring.

When we map the dependencies to the refactorings, we apply the refactoring to the system's representation according to the steps detailed in the Catalogue of Refactorings. These refactorings are not applied at the code level but at the file organisation level and relationships between classes/services. When these refactorings are used, we avoid repeating steps like creating existing files. Every time a new service is extracted and left with no dependencies, it is tagged as independent, and a snapshot of the current state of the system is created so that we can keep track of the system's evolution.

To accommodate the refactorings, snapshots, etc., our domain model expanded as seen in Figure 7.4. In this complete domain model, five new classes can be seen:

- *Codebase*: that represents the codebase and therefore is mapped as a list of *Class*(es).
- *Service Decomposition*: that clusters information in the input file of the intended microservice decomposition and is mapped by a list of *Service*(s).
- *Refactoring*: represents each refactoring, and it stores its id, name, level, the starting microservice, the microservice it depends on and the notes of applying that refactoring (what files are created, what service calls are changed, etc.).
- *RefactoringSequence*: a list of refactorings to extract all project services.
- *RefactoringRepresentation*: what we previously called snapshots of the system that have a version of the system and its service, whether they are already independent of the monolith.

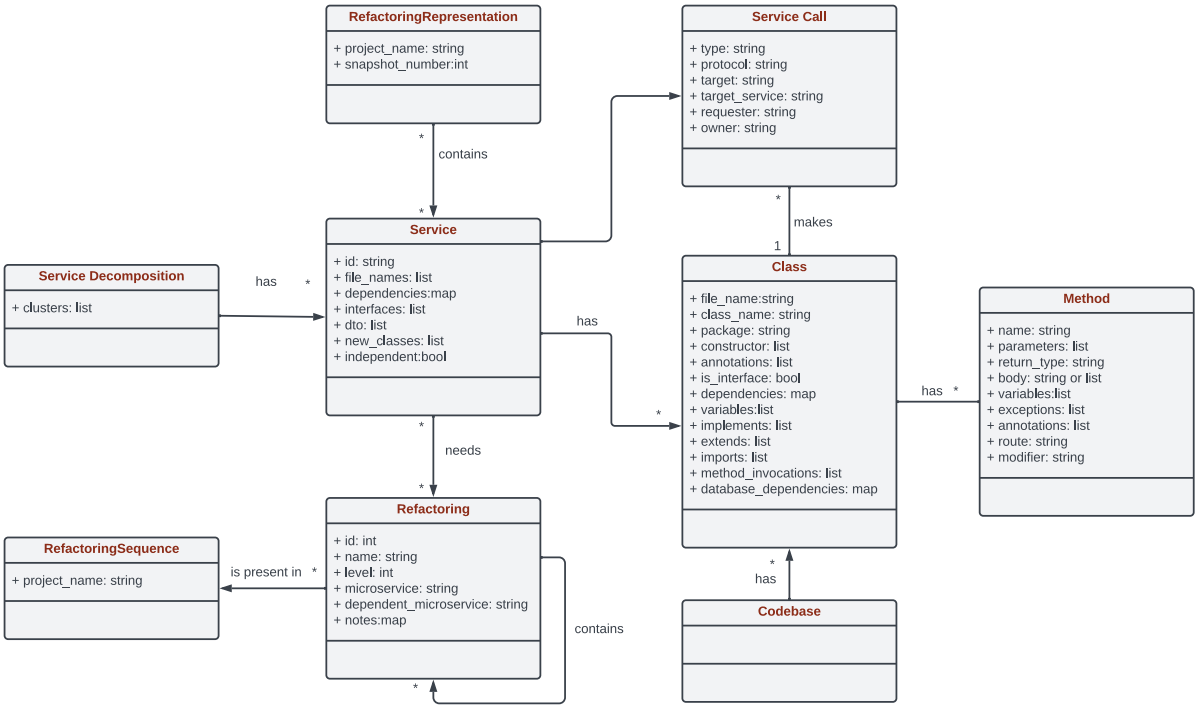


Figure 7.4: Tool's complete domain model

7.5 Output of the Refactoring Suggester

The last step of the tool's flow is to generate its output. When deciding what the result should be, we kept in mind this tool's goals: to provide a sequence of refactorings to allow the developer to iteratively perform the refactoring process on migrating a system to a microservices architecture.

Therefore, the output of this tool is three types of files, two of a unique kind and one with multiple instances. The first two are the dependencies file (whose structure was shown earlier in this chapter) and the file with the refactoring sequence.

An example of the structure of that file can be seen in Listing 7.4. It dumps to a file the object *RefactoringSequence* from our domain model.

```
{
    "project_name": "restaurantServer",
    "refactorings": [
        {
            "id": 1,
            "name": "EXTRACT MICROSERVICE",
            "level": 1,
            "microservice": "0",
            "dependent_microservice": "-1",
            "notes": {},
            "refactorings": [
                {
                    "id": 2,
                    "name": "BREAK DATA TYPE DEPENDENCY",
                    "level": 2,
                    "microservice": "0",
                    "dependent_microservice": "1",
                    "notes": {
                        "file": "WebSecurityConfiguration",
                        "dependent_file": "pl.edu.wat.wcy.pz.restaurantServer.security.
                        service.UserDetailsServiceImpl",
                        "dependencies": [
                            "variableType"
                        ]
                    },
                    "refactorings": [
                        {...(the same as this example)}
```

Listing 7.4: Example of refactoring sequence output file

The other file with multiple instances is the *RefactoringRepresentation* and, therefore, the representation of the system evolution. There are the number of services plus two instances of this file, as the first is the monolith representation, and the final is the representation of the microservices architecture. These files contain the project's name, the snapshot number and then the list of all the services of the system in an object representation having all the information of the *Service* class from the domain model. An example of this file is available in Appendix D.

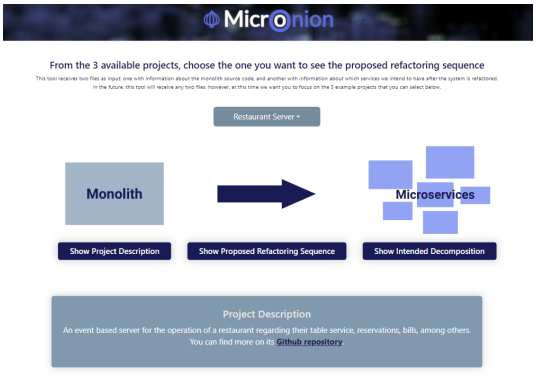
7.6 Visualization

Another major part of our tool is the visualisation that the creation of the website has accomplished. With this website, we want to combat the lack of visual feedback and assistance. Due to the fact that we want to assist the refactoring through the suggestion of a refactoring sequence that has refactorings of large-scale and smaller-scale refactorings and their steps, we called this tool MicroOnion as a metaphor for peeling the layers of the onion until we get to a level we can simply implement it. Our intention with this tool is to provide step-by-step migration support with

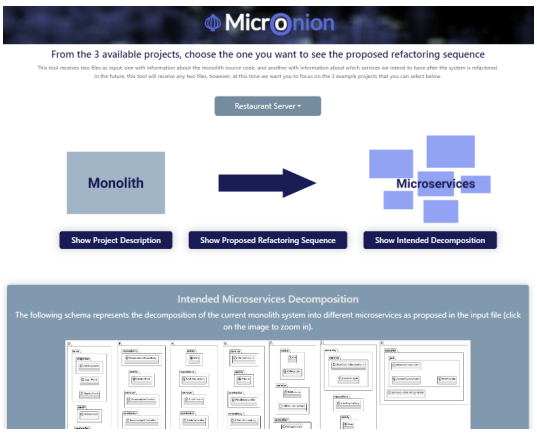
visualisation of the system's evolution, reducing developers' concern of not understanding what is happening to their system.

For this part of the tool, we want to provide more than just the refactoring suggestion. Therefore, we can see on the home page 4 different buttons that explore different migration categories that do not follow any particular order and whose refactorings can be applied in whichever order makes sense in the project at hand. These categories are: *Extract Services*, *Infrastructure Improvement*, *Deployment & Orchestration* and *Check Microservices Architecture Characteristics*. These last three display information common to Chapters 2 (cf. Appendix C.2), 3 (cf. Appendix C.3) and 4 (cf. Appendix C.4) of the Catalogue of Refactorings and serve the purpose of showing more information on the other parts of the migration that our suggester does not include at the moment.

To experiment with the *Extract Services* category, we included the three projects collected by MicroRefact, as previously explained: *RestaurantServer*, *ProyectoUNAM* and *HotelManagementSystem*. When clicking these categories button, the user is directed to a page where it can choose the project whose refactoring sequence for extracting the services it wants to analyse in the dropdown. The user can access the project's description and the intended decomposition to support this choice, as shown in Figure 7.5.



(a) Project's Description



(b) Intended Decomposition

Figure 7.5: Choose Project Page

Check the figure on a larger scale in Appendix D.4

After choosing the desired project, they can click on the *Show Proposed Refactoring Sequence* button that will redirect them to the page that displays the order by which we propose the extraction of the services, from least to most coupled. Figure 7.6 shows this page for the *Proyecto UNAM* 's project.

By clicking on each button on the *Extraction Sequence*, the user is redirected to the *Extract Service* page, where all the information to extract the service is presented. On this page, we are presented with a similar representation to the *Choose Project* page, where we can see the initial state being the Monolith and the final state being the Monolith and another extracted service. By clicking the "Check component's initial state" button, the user can see the current state of this

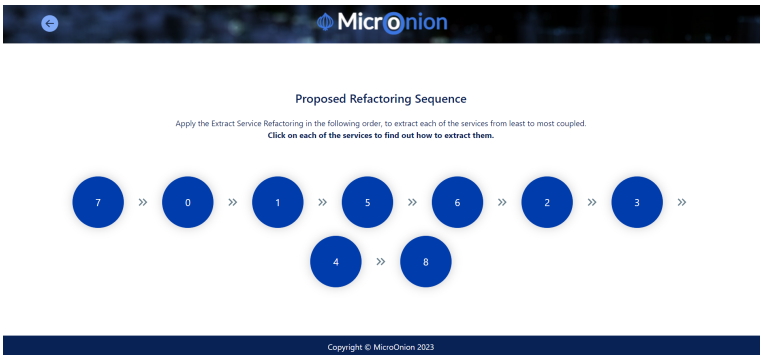
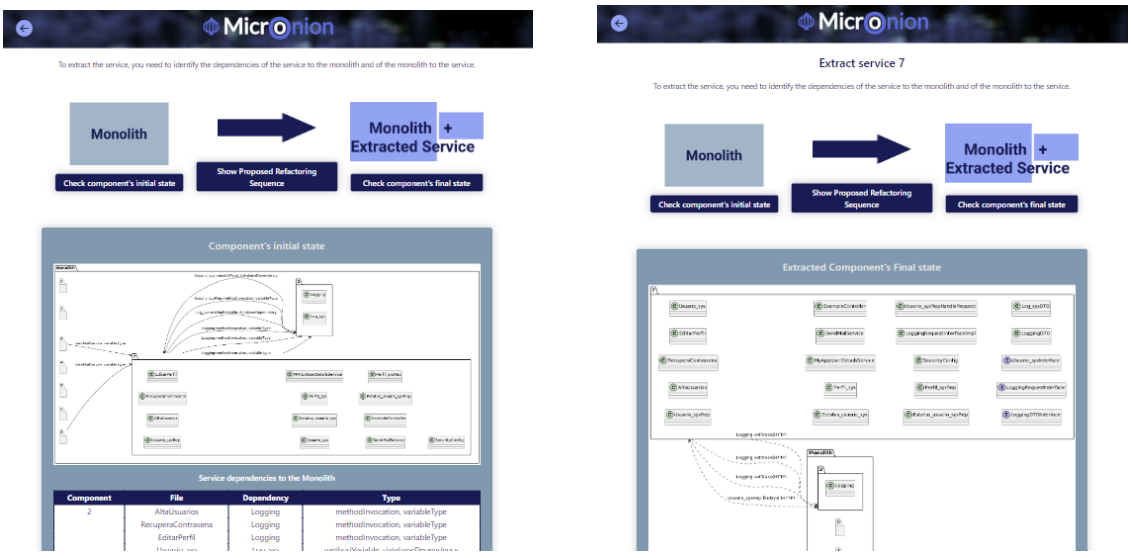


Figure 7.6: Project's services extraction sequence

Check the figure on a larger scale in Appendix D.5

component before being extracted to service, its files and its dependencies to the other components in the monolith. We never expect this component to have dependencies to the external services that were not already handled because each time a service is extracted, we break its dependencies to the monolith and the dependencies of the monolith to this component. By clicking the "Check component's final state" button, the user can see the representation of the system after extracting the service and how the dependencies were mitigated. Figure 7.7 shows these two views of the system for the extraction of service 7 of the *ProyectoUNAM* project.



(a) Component's initial state

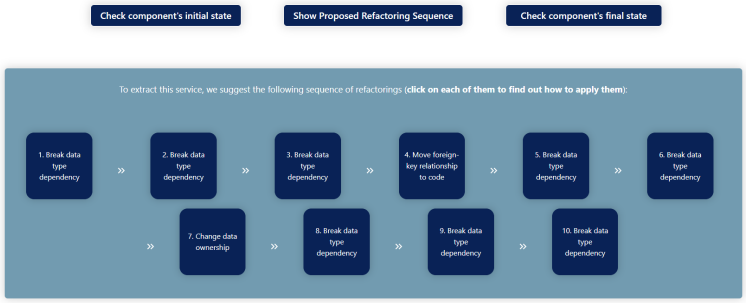
(b) Component's final state

Figure 7.7: Extract Service Page - Component's initial and final state

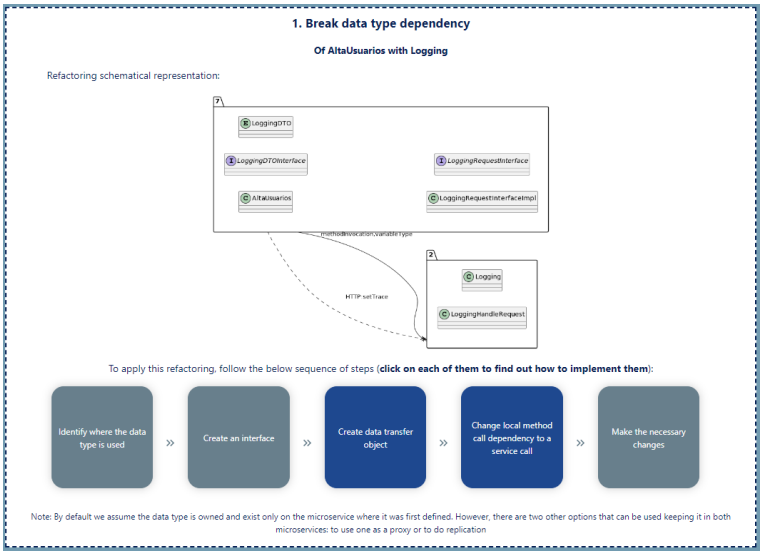
Check the figure on a larger scale in Appendix D.6

The middle button, which leads the way to extract the service, *Show Proposed Refactoring Sequence*, shows us the refactorings order by the way they should be applied to extract the referred

service. By clicking on each of these refactorings, the user is handed out the schematic representation of the refactoring implementation and the sequence of steps to follow to implement it. These steps can be simple steps or another refactoring, which are also clickable to a similar view. These refactoring implementation usually also contains some notes of optional ways to implement the refactoring or things to keep in mind. Figure 7.8 shows these two views mentioned.



(a) Project's Description



(b) Intended Decomposition

Figure 7.8: Choose Project Page

Check the figure on a larger scale in Appendix D.7

We generated all the schematic representations of the system evolution automatically using PlantUML[6]. To construct these automatically, we had to create a visual language to visually represent the refactorings and the system's state before and after extracting each service.

When representing the system before extracting a service, we represent the components still inside the monolith, that service files and its dependencies to and from the other components in the monolith, as well as the services already extracted. Figure 7.9 shows an example of this representation before extracting service 1 of the project *Restaurant Server*.

[6]"Open-source tool that uses simple textual descriptions to draw beautiful UML diagrams.," PlantUML.com, https://plantuml.com/ (accessed Jun. 10, 2023).

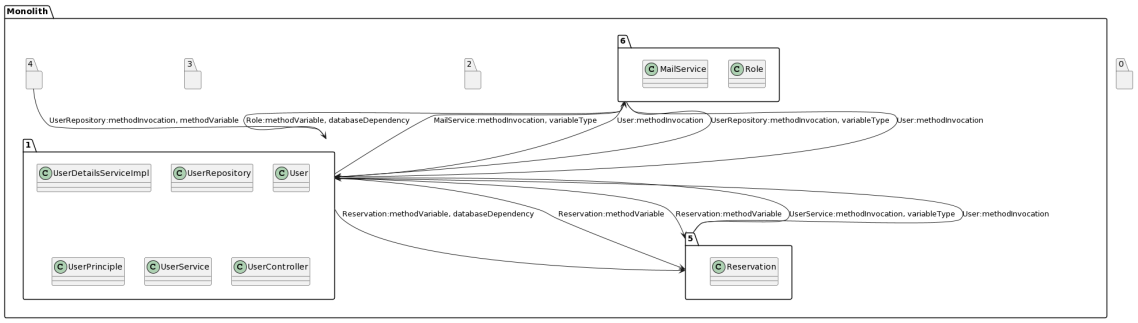


Figure 7.9: System before service 1 extraction

Check the figure on a larger scale in Appendix D.8

When representing the system after extracting a service, we once again represent the monolith and the components still inside the monolith. We represent the extracted services outside the monolith and the newly extracted services' new interactions with the other services and the components inside the monolith. Figure 7.10 shows an example of this representation after extracting service 1 of the project *Restaurant Server*. We assumed there were no service calls in the monolith.

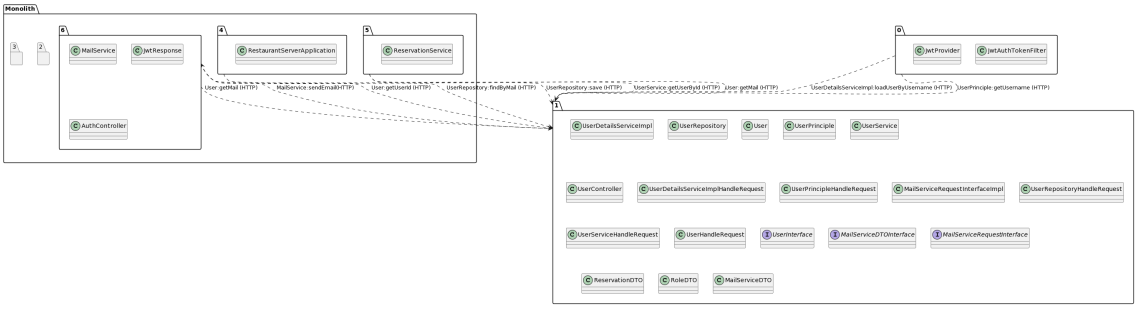


Figure 7.10: System after service 1 extraction

Check the figure at larger scale in Appendix D.9

For each refactoring we present in this tool, we also created a visual representation for it:

- Change local method call dependency to service call: here, we represent the creation of the new interfaces and classes as well as the new relationship with a service call, its protocol, method, etc.(Figure 7.11). The PlantUML code behind the schematic representation is available in Appendix D.10.
- Move foreign-key relationship to code and Change data ownership: represents the previous relation, the newly created interfaces and how these services will now communicate about these two entities through service calls (Figure 7.12). The PlantUML code behind the schematic representation is available in Appendix D.11.

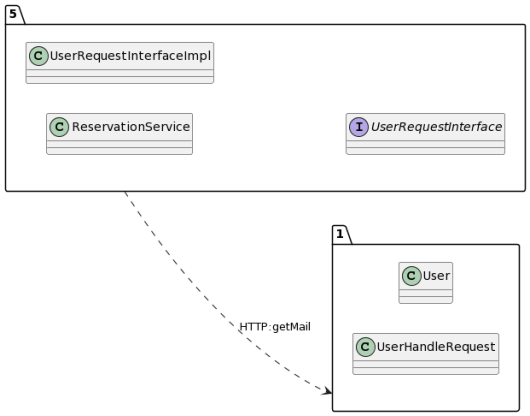


Figure 7.11: Change local method call dependency to service call schematic representation

Check the figure at larger scale in Appendix D.10

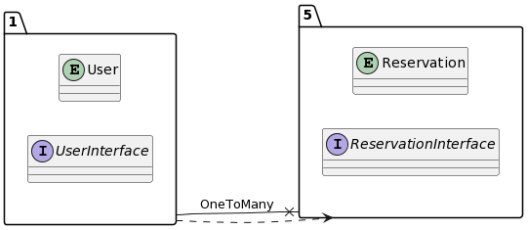


Figure 7.12: Move foreign-key relationship to code schematic representation

Check the figure at larger scale in Appendix D.11

- Break data type dependency: represents the previous data type dependency, the newly created DTO and, when needed, the interface created to access the DTO. The schematic representation and the PlantUML code behind it are available in Appendix D.12.
- Create data transfer object: show the creation of the data transfer object in the microservice that needs it. The schematic representation and the PlantUML code behind it are available in Appendix D.13.
- File/Import dependency: shows how we duplicate the file for the microservice that needs it. The schematic representation and the PlantUML code behind it are available in Appendix D.14.

7.7 How to Use

Our prototype source code is available at github[7]. There we have two folders, one for the refactoring suggester (*migration_tool*) and other for the visualisation (*visualization_tool*).

[7] https://github.com/SoftwareForHumans/MicroOnion

To run the refactoring suggested, we execute the following command on the *migration_tool* folder: `python main.py [filePath1] [filePath2] [projectName]`, where the arguments are the path to the file with the project's source code representation, the file to the project's intended decomposition representation and the project's name.

To run the visualisation part locally, it is necessary to run both the front and backend. The version of the packages used on either can be seen on the "package-lock.json" available in each folder. To run the frontend, run the following commands: `npm install && npm start`. To run the backend, run: `npm install && npm run dev`. The diagram visualisation available on the tool was generated using PlantUML. However, we used its online server instead of creating a server on our system.

The visualisation website is also available at:
https://feup-microservices-assisted-refactoring.vercel.app/.

7.8 MicroOnion's Current Limitations

MicroOnion is a prototype of the approach mentioned in the previous chapter. Therefore, because of the restricted timeline, it does not implement the proposed strategy completely and has some limitations.

The current limitations of MicroOnion and the future work necessary to maximise this tool are:

- **Technologies:** As mentioned, MicroOnion only accepts projects in Java using the Spring framework, which embraces an OOP paradigm. This is the first limitation of our tool, as there are various possible technology stacks the monolith can be built on. But the goal for this tool is to be broad enough that it can function with any programming language or framework with the necessary abstractions.

- **Validation:** Currently, MicroOnion does not validate the files' structure, which should be done for safety. However, as this may not be definitive due to the previous sections, we decided it was not a priority. Additionally, we do not recognise entities or interfaces when creating the system's representation, treating everything like files.

- **Input:** For now, we receive as a source code and database model representation a file similar to an AST. This poses a limitation because although it is valid, it may take a lot of work to obtain it, and much of the information may end up being discarded. Nevertheless, the necessity arose from needing to know the type of all the variables used throughout the projects, the return types of methods, etc. Besides, it contains simply information about the code, which can give us a wrong notion of the dependencies' strength. Information on the system in runtime could provide us with more insight, which could be obtained with dynamic analysis [51].

- **Assumptions:** We follow the premise that to extract a service at the first stage of migration, it's necessary to break its dependencies to the monolith and of the monolith to this service. This is not totally accurate, as this assumes that other systems failing is not the service being extracted problem and much more.
- **Databases:** We focused on relational databases, so some issues that are more specific to non-relational databases may not be included in the scope of this tool, even though the relational/non-relational databases have similar limitations. We are also focusing on projects that use an ORM; when not, we can expect significant differences in how the refactoring is performed.
- **Database annotations:** We do not deal with database annotations in the methods, which is possible to occur, but very specific and for that reason, we did not have cases to test, so it was not performed.
- **Synchronous vs Asynchronous calls:** Unfortunately, we cannot decide if a call should be synchronous or asynchronous because we need more data like production logs or flow diagrams.
- **Reflection:** In the cases of reflection, we do not yet have a way to identify it and deal with it accordingly. In these cases, we could detect imports of files of the project that were not caught in the other types of dependencies. Currently, if it is a dependency of a service, it is being solved by duplicating the file to the microservices that use it.
- **Systems' representation:** We are aware that the systems' representation presented in our tool does not correspond to the best way we can display the information. However, we focused only on delivering class diagrams. Database models and other information could assist the migration.
- **Output:** The current output could also be improved to be a single file, and the structure of the current files may not be ideal.
- **Identifying refactorings:** Some refactorings can not be identified because there is currently insufficient information to allow us to integrate them into the refactoring sequences. Therefore, we only present the refactorings that we can identify. More refactoring from the catalogue can be later implemented in our tool.
- **Class repetition**: It should handle cases with the same class in multiple microservices and classes with the same name.
- **User input:** Lastly, ideally, we would have used the user input throughout the process, so some choices of how to implement the refactorings (for example, if a service call is synchronous or asynchronous, data management decisions) could be made by the developer and following its business requirements.

- **Ensure the correctness:** We do not ensure the correctness after the refactoring, at it is out of the scope of our work.

Most of these limitations were not mitigated due to the time constraint and did not pose threats to the prototype validation.

Chapter 8

Empirical Evaluation

To evaluate the contributions of this dissertation, a catalogue of refactorings and a tool to assist the incremental migration towards a microservices architecture and our approach, we conducted an empirical study. This chapter describes the methodology used and discusses its results and main threats to validity.

8.1 Goals

This study aims to evaluate the refactorings, which we present as refactorings to "break" dependencies and assess our tool's ability to provide meaningful assistance to developers.

With that in mind, we aim with this study to find ways to answer the proposed research questions of this dissertation:

- **RQ1.** To which extent can known microservice migration techniques be described as sequences of smaller-scale refactorings?

  We expect to improve our catalogue with the results of this study. Therefore, we would like to understand if the catalogue is able to help to apply a refactoring in the most common cases, if the mechanics are descriptive enough for a developer to implement it and if the example clearly shows how it works. Furthermore, we would like to understand if the catalogue can successfully describe migration techniques as a sequence of smaller-scale refactorings and assist practitioners.

- **RQ2.** To what extent can we systematically generate a refactoring-based plan to migrate to a microservices architecture given a desired decomposition?

  With this question, we want to assess our tool and investigate if the respondents trust its output to suggest how to carry the decomposition, if they can think of other ways to sort the services for the extraction and if they are familiar with the procedures mentioned in the tool.

- **RQ3.** Is it possible to assess the impact of a refactoring sequence on the system's evolution step by step?

The visualisation aspect of our tool is, in our opinion, as essential as its output, as it provides a step-by-step assessment of the system's evolution. Hence, we examine if it brings more trust to developers, if it assists the developers and if having more information than just the extraction of the services helps to improve the user's perception of the tool.

8.2 Design

The empirical validation was conducted as a qualitative survey [80].

We interviewed participants with a semi-structured survey in which the questions were given in the form of a survey prepared on Google Forms (available in Appendix E). From there, we explored any questions that may have come from the participants' answers to provide additional insight into the respondents' responses. We chose this format because we wanted to provide more guidance to respondents throughout the study, as it was a lot to digest at first, and there was the risk that parts of the survey were perceived as confusing. In other words, a quantitative survey might have rendered less reliable answers in case participants did not understand what was asked, and interviews might easily deviate from our research objectives.

The expected time of completion was between forty-five minutes to one hour.

The survey was organised into three (3) sections:

- **Part 1: Experience and Background (3 min).** This section was identical to the first survey performed in this dissertation because we wanted to leave the option to cross the data from both surveys. In addition to these questions, we added another one to know if they were familiar with any tool that assists in the refactoring towards a microservice architecture.

- **Part 2: Refactorings Implementation (20 min).** This section was regarding the catalogue of refactorings. We wanted to know if the respondent agreed with the mechanics and examples provided for each of the five refactorings presented. The examples presented do not correspond to the examples in the catalogue but are schematic examples similar to the ones presented in the tool.

- **Part 3: MicroOnion: A tool for assisting the refactoring to microservices (10 min).** This last section concerned the tool developed, how the user perceived it, its level of trust, feedback on how to improve and other strategies to order the service extraction.

The complete survey used to guide the interview can be found in Appendix E

8.3 Sample

The sample for this study was constructed via the dissemination of emails to personal network contacts. The objective was not to acquire a large sample size, as we were designing a semi-structured interview for the study. Instead, we aimed to gather a sample characterised by significant expertise and availability for a lengthy participation. We did not aim to evaluate metrics like coupling,

cohesion or even modularity, so we wanted to know how practitioners and teams felt about MicroOnion. Hence, we use the practice as the baseline of our study and validate it by asking the experts who know how it is currently done. Therefore, we restricted the sample to people currently working in the industry and who have participated in at least one migration. The study started on 2023-06-01 and lasted until 2023-06-15. Unfortunately, we only interviewed two respondents due to the specific degree of experience we sought. It is also a holiday season, making it more challenging to find eligible people available.

8.4 Data Analysis

In this section, we report our analysis and the insight by the sections we divided the survey into.

This data analysis is qualitative and focused on generating insights from the participation of our two respondents.

Experience and Background

This section characterises our sample in their experience, background and demography.

Both our respondents work in Portugal, and we have one respondent in the area of Software Development and Architecture and one in Software Architecture, Operations, Quality Assurance and Infrastructure. It is beneficial that both participants can provide perspectives from different angles, considering their respective profiles. One is a company's CIO & CTO, and the other is a Senior Developer. They both have around four to five years of experience in microservices. However, the CTO has migrated over twenty projects to microservices and the Senior Developer two. As for the projects they work on, the CTO works on retail and supply chain management, the projects have between one hundred thousand to one million users, and the project's team was between ten to fifty elements. The Senior Developer works on supply chain management in projects of one hundred to one thousand users and teams between ten to fifty elements. Neither of the participants is familiar with any tool that assists in the refactoring towards a microservice architecture.

Refactorings Implementation

In this section, we will go through each refactoring and analyse the responses of the participants as well as the answers to the general questions about the refactorings implementation.

- **Change local method call dependency to a service call**: both respondents have applied the refactoring before and agree with its steps. As for the cases that were not mentioned in the mechanics, they pointed out the large volume of data with latency issues and the fact that sometimes, when updating data, multiple services may also need to be updated. Therefore, a synchronous request may lead to inconsistency between them. They both agreed that the figure clearly illustrated the refactoring and one of them mentioned that the figure should include the notion of temporal latency of synchronous calls.

- **Move Foreign-key relationship to code:** They have both previously implemented the refactoring, and one did not agree with the steps to apply it. They stated that the case of a user interface versus service decoupling did not represent the cases of splitting back-end services where integration layers can be required for functional decoupling. The other pointed out that sometimes the migration needs to carry more information than a single database table and that when that happens is not always clear that those dependencies need to be migrated as well. Neither agreed that the figure clearly represented the refactoring, noting that it should show the technical architecture vision for software deployment and interfaces and that the database schema should be included.

- **Replicate Data across microservices:** Both participants have used the refactoring before and agree with the proposed steps, having nothing to add. Regarding the figure, one of them did not agree that it clearly represented the refactoring because exception management and operations require additional mandatory components.

- **Split Database across microservices:** Only one of them has used it before, and he agrees with the steps with nothing to add to them. However, the other respondent disagrees entirely with this refactoring because, regarding the example, the customer management service should be notified of missing payments and check if the customer should be suspended and not the Finance worrying about giving them that data. Regarding the figure, the first respondent did not agree with it because it was missing visibility of the data fields that remain master/slave on each, while the other respondent agreed with it.

- **Break data type dependency:** has been used by both participants. They both agree with the steps and have nothing to add. However, regarding the figure, one believes it mainly focuses on software architecture and misses functional, data, integration and deployment.

Concerning the question if they agree with the premise that to extract a service at the first stage of migration, it is necessary to break its dependencies to the monolith and of the monolith to this service, the answers were maybe because they believe it depends if the migration is driven by functional decoupling or pure technical migration (in the latest the breaking of dependencies would be much harder as not addressing the division of functions); and yes.

As for the types of dependencies that occur and lead to refactoring that we did not mention, one of the participants mentioned the non-software "code" dependencies such as infrastructure, security components, and network that may require additional work not identified previously.

MicroOnion: A tool for assisting the refactoring to microservices

In this last section, we look into the perception of the participants on our tool and the suggestions for improvement.

Both participants were familiar with the refactorings suggested by the tool, agreed that the visual representations assisted in understanding the suggested refactorings and that a tool offers

sufficient information to carry out the suggested refactorings. Regarding whether the tool helped them understand the process of refactoring, one of the participants responded Neutral because if they were unfamiliar with the concepts, the tool would not be much help. In terms of trusting the tool's recommendation, one of the participants responded neutrally because they believed a deep analysis was needed to agree with the statement, and he was not familiar enough with the project to assess it this way.

They both agreed they trust the tool's recommendation more because it shows the overall sequence of refactorings from monolith to microservices. One was neutral regarding its ability to show how the system evolves with each step, and the other agreed. One disagrees with its ease of use because it says the UX should be simplified and provide a better-navigated tree view or a way to know where we are in the big picture. The other agreed and suggested showing more class properties in the schemas.

In the last question, we asked what other strategies they would use to sort the services extraction order. They answered most dependencies first, the ones with more database dependencies first, team availability, business maturity/change, and performing a brute force algorithm trying all possible orders and choosing the one that leads to the least number of refactorings. Additionally, they stated that businesses have other factors with strong implications on the order of refactoring and services extraction that make the tool useless if it only considers the software part or cannot be changed in context. Therefore, we should also be able to order by business necessity chosen by the user.

8.5 Analysis and Discussion

Even though we did not have enough participants to draw generalised conclusions, we believe this study still provides meaningful conclusions because we got feedback from very experienced professionals. The two participants have very different levels of expertise and background and did not show much disagreement, which, at first, may tell us that the results may be general.

Assistance to Practitioners

Overall, the indications for the refactorings improvement tell us that the mechanics have the necessary description for implementation. Some critical cases connected with large volumes of data, scale, performance and division of responsibilities may not be mentioned. These concerns are addressed in other chapters of the catalogue available in Appendix C.

Regarding the examples, participants have pointed out the need for the notion of the temporal latency of synchronous calls, the not-so-technical architecture vision and database schema. Which we understand are critical. However, displaying all of this information without overwhelming the developer is challenging.

There is room for improvement to be more inclusive, but the refactorings were serving its cause of assisting practitioners through the migration.

We highlight that Chapter 5 does not contain the feedback received in this survey, but the complete catalogue in Appendix C is the new version with the feedback taken into account. Also, the examples on this survey were regarding the examples of the tool in its visualisation part. Therefore, any insights on that will be applied in the future in our tool.

Trust in the output of MicroOnion

The users will trust the tool's recommendations more when they can assess it with familiar projects. One of them answered that they do not directly say they trust because they wanted to see it working on one of their projects to assess, while the other admitted to trusting the tool output.

They believe it carries sufficient information to guide the refactoring and say that it helps understand the migration process as they are familiar with the procedures mentioned.

Regarding strategies to sort the order by which we extract the services, it is consensual that allowing to order by business necessity is a must as it causes many migration limitations. Therefore, this strategy's implementation shall be included in the future work of the tool.

Participants trust the tool more because it shows the overall sequence of refactorings from monoliths to microservices and how it shows the evolution of the system at each step.

Trust in the output of MicroOnion supported by the visualisation it provides

The visual representations in the tool enhance the comprehension of the migration refactoring techniques and increase users' trust in the generated output. These visualisations provide a clear and intuitive way to understand the changes being made. However, it is worth noting that the tool's ease of use is not universally agreed upon. This suggests that further attention and improvements are necessary in terms of the tool's design and user interface. Enhancing the tool's usability can make it more useful and valuable for a wider range of users.

Comparing to the Empirical Study on Professionals' Perspective on Microservices migration

Although the results are not representative enough, we do not discard the conclusions obtained. The first study conducted in this dissertation had conclusions made using statistics. However, this study is different. Nevertheless, we have a participant in this second study that migrated more projects than any participant in the previous research, and both our respondents have four/five years of experience in microservices which is reasonable compared to the last survey. Both of them work in areas that were significantly present in the first study.

In the first study, we concluded that participants would appreciate more tool support on, among others, Microservice API design and understanding the monolith, and we believe part of it is achieved through our tool, as we show how to change method calls to service calls and provide visualisation of the monolith and of how it evolves through iterations. We use the agreed most important technique, the Strangler Fig, as the primary refactoring technique of our tool. We face critical challenges mentioned in the previous study, like database migration and data store splitting,

dealing with consistency (in our catalogue), communication among services and decoupling services from the monolith. Lastly, we highly recommend the improvement of some quality metrics during the development by suggesting sources to find ways of doing so.

Most of the insights obtained in this study we were already expecting because of the short time to complete the implementation. However, some important new insights also appeared, and the tool's future will benefit greatly.

8.6 Threats to Validity

Identifying the main threats to the results' validity and understanding our results' trustworthiness is crucial. In this section, we will point out the main internal and external validity threats.

Internal Validity

Internal validity shows how well the study was structured so that the results represent the truth of what was observed for that population [102] [78].

- **Insufficient and unfamiliar projects:** The fact that the tool only allows the analysis of pre-defined projects that the participants are unfamiliar with makes it harder for them to understand what is happening. In the future, with the generalisation of the tool, this issue would be mitigated as we could accept any project to provide the resulting sequence of refactorings to migrate it.
- **Tool's dependencies:** Additionally, the tool's output highly depends on the tool that provides the microservice decomposition, and we cannot guarantee that it is optimal. Many of the time, participants may focus a lot on this, forgetting the core importance of the refactorings.
- **Experimenter Bias:** Since the validation was conducted through a well-structured interview, some follow-up questions differed between participants, which may have led to additional thoughts on the respondents that guided their answers in another direction.
- **Interpretation Bias:** When interpreting the data, we could introduce our biases or preconceived notions. That is why we tried to be clear on what we were asking and explore what we asked further to the point that we understood enough to avoid making any assumptions about what was being answered.

External Validity

External validity relates to how generalisable our findings are [87].

- **Participant Selection Bias:** The selection for this study ended up being through contacts of personal networks. People with experience migrating to a microservices architecture were

contacted to participate in our research. We also shared it on our social media platforms, but no participant was acquired there. Because of that, that may exist bias from respondents based on their prior contact with the researchers. Additionally, these participants were volunteers, and because of that, we may think that people who are more willing to participate and contribute are represented. Those with different postures are not, which can make the results unable to be generalised. However, this is the best alternative because if we gave any compensation to people, we would need to be sure people were doing so in the right mindset, with critical thinking and will to help improve.

- **Sample size:** The number of participants could have been higher, limiting the confidence level in the conclusions and their generalizability, as the population sample was small. Since the target group was particular, the time to complete the study was relatively high, and it was performed during a season of many holidays. Hence, it took much work to find suitable participants available. Nevertheless, the insights extracted are still meaningful for improvement.

Chapter 9

Conclusion

The process of migrating a monolithic system into a microservices architecture would greatly benefit from systematisation. Besides contributing to the assistance of developers, it would facilitate the evolution of this process in terms of automation.

The way the migration is performed nowadays still deals with many challenges, little tool support and systematisation, and given the complexity of the implementation of this process, it is in desperate need of it to potentiate automation. Developers either do not have the knowledge of the existing tools or do not accept the output they produce, having a hard time trusting the low negative impact of the suggestions in their system.

With this dissertation, it is possible to automatically generate viable sequences of refactoring that assist the migration towards a microservices architecture. Looking at the related work of refactoring sequences and refactoring tools, we believe that a technology of this kind is possible, although we are aware that the challenges of achieving it stood on the conceptual side of the problem, how to identify the refactoring needs on the code, the order of performing them and how they affect the system.

9.1 Main Contributions

During this dissertation, we focused on creating a prototype of an assisted refactoring tool towards a microservice architecture. Therefore, we conducted a survey to understand how the migration is carried out by the practitioners, which provides interesting insights for further research, created a catalogue of refactorings that identifies some common refactorings to perform a migration and designed an approach and implemented a prototype tool to support the refactoring of monoliths to microservices by suggesting a refactoring sequence in an assisted and incremental way. To evaluate these contributions, the catalogue, the approach and the prototype developed, we performed an empirical study with software professionals.

Empirical Study on Professionals' Perspective on Microservices migration

We performed an empirical study through a quantitative survey to assess the perspective on microservices migration, how they do it, what processes they follow, and what tools they use. We concluded that the *Strangler Fig is the most commonly used technique* and they mostly perform the migration interspersed with the product evolution. Most of them do not use tools to assist their migration, and they need tools to decide service boundaries and refactor code, among others. Additionally, they need tools that are easy to use and can provide a visualisation of candidate decompositions.

Catalogue of Refactorings

The catalogue of refactorings was developed to contribute to the systematisation of refactoring monoliths to a microservices architecture. It intends to be a resource for developers to assist them during the migration.

MicroOnion: an Assisted and Incremental Refactoring Tool

We designed and prototyped MicroOnion, an assisted and incremental refactoring tool that can suggest a sequence of refactorings to perform the migration given a system representation and an intended decomposition in microservices. This tool also provides visualisation of how to apply the refactorings, their order and how it affects the system. Additionally, we provide the user with some extra informational content on how to conduct the migration in various categories besides extracting services: infrastructure improvement, deployment and orchestration and the microservices architecture characteristics.

Empirical study on the ability of MicroOnion to assist developers

We conducted an empirical evaluation through semi-structured interviews with experienced software professionals to assess both our approach and the developed prototype. Based on this study, we concluded that the refactoring catalog generally provides valuable assistance to developers during the migration process, although there is room for improvement in addressing specific cases more inclusively. The respondents demonstrated trust in the tool's recommendations to the extent that they expressed interest in further analysing it within a familiar project.

Adding the visual representation of the output enhances the understanding of migration techniques and increases developers' confidence in the tool. However, there is a need for improved user-friendliness.

9.2 Research Answers

Reaching the end of this dissertation research, we can now answer the research questions enunciated in Section 4.2.

9.2.1 RQ1. To which extent can known microservice migration techniques be described as sequences of smaller-scale refactorings?

We developed a catalogue of refactorings based on the literature review and the industry survey performed, where we identically describe the refactorings to Martin Fowler's description of code refactorings [33], with a title, motivation to use the refactoring, mechanics with the implementation explained step by step and an example of application. This catalogue contains high to low-level refactorings and mentions other topics besides refactorings.

In conclusion, it is possible to describe migration techniques, often disguised as migration patterns, as a sequence of smaller-scale refactorings, detailed to the point of explaining how to apply it directly in the code in different contexts.

9.2.2 RQ2. To what extent can a tool plan a migration to microservices architecture given a desired decomposition?

MicroOnion is a tool that receives a desired decomposition and a system representation and can suggest the sequence of refactorings to apply to go from the current state (monolith) to the desired decomposed state (microservices). With the visualisation we provide, the developer can plan when to perform each service extraction, deciding its own pace.

We performed a study that tells us this tool to plan the migration greatly benefits from visualisation. Although participants told us it is easier to assess it by being familiar with the projects available, the main conclusion is that it serves its purpose well. However, the UX should be enhanced to potentiate its value.

9.2.3 RQ3. Is it possible to step by step assess the impact of a sequence of refactoring on the system's evolution?

Our tool allows us to step by step assess the impact the sequence of refactoring has on the system's organisation and dependencies. However, it has yet to be possible to assess it in terms of quality attributes, although we highly advise that at each service extraction, to deploy the system and calculate metrics in run time to understand its impact on the system's quality attributes.

9.3 Future Work

Going forward, many changes could be made to this dissertation, and more experiments could be performed.

The catalogue of refactorings evolved from the catalogue of refactorings started by João Pinto [79] and added many more categories and small-scale refactorings. Although it has an already considerable size, it was only a first iteration of a complete catalogue. The refactorings presented may not be 100% complete, but they are still a good starting point for future work.

It can be improved on many levels, namely, refinement of the steps, inclusion of more edge cases and more examples. The goal shall always be to add things that bring value and are relevant

for developers. This also means that new techniques and refactorings will potentially appear and should be added to the catalogue, allowing it to evolve with the evolution of knowledge and new technologies. Therefore, we expected it to suffer multiple iterations to keep it updated and relevant.

The microservices community would greatly benefit from having more researchers and practitioners refining and contributing to this catalogue. Therefore, the catalogue should be made available and open-source so anyone can contribute following determined rules and with moderation. Additionally, more validation should be performed through case studies, surveys and experiments to contribute to the power of this catalogue, validate the remaining parts of the catalogue and better understand its benefits and gaps.

Regarding the assisted and incremental refactoring tool, besides the limitations of the prototype already mentioned in Section 7.8, we identified other things as future work.

The tool would greatly benefit from having user input recurrently during the various steps to customise the output to the user's needs and wants. We also note that we are not experts in design, so the tool has much space for improvement of UI & UX.

Since we focused on our specific project paradigm, Java projects using the Spring framework in an OOP paradigm, it is important to assess what changes are necessary to generalise this tool. This assessment is crucial because we do not want to limit developers because of the technology stack their projects use. Although we may find the case of a developer wishing to perform this refactoring but also wanting to change the technological stack, our tool does not expect to accommodate rewriting of the system as it raises many more challenges than simply performing the architectural refactoring.

The generalisation will never be 100%. By this, we mean that it is impossible not to have conditionals regarding the programming language or framework. However, a good start was to create input files as standard as possible while keeping all the information we needed. Most of the concepts used in the source code representation notation are used across all programming languages.

Nevertheless, we expect significant changes regarding dependencies identification, refactorings identification and refactorings application. These depend not only on the programming language/framework but also on the database paradigm if it is an ORM, relational or not, defined in a file or through code annotations, among others.

The dependencies identification will vary according to the programming languages, and this also means that not all dependencies can be encountered in all programming languages and that the standard way to solve them is not the same. So all of this will have to be implemented case by case for the different technologies. Using a relational or non-relational database also modifies how the database is partitioned and how the project is assessed. Lastly, the file organisation may differ according to the technologies or architectural patterns.

There may be additional cases specific to certain technologies we must remember when performing this. Therefore, we believe MicroOnion should know what technologies are being used to make confident decisions on what data to use but still keep the abstraction from the programming language.

Having identified the problem areas, we identified a solution to facilitate the accommodation of different technology stacks for the issues mentioned:

- Start by asking about the technologies used in the project when the developer uploads the input files.
- Apply the Abstract Factory design pattern [40] [1] to accommodate any technologies input the user may give
- Implement the dependencies identification, refactoring identification and refactoring application for each technology stack, always considering the technologies specificities.

Another solution may be to propose something similar to what was proposed by Teixeira et al. [99], where they propose an abstraction at the source code level to support multiple languages by sharing language specifications. This way, we could implement the refactorings for the abstraction created and let it apply to the actual programming language.

This tool should evolve to apply the refactorings on the code and fully automate the process, making part of the tool possibly a plugin for an IDE. However, in this case, the visualisation tool is still valid. Connecting these two tools in the way that one displays the specific decisions the user is making in the code and the other is updated with these decisions providing a high-level view of the state of the migration, assisting more in the planning of the migration, would be a plus. Furthermore, having a vision of the changes in the initial characteristics of the system (most likely regarding quality attributes) could contribute to the improvement of the trust of developers in this tool.

It also needs more validation and studies to obtain more insights for improvement, like case studies and experiments on real projects and surveys to assess its UI & UX, trust and ease of use.

We have recently seen some cases of returning to the monolithic architecture from microservices. Therefore, it would be interesting to allow the reverse path. Instead of going mono to micro, going micro to mono. Because what we want is constantly changing according to the circumstances. What is now the ideal architecture may not be in the future, and developers also need assistance in these cases.

[1] Abstract factory, Refactoring.Guru. Available at: https://refactoring.guru/design-patterns/abstract-factory (Accessed: 16 June 2023).

References

[1] Muhammad Abdullah, Waheed Iqbal, and Abdelkarim Erradi. Unsupervised learning approach for web application auto-decomposition into microservices. *Journal of Systems and Software*, 151:243–257, May 2019.

[2] Omar Al-Debagy and Péter Martinek. Extracting Microservices' Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach. In *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*, pages 289–294, June 2020.

[3] Carlos Albuquerque and Filipe Figueiredo Correia. Deployment tracking and exception tracking patterns: More monitoring design patterns for cloud-native applications. In *Proceedings of the 28th European Conference on Pattern Languages of Programs*, pages 1–9, 2023.

[4] Carlos Albuquerque, Kadu Relvas, Filipe Figueiredo Correia, and Kyle Brown. Proactive monitoring design patterns for cloud-native applications. In *Proceedings of the 27th European Conference on Pattern Languages of Programs*, pages 1–13, 2022.

[5] João Franscisco Almeida and António Rito Silva. Monolith Migration Complexity Tuning Through the Application of Microservices Patterns. In Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, and Olaf Zimmermann, editors, *Software Architecture*, Lecture Notes in Computer Science, pages 39–54, Cham, 2020. Springer International Publishing.

[6] Alexandra Altvater. What are Software Metrics? Examples & Best Practices. https://stackify.com/track-software-metrics/, September 2017. Accessed February 05, 2023.

[7] Bernardo Andrade, Samuel Santos, and António Rito Silva. From Monolith to Microservices: Static and Dynamic Analysis Comparison. 2022. Publisher: arXiv Version Number: 1.

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

[9] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A. Tamburri, and Theo Lynn. Microservices migration patterns. *Software: Practice and Experience*, 48(11):2019–2042, 2018. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2608.

[10] João Barbosa. Towards a Smart Recommender for Code Refactoring. Master's thesis, Faculdade de Engenharia da Universidade do Porto, Porto, PT, July 2020.

[11] Jacqui Bartram. Library: Literature reviews: Scoping and planning. https://libguides.hull.ac.uk/literaturereviews/scoping.

[12] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Industry practices and challenges for the evolvability assurance of microservices: An interview study and systematic grey literature review. *Empirical Software Engineering*, 26(5):104, September 2021.

[13] Miguel Brito, Jácome Cunha, and João Saraiva. Identification of microservices from monolithic applications through topic modelling. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1409–1418, Virtual Event Republic of Korea, March 2021. ACM.

[14] Kyle Brown, Bobby Woolf, Cees De Groot, Chris Hay, and Joseph Yoder. Patterns for developers and architects building for the cloud, 2021.

[15] Antonio Bucchiarone, Kemal Soysal, and Claudio Guidi. A Model-Driven Approach Towards Automatic Migration to Microservices. In Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, Lecture Notes in Computer Science, pages 15–36, Cham, 2020. Springer International Publishing.

[16] Rui Chen, Shanshan Li, and Zheng Li. From Monolith to Microservices: A Dataflow-Driven Approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475, December 2017.

[17] Chris Richardson of Eventuate, Inc. Service discovery in a microservices architecture. https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/, 2015. Accessed June 26, 2023.

[18] Anuradha Chug and Sandhya Tarwani. Determination of optimum refactoring sequence using A* algorithm after prioritization of classes. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1624–1630, September 2017.

[19] Christian Ciceri, Dave Farley, Neal Ford, Andrew Harmel-Law, Michael Keeling, Carola Lilienthal, and João Rosa. *Software Architecture Metrics*. O'Reilly Media, Sebastopol, CA, 2022.

[20] Michel-Daniel Cojocaru, Alexandru Uta, and Ana-Maria Oprescu. Attributes Assessing the Quality of Microservices Automatically Decomposed from Monolithic Applications. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 84–93, June 2019. ISSN: 2379-5352.

[21] Melvin E Conway. HOW DO COMMITTEES INVENT? *Datamation*, 1968.

[22] José Correia and António Rito Silva. Identification of monolith functionality refactorings for microservices migration. *Software: Practice and Experience*, 52(12):2664–2683, 2022. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3141.

[23] Derek Comartin. Anti-corruption layer for mapping between boundaries. https://codeopinion.com/anti-corruption-layer-for-mapping-between-boundaries/, 2022. Accessed June 26, 2023.

[24] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. Migrating Towards Microservice Architectures: An Industrial Survey. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 29–2909, April 2018.

[25] João Tiago Duarte Maia and Filipe Figueiredo Correia. Service mesh patterns. In *Proceedings of the 27th European Conference on Pattern Languages of Programs*, pages 1–12, 2022.

[26] John Edvald. Seven hard-earned lessons learned migrating a monolith to microservices. https://www.infoq.com/articles/lessons-learned-monolith-microservices/, 2020. Accessed June 26. 2023.

[27] Thomas Engel, Melanie Langermeier, Bernhard Bauer, and Alexander Hofmann. Evaluation of Microservice Architectures: A Metric and Tool-Based Approach. In Jan Mendling and Haralambos Mouratidis, editors, *Information Systems in the Big Data Era*, Lecture Notes in Business Information Processing, pages 74–89, Cham, 2018. Springer International Publishing.

[28] Erik Rahtjen. An introduction to kubernetes. https://stablekernel.com/article/an-introduction-to-kubernetes/. Accessed June 26, 2023.

[29] Christian Esposito, Aniello Castiglione, and Kim-Kwang Raymond Choo. Challenges in Delivering Software in the Cloud as Microservices. *IEEE Cloud Computing*, 3(5):10–14, September 2016. Conference Name: IEEE Cloud Computing.

[30] Neal Ford, Rebecca Parsons, and Patrick Kua. *Building Evolutionary Architectures*. O'Reilly Media, Inc., Sebastopol, CA, 2017.

[31] Martin Fowler. bliki: CodeSmell. https://martinfowler.com/bliki/CodeSmell.html, 2006. Accessed June 19, 2023.

[32] Martin Fowler. Tolerantreader. https://martinfowler.com/bliki/TolerantReader.html, 2011. Accessed June 12, 2023.

[33] Martin Fowler. *Refactoring: Improving the Design of Existing Code: 2nd Edition*. Addison-Wesley, Reading, MA, 2019.

[34] Martin Fowler. Monolithfirst. Available at https://martinfowler.com/bliki/MonolithFirst.html, 2022. Accessed November 22, 2022.

[35] Marting Fowler. Event Sourcing. https://martinfowler.com/eaaDev/EventSourcing.html, 2005. Accessed June 12, 2023.

[36] Francisco Freitas, André Ferreira, and Jácome Cunha. Refactoring java monoliths into executable microservice-based applications. Master's thesis, Universidade do Minho - Escola de Engenharia, Braga, PT, December 2021.

[37] Francisco Freitas, André Ferreira, and Jácome Cunha. Refactoring Java Monoliths into Executable Microservice-Based Applications. In *25th Brazilian Symposium on Programming Languages*, pages 100–107, Joinville Brazil, September 2021. ACM.

[38] Jonas Fritzsch, Justus Bogner, Stefan Wagner, and Alfred Zimmermann. Microservices Migration in Industry: Intentions, Strategies, and Challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 481–490, September 2019. ISSN: 2576-3148.

[39] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. From Monolith to Microservices: A Classification of Refactoring Approaches. In Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, Lecture Notes in Computer Science, pages 128–141, Cham, 2019. Springer International Publishing.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[41] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1303–1313, Athens Greece, August 2021. ACM.

[42] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service Cutter: A Systematic Approach to Service Decomposition. In Marco Aiello, Einar Broch Johnsen, Schahram Dustdar, and Ilche Georgievski, editors, *Service-Oriented and Cloud Computing*, Lecture Notes in Computer Science, pages 185–200, Cham, 2016. Springer International Publishing.

[43] Mark Harman and Laurence Tratt. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1106–1113, London England, July 2007. ACM.

[44] Thiago Henrique. Saga pattern para microservices. https://dev.to/thiagosilva95/saga-pattern-para-microservices-2pb6, 2021. Accessed June 26, 2023.

[45] IBM. What is automation? | IBM. https://www.ibm.com/topics/automation. Accessed January 31, 2023.

[46] Joseph Ingeno. *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing Ltd, August 2018. Google-Books-ID: 6EZsDwAAQBAJ.

[47] James Ivers, Robert L. Nord, Ipek Ozkaya, Chris Seifried, Christopher S. Timperley, and Marouane Kessentini. Industry experiences with large-scale refactoring. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1544–1554, Singapore Singapore, November 2022. ACM.

[48] James Ivers, Robert L. Nord, Ipek Ozkaya, Chris Seifried, Christopher S. Timperley, and Marouane Kessentini. Industry's Cry for Tools that Support Large-Scale Refactoring. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Pittsburgh, PA, USA, 2022. IEEE.

[49] I. Ivkovic and K. Kontogiannis. A framework for software architecture refactoring using model transformations and semantic annotations. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 10 pp.–144, March 2006. ISSN: 1534-5351.

[50] Susan Jamieson. Likert scales: How to (ab)use them. *Medical education*, 38(12):1217–1218, 2004.

[51] Rubén Jesus. From Monolith to Microservices: automating service boundary detection. Master's thesis, Faculdade de Engenharia da Universidade do Porto, Porto, PT, July 2021.

[52] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng. Service Candidate Identification from Monolithic Systems Based on Execution Traces. *IEEE Transactions on Software Engineering*, 47(5):987–1007, May 2021. Conference Name: IEEE Transactions on Software Engineering.

[53] Jonathan Johnson and Laura Shiff. What is microservice architecture? microservices explained. https://www.bmc.com/blogs/microservices-architecture/, 2021. Accessed June 26, 2023.

[54] Anup K. Kalia, Jin Xiao, Chen Lin, Saurabh Sinha, John Rofrano, Maja Vukovic, and Debasish Banerjee. Mono2Micro: an AI-based toolchain for evolving monolithic enterprise applications to a microservice architecture. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1606–1610, Virtual Event USA, November 2020. ACM.

[55] 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*, Lecture Notes in Computer Science, pages 32–47, Cham, 2018. Springer International Publishing.

[56] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley Professional, hardcover edition, 8 2004.

[57] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering – A systematic literature review. *Information and Software Technology*, 51(1):7–15, January 2009.

[58] Gopala Krishna Behara. Microservices Governance - Wipro. https://www.wipro.com/blogs/dr-gopala-krishna-behara/microservices-governance/, 2019. Accessed June 12, 2023.

[59] Sulabh Kumar. Adapter Pattern. https://www.geeksforgeeks.org/adapter-pattern/, May 2016. Accessed June 12, 2023.

[60] Dilshodbek Kuryazov, Dilshod Jabborov, and Bekmurod Khujamuratov. Towards Decomposing Monolithic Applications into Microservices. In *2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT)*, pages 1–4, October 2020. ISSN: 2472-8586.

[61] Young-Woo Kwon and Eli Tilevich. Cloud refactoring: automated transitioning to cloud-based services. *Automated Software Engineering*, 21(3):345–372, September 2014.

[62] James Lewis and Martin Fowler. Microservices. https://martinfowler.com/articles/microservices.html, 2014. Accessed January 26, 2023.

[63] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 22(140):5–55, 1932.

[64] Bo Liu, Jingliu Xiong, Qiurong Ren, Shmuel Tyszberowicz, and Zheng Yang. Log2MS: a framework for automated refactoring monolith into microservices using execution logs. In *2022 IEEE International Conference on Web Services (ICWS)*, pages 391–396, July 2022.

[65] Gastón Márquez, Mónica M Villegas, and Hernán Astudillo. A pattern language for scalable microservices-based systems. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, pages 1–7, 2018.

[66] Martin Fowler. Circuit breaker. https://martinfowler.com/bliki/CircuitBreaker.html, 2014. Accessed June 26, 2023.

[67] Tiago Matias, Filipe F. Correia, Jonas Fritzsch, Justus Bogner, Hugo S. Ferreira, and André Restivo. Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis. In Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, and Olaf Zimmermann, editors, *Software Architecture*, Lecture Notes in Computer Science, pages 315–332, Cham, 2020. Springer International Publishing.

[68] Tiago Cardoso Matias. Streamlined Refactoring of Modern Web Frameworks to Microservices. Master's thesis, Faculdade de Engenharia da Universidade do Porto, Porto, PT, July 2019.

[69] Genc Mazlami, Jürgen Cito, and Philipp Leitner. Extraction of Microservices from Monolithic Software Architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531, June 2017.

[70] Panita Meananeatra. Identifying refactoring sequences for improving software maintainability. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 406–409, September 2012.

[71] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004. Conference Name: IEEE Transactions on Software Engineering.

[72] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering*, 21(6):2503–2545, December 2016.

[73] MuleSoft. Microservices vs monolithic architecture. https://www.mulesoft.com/resources/api/microservices-vs-monolithict. Accessed January 26, 2023.

[74] Sam Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, Beijing Sebastopol, CA, first edition edition, 2015. OCLC: ocn881657228.

[75] Sam Newman. *Monolith to Microservices: Evolutionary Patterns to Transform your Monolith*. O'Reilly Media, Inc., Sebastopol, CA, 2019.

[76] Luís Nunes, Nuno Santos, and António Rito Silva. From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts. In Tomas Bures, Laurence Duchien, and Paola Inverardi, editors, *Software Architecture*, Lecture Notes in Computer Science, pages 37–52, Cham, 2019. Springer International Publishing.

[77] Benoît Otjacques, Patrik Hitzelberger, Stefan Naumann, and Volker Wohlgemuth, editors. *From Science to Society: New Trends in Environmental Informatics*. Progress in IS. Springer International Publishing, Cham, 2018.

[78] Cecilia Maria Patino and Juliana Carvalho Ferreira. Internal and external validity: can you apply research study results to your patients? *Jornal Brasileiro de Pneumologia*, 44(3):183, 2018.

[79] João Paiva da Costa Pinto. Refactoring Monoliths to Microservices. Master's thesis, Faculdade de Engenharia da Universidade do Porto, Porto, PT, July 2019.

[80] Paul Ralph. ACM SIGSOFT Empirical Standards Released. *ACM SIGSOFT Software Engineering Notes*, 46(1):19–19, January 2021.

[81] David Reis, Bruno Piedade, Filipe F. Correia, João Pedro Dias, and Ademar Aguiar. Developing Docker and Docker-Compose Specifications: A Developers' Survey. *IEEE Access*, 10:2318–2329, 2022. Conference Name: IEEE Access.

[82] Chris Richardson. Circuit Breaker. https://microservices.io/patterns/reliability/circuit-breaker.html. Accessed June 12, 2023.

[83] Chris Richardson. Pattern: Monolithic architecture. https://microservices.io/patterns/monolithic.html. Accessed January 26, 2023.

[84] Chris Richardson. *Microservices patterns: with examples in Java*. Manning Publications, Shelter Island, New York, 2019. OCLC: on1002834182.

[85] Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma. Refactoring for software architecture smells. *Proceedings of the 1st International Workshop on Software Refactoring*, 2016.

[86] Sebastian Peyrott. An introduction to microservices, part 3: The service registry. https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/, 2015. Accessed June 26, 2023.

[87] Julia Simkus. Internal Vs. External Validity. https://www.simplypsychology.org/internal-vs-external-validity.html, March 2023. Accessed June 15, 2023.

[88] Leonardo Sousa, Willian Oizumi, Alessandro Garcia, Anderson Oliveira, Diego Cedrim, and Carlos Lucena. When Are Smells Indicators of Architectural Refactoring Opportunities: A Study of 50 Software Projects. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 354–365, Seoul Republic of Korea, July 2020. ACM.

[89] Tiago Boldt Sousa, Ademar Aguiar, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. Engineering software for the cloud: patterns and sequences. In *Proceedings of the 11th Latin-American Conference on Pattern Languages of Programming*, pages 1–8, 2016.

[90] Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. Patterns for software orchestration on the cloud. In *Proceedings of the 22nd Conference on Pattern Languages of Programs*, pages 1–12, 2015.

[91] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: Messaging systems and logging. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, pages 1–14, 2017.

[92] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: Automated recovery and scheduler. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–8, 2018.

[93] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: External monitoring and failure injection. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–8, 2018.

[94] Simone Staffa, Giovanni Quattrocchi, Alessandro Margara, and Gianpaolo Cugola. Pangaea: Semi-automated Monolith Decomposition into Microservices. In Hakim Hacid, Odej Kao, Massimo Mecella, Naouel Moha, and Hye-young Paik, editors, *Service-Oriented Computing*, Lecture Notes in Computer Science, pages 830–838, Cham, 2021. Springer International Publishing.

[95] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing*, 4(5):22–32, September 2017. Conference Name: IEEE Cloud Computing.

[96] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural Patterns for Microservices: A Systematic Mapping Study:. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science*, pages 221–232, Funchal, Madeira, Portugal, 2018. SCITEPRESS - Science and Technology Publications.

[97] Davide Taibi and Kari Systä. A Decomposition and Metric-Based Evaluation Framework for Microservices. In Donald Ferguson, Víctor Méndez Muñoz, Claus Pahl, and Markus Helfert, editors, *Cloud Computing and Services Science*, Communications in Computer and Information Science, pages 133–149, Cham, 2020. Springer International Publishing.

[98] Sandhya Tarwani and Anuradha Chug. Sequencing of refactoring techniques by Greedy algorithm for maximizing maintainability. In *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1397–1403, September 2016.

[99] Gil Teixeira, João Bispo, and Filipe F. Correia. Multi-language static code analysis on the LARA framework. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2021, pages 31–36, New York, NY, USA, June 2021. Association for Computing Machinery.

[100] John Thompson. Adapter Pattern. https://springframework.guru/gang-of-four-design-patterns/adapter-pattern/, September 2022. Accessed June 12, 2023.

[101] Tomas Fernandez. Release management for microservices. https://semaphoreci.com/blog/release-management-microservices/, 2022. Accessed June 26, 2023.

[102] William M. K. Trochim. Internal Validity. https://conjointly.com/kb/internal-validity/. Accessed June 15, 2023.

[103] Shmuel Tyszberowicz, Robert Heinrich, Bo Liu, and Zhiming Liu. Identifying Microservices Using Functional Decomposition. In Xinyu Feng, Markus Müller-Olm, and Zijiang Yang, editors, *Dependable Software Engineering. Theories, Tools, and Applications*, Lecture Notes in Computer Science, pages 50–65, Cham, 2018. Springer International Publishing.

[104] Siraj ul Haq. Introduction to monolithic architecture and microservices architecture. https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211 2018. Accessed January 26, 2023.

[105] Guilherme Vale, Filipe Figueiredo Correia, Eduardo Martins Guerra, Thatiane de Oliveira Rosa, Jonas Fritzsch, and Justus Bogner. Designing microservice systems using patterns: An empirical study on quality trade-offs. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 57–57, 2022.

[106] Victor Velepucha and Pamela Flores. Monoliths to microservices - Migration Problems and Challenges: A SMS. In *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*, pages 135–142, March 2021.

[107] VMware. What is IT Automation? | VMware Glossary. https://www.vmware.com/topics/glossary/content/it-automation.html. Accessed January 31, 2023.

[108] Joseph W. Yoder and Paulo Merson. Strangler patterns. In *Proceedings of the 27th Conference on Pattern Languages of Programs*, PLoP '20, pages 1–25, USA, January 2022. The Hillside Group.

[109] Yukun Zhang, Bo Liu, Liyun Dai, Kang Chen, and Xuelian Cao. Automated Microservice Identification in Legacy Systems with Functional and Non-Functional Metrics. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 135–145, March 2020.

[110] Olaf Zimmermann. Architectural refactoring for the cloud: a decision-centric view on cloud migration. *Computing*, 99(2):129–145, February 2017.

[111] Rob Zuber. Letting change and uncertainty advance your software architecture. https://circleci.com/blog/letting-change-and-uncertainty-advance-your-software-architecture/, 2020. Accessed June 26, 2023.

[112] Bc Jiří Široký. From Monolith to Microservices: Refactoring Patterns. Master's thesis, Massachusetts Institute of Technology, Brno, CZ, 2021.

Appendix A

Migration Patterns

This appendix the descriptions of the migration patterns used in the context of the Industry Survey. These are all based on Sam Newman's book *"Monolith to Microservices: Evolutionary Patterns to Transform your Monolith"* [75].

A.1 Strangler Fig Application

When to use:When we have decided to evolve a system to a microservices architecture, and we want to take incremental steps toward the new architecture but also ensure that each step is easily reversible, reducing risks.

How to apply it: Make minimal changes to the existing system and gradually move functionality over to the new microservices architecture. Do this by replacing or rewriting existing features in parallel to the old architecture, one at a time, until the old architecture has been fully replaced. Usually, we will need to create a proxy or façade that provides a stable API for old clients throughout the migration.

A.2 UI Composition

When to use: As we incrementally migrate a monolith to microservices, the functionality the user-interface provides needs to be served partly by an existing monolith and partly by the new microservice architecture.

How to apply it: Different variants of UI Composition may be used: Page-based Composition consists of having different services serve different pages; Widget Composition consists of embedding a piece of user-interface provided by a new service into the monolith-provided user interface; and Micro Frontends consist of taking a microservice-based approach to frontend development.

A.3 Branch by Abstraction

When to use: When changes to the existing codebase will likely take time to carry out, and we want to avoid any disruption. It assumes we can change the code of the current system.

How to apply it: Develop, in the monolith, an alternative implementation for a module so that the new and old implementations comply with the same interface/abstraction. The new implementation will typically call a new external service, whereas the old would implement the functionality internally. At some point, switch from the old implementation to the new implementation.

A.4 Parallel Run

When to use: When the failure of the functionality being worked implies a high risk for the business.

How to apply it: With a parallel run, rather than calling either the old or the new implementation, we call both, allowing us to compare the results to ensure they are equivalent. Despite calling both implementations, only one is considered the source of truth at any given time. Typically, the old implementation is considered the source of truth until the ongoing verification reveals that we can trust the new implementation.

A.5 Decorating Collaborator

When to use: When we want to trigger some behavior based on something happening inside the monolith, but we want to avoid changing the monolith itself. This technique assumes we can intercept responses returned by the monolith before they reach the clients.

How to apply it: Use the decorator pattern to make it appear from the outside that we have added new logic to the monolith without actually changing it. The technique consists of routing client requests to a proxy which will allow the call to reach the monolith, as it normally would, and based on the result of this call, call out to a new microservice, and possibly modify the result in some way before returning it to the client.

A.6 Change Data Capture

When to use: When we need to react to a change in data in the monolith but cannot intercept this change at the perimeter of the monolith (e.g., using a decorator) or change its implementation.

How to apply it: Rather than trying to intercept and act on calls made into the monolith, we react to changes made in a datastore. The underlying capture system will be coupled to the monolith's datastore, and can rely on database triggers, transaction log pollers (usually a file, into which is written a record of all the changes that have been made) or batch delta copier (a program that regularly scans the database in question for what data has changed).

A.7 Change code dependency to service call

When to use: When a software system is decomposed into a set of smaller services to use a microservices architectural style, some components in the system will act as dependencies to other services or components.

How to apply it: Try to keep the services' code as separate as possible. When services depend on the same piece of code, there is a chance that changes to that code motivated by the needs of one service might negatively affect the other service. If this is code shared as a library that rarely changes (e.g., a string manipulation library) it is reasonable to have different services depend on it. On the other hand, if this is code related to internal entities, or entails different scalability needs, we share it as a new service so that it can be changed independently and gradually, or scaled independently.

A.8 Database View

When to use: We want a single source of data for multiple services, but it is impractical to change all clients to point to the new service(s). For clients that will keep considering the monolith as their source of data, we want to mitigate concerns regarding coupling to specific parts of the data schema. There are more clients reading data than writing it.

How to apply it: For clients that only read data, create a dedicated schema that hosts views looking like the old schema, and that are assembled from data now owned by the new service(s). Have clients that only read data point at those views instead of the original tables. We will need to change only the clients that need to write data so that they start using the new services directly, instead of the monolith.

A.9 Database Wrapping Service

When to use: When multiple clients depend on the same datastore, but it is just too hard to consider pulling apart the underlying schema.

How to apply it: Hide the database behind a service that acts as a thin wrapper and make clients depend on this new service instead of the database. Encourage developers writing the different client applications to think of this new service as someone else's and start storing their own data locally.

A.10 Database-as-a-Service Interface

When to use: When we have clients that really just need a database to query. This could be because they need to query large amounts of data, or because external parties are already using toolchains that require a SQL endpoint to work against.

How to apply it: Create a dedicated database to be exposed as a read-only endpoint, and have this database populated when the data in the underlying database changes. Take care to separate the database we expose from the database we use inside our service boundary. A mapping engine takes changes in the internal database, and works out what changes need to be made in the external database. When our internal database changes structure, the mapping engine will need to change to ensure that the public-facing database remains consistent.

A.11 Aggregate Exposing the Monolith

When to use: As we extract services from the monolith, we often realize that some of the data should become part of the new services, and some of it should stay where it is. When the monolith still owns the data we want to access, we will need to allow the new services the means to access this data.

How to apply it: Make explicit what information the new services need from the monolith by exposing it via dedicated service endpoints. The monolith still owns the data and decides what state changes are valid; it is not just like a wrapper around a database. Beyond just exposing data, expose operations that allow external parties to query the current state and to make requests for new state changes.

A.12 Change Data Ownership

When to use: As we extract services from the monolith, we often realize that some of the data should become part of the new services, and some of it should stay where it is. When the newly extracted service encapsulates the business logic that changes some data, that data should be under the new service's control.

How to apply it: Move the data from the monolith over into the new service. Change the monolith to treat the new service as the source of truth for the data it now owns, calling it to read or change data when needed. When doing this, we may have to consider the consequences of breaking foreign-key constraints and transactional boundaries.

A.13 Synchronize Data in Application

When to use: When we want to split the schema of the monolith into two separate datastores, in preparation for extracting a new service.

How to apply it: Do this in a sequence of three steps. The first step is to create the new datastore and bulk-import data from the monolith—only the part of the schema we are interested in splitting needs to be imported. The second step consists of making the monolith write the same data to both datastores, ensuring they are kept in-sync, and that writes to the new datastore are working well. In the third step, with a simple change to the monolith, we make the new datastore

the source of truth and ensure that the reads also work. As we still write to both datastores, we can easily fallback to reading from the old datastore if any issue is found.

A.14 Tracer Write

When to use: When we need to move the ownership of some data from the monolith to a new service in an incremental fashion, tolerating there being two sources of truth during the migration.

How to apply it: First, identify the part of the monolith data schema whose ownership we want to change and the service that will host it. Then, ensure this data is kept in-sync between the monolith and the new service. This can be achieved in a number of ways: a) Allowing writes only on one of the sources of truth, and making it in-charge of replicating the change in the other one; b) Have clients send writes to both sources; or c) Allow clients to send writes to either source, and synchronize data behind the scenes. Once all clients use the new source of truth, the old source of truth can be retired. Note: Synchronize Data in Application focuses on splitting the schema, and this technique focuses on moving a part of the schema to the ownership of a previously created service.

A.15 Split Table

When to use: When we are using a relational database and we find that columns of a single table need to be split across two or more service boundaries.

How to apply it: First, split the columns of the table apart in the existing schema, placing one (or more) of them in a new table. Then, move the new table to a different service. If there was code updating columns that are now owned by different services, the monolith will need to call the new service to update the split column.

A.16 Move Foreign-Key Relationship to Code

When to use: When we are using a relational database, and we are moving some functionality to a new service but we want to keep some of the data used by that functionality in the monolith.

How to apply it: With a monolithic system, joining different relational tables is often done by database queries, but when one of those tables is moved to a new service, this new service may need to fetch from the monolith any data that it requires that is still kept there, and join it "in memory" with the data it owns, rather than "in the database".

Appendix B

Migration of Monoliths to Microservices Survey

B.1 Survey

The following survey is a copy of the survey handed to developers that participated in the study of how practitioners perform the migration from monoliths to microservices (cf. 3).

Migration of Monoliths to Microservices

This survey is part of ongoing research on how refactoring towards a microservice architecture is currently done by professionals. We are studying what processes are followed, what tools are used and how decomposition results are evaluated.

Please answer it if you have been involved in at least one migration of a monolith to a microservices architecture. The expected time to answer is under 30 minutes.

* Indica uma pergunta obrigatória

Confidentiality Statement

By collaborating on this research, you understand that the data collected through this form:
1. Includes no personally identifying information.
2. Will be used by the researchers exclusively for the purpose of scientific inquiry.
3. May be published by the researchers (e.g., in journals, conferences, or blog posts).
4. Will go through additional anonymization and aggregation steps before being published.
5. Will be kept by the researchers in perpetuity and can used for future academic studies.
6. Is collected using Google Forms and, therefore, its collection and use is subject to Google's Privacy Policy (policies.google.com/privacy).

Authors

The research is being conducted by Rita Peixoto, Filipe Correia and Tiago Boldt Sousa (University of Porto), Jonas Fritzsch and Justus Bogner (University of Stuttgart), Nour Ali (Brunel University London), Eduardo Guerra (Free University of Bozen-Bolzano) and Cesare Pautasso (University of Lugano). If you have any questions, please get in touch with Rita Peixoto and Filipe Correia (up201806257@g.uporto.pt, filipe.correia@fe.up.pt).

Sources

Some of the materials used in this survey are based on (and sometimes paraphrase) the book "*Monolith to Microsevices: Evolutionary Patterns to Transform Your Monolith*", by Sam Newman. O'Reilly Media, 2019.

1. Experience and Background

1. 1.1 What areas have you been working on these past 5 years? *
   Tick all that apply.

   Marcar tudo o que for aplicável.

   - [ ] Software Development
   - [ ] Software Architecture
   - [ ] Operations
   - [ ] Quality assurance
   - [ ] Product Management
   - [ ] Coaching
   - [ ] Teaching
   - [ ] Scientific Research
   - [ ] Outra: _____

2. 1.2. What is your title? *
   E.g., Senior Developer, Architect, Principal Software Engineer, Tester, etc.

   _____

3. 1.3. Which country are you working from? *

   _____

4. 1.4. What is your professional experience in Microservices (in years)? *

   _____

5. 1.5. How many projects that involved the migration of monoliths to microservices have you been involved in? *

   _____

6. 1.6. What were the domain areas of these projects? *

Marcar tudo o que for aplicável.

- [ ] Healthcare
- [ ] Finance
- [ ] Retail
- [ ] Social Media
- [ ] Security
- [ ] Manufacturing
- [ ] Education
- [ ] Travel and Tourism
- [ ] Insurance
- [ ] Construction
- [ ] Real Estate
- [ ] Supply Chain Management
- [ ] Automotive
- [ ] Outra: _____

7. 1.7. Roughly how many monthly active users did these systems serve when the process of migrating to microservices was started? *

Marcar apenas uma oval.

- ◯ < 100
- ◯ 100 – 1K
- ◯ 1K – 10K
- ◯ 10K – 100K
- ◯ 100K – 1M
- ◯ > 1M

8. 1.8. Roughly how many people were working on these systems when the process of migrating to microservices was started? *

Marcar apenas uma oval.

- ◯ < 10
- ◯ 10 – 50
- ◯ 50 – 200
- ◯ 200 – 600
- ◯ > 600

2. Strategies and Processes

9. 2.1. Where do you look for guidance when migrating a monolith system to microservices? *
Tick all that apply

Marcar tudo o que for aplicável.

- ☐ Scientific articles
- ☐ Books
- ☐ Conference presentations
- ☐ Web resources, blogs
- ☐ Other practitioners' experiences
- ☐ External consulting
- ☐ Internal consulting
- ☐ Outra: _____

10. 2.2. How do you often plan the migration of a monolith to microservices in regards to the evolution of the product? *
Tick all that apply

Marcar tudo o que for aplicável.

- [ ] Rewrite/rebuild the entire system from scratch
- [ ] Stop product evolution, refactor the system to microservices, and then proceed with evolving the product
- [ ] Continuous refactoring interspersed with product evolution
- [ ] Outra: _____

11. 2.3. How likely are you to consider these data sources when deciding how to decompose a monolith into different services? *

Marcar apenas uma oval por linha.

	Very unlikely	Unlikely	Neutral	Likely	Very likely
The output of static analysis tools (e.g., based on source code repository analysis)					
The output of dynamic analysis tools (e.g., based on runtime data, such as logs or traces)					
Development process data (e.g., from version-control, project management tools)					
Software Documentation (e.g., business logic/capabilities/objects, data flow) and related artifacts (e.g., source code, configuration files)					

12. 2.4. Which criteria do you use often for deciding the new service boundaries when decomposing a monolith into different services? *
Tick all that apply

Marcar tudo o que for aplicável.

- [ ] Decomposition by business capability (i.e., so that the decomposition reflects the structure of the organization, its teams and business units)
- [ ] Decomposition by subdomain (i.e., so that the decomposition reflects an analysis of the processes and information flows of the business. e.g., using Domain-Driven Design)
- [ ] Based on coupling and cohesion of the business logic of the monolith (i.e., keep strongly-coupled elements of the implementation in the same services)
- [ ] Based on coupling and cohesion of the data owned by the monolith (i.e., keep strongly-coupled data entities managed by the same services)
- [ ] Based on lexical similarity (e.g., joining in the same service API endpoints with similar names)
- [ ] Outra: _____

3. Tools

13. 3.1. Thinking of the migration from monolith to microservices projects you have been involved in, have you been assisted by any automated or semi-automated tools? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

14. 3.2. If yes, which tools were used and for which purpose?

_____

15. 3.3. Which type of tools did you find the most useful?

_____

16. 3.4. In which of these activities would you most appreciate additional tool support? *
Tick at most 5 options.

Marcar tudo o que for aplicável.

- [ ] Understanding the monolith
- [ ] Deciding services boundaries
- [ ] Refactoring code
- [ ] Evaluating a decomposition result
- [ ] Regression testing
- [ ] Deployment automation
- [ ] Analysis of organizational restructuring needs (team size, team organization, etc.)
- [ ] Planning and managing the refactoring process
- [ ] Microservice API design
- [ ] Outra: _____

17. 3.5. What characteristics would you find most important in a tool for refactoring towards a microservice architecture? *
Tick at most 8 options.

Marcar tudo o que for aplicável.

- [ ] Ease of use
- [ ] Versatility – usable via command line, within IDEs, independently, etc.
- [ ] Support for various programming languages
- [ ] Requiring minimal manual inputs
- [ ] Doing the refactoring quickly
- [ ] Being actively maintained
- [ ] Having successful case studies reported
- [ ] Ability to apply restrictions to the decomposition result
- [ ] Easy modifications on restrictions without having to start over
- [ ] Multiple decompositions alternatives
- [ ] Provide an explanation of candidate decompositions
- [ ] Provide a visualization of candidate decompositions
- [ ] Edit/model a candidate decomposition
- [ ] Provide comprehensive decomposition guidance
- [ ] Provide an analysis of the changes of quality attributes and metrics
- [ ] Outra: _____

4. Refactoring Techniques - Splitting the Monolith

In this section, we explore a selection of refactoring techniques for splitting a monolith. The goal is to understand how often these techniques are used, how they are used, their challenges and which techniques are used together.

Please take the next ~5min to read about each one of the refactoring techniques below.

Strangler Fig Application

When to use: When we have decided to evolve a system to a microservices architecture, and we want to take incremental steps toward the new architecture but also ensure that each step is easily reversible, reducing risks.

How to apply it: Make minimal changes to the existing system and gradually move functionality over to the new microservices architecture. Do this by replacing or rewriting existing features in parallel to the old architecture, one at a time, until the old architecture has been fully replaced. Usually, we will need to create a proxy or façade that provides a stable API for old clients throughout the migration.

UI Composition

When to use: As we incrementally migrate a monolith to microservices, the functionality the user-interface provides needs to be served partly by an existing monolith and partly by the new microservice architecture.

How to apply it: Different variants of *UI Composition* may be used: *Page-based Composition* consists of having different services serve different pages; *Widget Composition* consists of embedding a piece of user-interface provided by a new service into the monolith-provided user interface; and *Micro Frontends* consist of taking a microservice-based approach to frontend development.

Branch by Abstraction

When to use: When changes to the existing codebase will likely take time to carry out, and we want to avoid any disruption. It assumes we can change the code of the current system.

How to apply it: Develop, in the monolith, an alternative implementation for a module so that the new and old implementations comply with the same interface/abstraction. The new implementation will typically call a new external service, whereas the old would implement the functionality internally. At some point, switch from the old implementation to the new implementation.

Parallel Run

When to use: When the failure of the functionality being worked implies a high risk for the business.

How to apply it: With a parallel run, rather than calling either the old or the new implementation, we call both, allowing us to compare the results to ensure they are equivalent. Despite calling both implementations, only one is considered the source of truth at any given time. Typically, the old implementation is considered the source of truth until the ongoing verification reveals that we can trust the new implementation.

Decorating Collaborator

When to use: When we want to trigger some behavior based on something happening inside the monolith, but we want to avoid changing the monolith itself. This technique assumes we can intercept responses returned by the monolith before they reach the clients.

How to apply it: Use the *decorator* pattern to make it appear from the outside that we have added new logic to the monolith without actually changing it. The technique consists of routing client requests to a proxy which will allow the call to reach the monolith, as it normally would, and based on the result of this call, call out to a new microservice, and possibly modify the result in some way before returning it to the client.

Change Data Capture

When to use it: When we need to react to a change in data in the monolith but cannot intercept this change at the perimeter of the monolith (e.g., using a decorator) or change its implementation.

How to apply it: Rather than trying to intercept and act on calls made into the monolith, we react to changes made in a datastore. The underlying capture system will be coupled to the monolith's datastore, and can rely on database triggers, transaction log pollers (usually a file, into which is written a record of all the changes that have been made) or batch delta copier (a program that regularly scans the database in question for what data has changed).

Change code dependency to service call

When to use it: When a software system is decomposed into a set of smaller services to use a microservices architectural style, some components in the system will act as dependencies to other services or components.

How to use it:

Try to keep the services' code as separate as possible. When services depend on the same piece of code, there is a chance that changes to that code motivated by the needs of one service might negatively affect the other service. If this is code shared as a library that rarely changes (e.g., a string manipulation library) it is reasonable to have different services depend on it. On the other hand, if this is code related to internal entities, or entails different scalability needs, we share it as a new service so that it can be changed independently and gradually, or scaled independently.

18. **4.1. Tell us to which degree you agree with the following statements in the scope of the techniques presented below:** *

"In the migration project(s) in which I have been involved the use of this technique was important"

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Strangler Fig					
UI Composition					
Branch by Abstraction					
Parallel Run					
Decorating Collaborator					
Change Data Capture					
Change code dependency to service call					

19. 4.2. Which of these techniques do you often use together? Do you use them "before", "after" or "as a step of"?

In answering this question, consider the techniques above but also any other that might have been used in the migrations that you have been involved in.

20. 4.3. In your experience, what are the main challenges of applying these techniques?

5. Refactoring Techniques - Decomposing the Database

In this section, we explore a selection of refactoring techniques for decomposing the database. The goal is to understand how often these techniques are used, how they are used, their challenges and which techniques are used together.

Please take the next ~5min to read about each one of the refactoring techniques below.

Database View

When to use it: We want a single source of data for multiple services, but it is impractical to change all clients to point to the new service(s). For clients that will keep considering the monolith as their source of data, we want to mitigate concerns regarding coupling to specific parts of the data schema. There are more clients *reading* data than *writing* it.
How to use it: For clients that only read data, create a dedicated schema that hosts views looking like the old schema, and that are assembled from data now owned by the new service(s). Have clients that only read data point at those views instead of the original tables. We will need to change only the clients that need to write data so that they start using the new services directly, instead of the monolith.

Database Wrapping Service

When to use it: When multiple clients depend on the same datastore, but it is just too hard to consider pulling apart the underlying schema.
How to use it: Hide the database behind a service that acts as a thin wrapper and make clients depend on this new service instead of the database. Encourage developers writing the different client applications to think of this new service as someone else's and start storing their own data locally.

Database-as-a-Service Interface

When to use it: When we have clients that really just need a database to query. This could be because they need to query large amounts of data, or because external parties are already using toolchains that require a SQL endpoint to work against.
How to use it: Create a dedicated database to be exposed as a read-only endpoint, and have this database populated when the data in the underlying database changes. Take care to separate the database we expose from the database we use inside our service boundary. A mapping engine takes changes in the internal database, and works out what changes need to be made in the external database. When our internal database changes structure, the mapping engine will need to change to ensure that the public-facing database remains consistent.

Aggregate Exposing the Monolith

When to use it: As we extract services from the monolith, we often realize that some of the data should become part of the new services, and some of it should stay where it is. When the monolith still owns the data we want to access, we will need to allow the new services the means to access this data.
How to use it: Make explicit what information the new services need from the monolith by exposing it via dedicated service endpoints. The monolith still owns the data and decides

what state changes are valid; it is not just like a wrapper around a database. Beyond just exposing data, expose operations that allow external parties to query the current state and to make requests for new state changes.

Change Data Ownership

When to use it: As we extract services from the monolith, we often realize that some of the data should become part of the new services, and some of it should stay where it is. When the newly extracted service encapsulates the business logic that changes some data, that data should be under the new service's control.

How to use it: Move the data from the monolith over into the new service. Change the monolith to treat the new service as the source of truth for the data it now owns, calling it to read or change data when needed. When doing this, we may have to consider the consequences of breaking foreign-key constraints and transactional boundaries.

Synchronize Data in Application

When to use it: When we want to split the schema of the monolith into two separate datastores, in preparation for extracting a new service.

How to use it: Do this in a sequence of three steps. The first step is to create the new datastore and bulk-import data from the monolith—only the part of the schema we are interested in splitting needs to be imported. The second step consists of making the monolith write the same data to both datastores, ensuring they are kept in-sync, and that writes to the new datastore are working well. In the third step, with a simple change to the monolith, we make the new datastore the source of truth and ensure that the reads also work. As we still write to both datastores, we can easily fallback to reading from the old datastore if any issue is found.

Tracer Write

When to use it: When we need to move the ownership of some data from the monolith to a new service in an incremental fashion, tolerating there being two sources of truth during the migration.

How to use it: First, identify the part of the monolith data schema whose ownership we want to change and the service that will host it. Then, ensure this data is kept in-sync between the monolith and the new service. This can be achieved in a number of ways: a) Allowing writes only on one of the sources of truth, and making it in-charge of replicating the change in the other one; b) Have clients send writes to both sources; or c) Allow clients to send writes to either source, and synchronize data behind the scenes. Once all clients use the new source of truth, the old source of truth can be retired.

Note: *Synchronize Data in Application* focuses on splitting the schema, and this technique focuses on moving a part of the schema to the ownership of a previously created service.

Split Table

When to use it: When we are using a relational database and we find that columns of a single table need to be split across two or more service boundaries.

How to use it: First, split the columns of the table apart in the existing schema, placing one (or more) of them in a new table. Then, move the new table to a different service. If there was code updating columns that are now owned by different services, the monolith will need to call the new service to update the split column.

Move Foreign-Key Relationship to Code

When to use it: When we are using a relational database, and we are moving some functionality to a new service but we want to keep some of the data used by that functionality in the monolith.

How to use it: With a monolithic system, joining different relational tables is often done by database queries, but when one of those tables is moved to a new service, this new service may need to fetch from the monolith any data that it requires that is still kept there, and join it "in memory" with the data it owns, rather than "in the database".

21. **5.1. Tell us to which degree you agree with the following statements in the scope of the techniques presented below:** *

"In the migration project(s) in which I have been involved the use of this technique was important"

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Database View					
Database Wrapping Service					
Database-as-a-Service Interface					
Aggregate Exposing the Monolith					
Change Data Ownership					
Synchronize Data in Application					
Tracer Write					
Split Table					
Move Foreign-Key Relationship to Code					

22. 5.2. Which of these techniques do you often use together? Do you use them "before", "after" or "as a step of" each other?

In answering this question, consider the techniques above but also any other that might have been used in the migrations that you have been involved in.

23. 5.3. In your experience, what are the main challenges of applying these techniques?

6. Refactoring Techniques

The following questions are general to all refactoring techniques.

24. 6.1. What other *techniques* or *patterns* have you used in the migration process that were not mentioned in this survey?

25. 6.2. In your opinion, which are the most important challenges faced in the migration process? *
Tick at most 9 options.

Marcar tudo o que for aplicável.

- [ ] Database migration and data store splitting
- [ ] Dealing with data consistency
- [ ] Communication among services
- [ ] Ensuring code maintainability
- [ ] Decoupling services from the monolith
- [ ] Service orchestration
- [ ] Multitenancy
- [ ] Microservices statefulness
- [ ] Continuous deployment/integration
- [ ] Effort estimation
- [ ] Expected long-term return on investment (ROI)
- [ ] Team organization
- [ ] Infrastructure to support microservices
- [ ] Selecting infrastructure patterns
- [ ] Ensuring reliability
- [ ] Ensuring scalability
- [ ] Ensuring security
- [ ] Outra: _____

7. Evaluation

26. 7.1. Do you usually evaluate the result of a decomposition? How? *

_____

_____

_____

_____

_____

27. 7.2. What quality attributes do you assess when evaluating the decomposition result? *

Tick at most 4 options.

Marcar tudo o que for aplicável.

- [ ] Maintainability
- [ ] Performance, Efficiency
- [ ] Business-related indicators
- [ ] Reusability
- [ ] Security
- [ ] Scalability
- [ ] Availability
- [ ] Team Agility
- [ ] Outra: _____

28. 7.3. Why?

_____

_____

_____

_____

_____

29. 7.4. What *metrics* would you use to evaluate these quality attributes? Why? * *Example: I use the throughput (e.g., requests per second) to measure the efficiency.*

_____

_____

_____

_____

_____

30. 7.5. In which environments do you evaluate a decomposition result? *
Tick all that apply

Marcar tudo o que for aplicável.

- [ ] Development
- [ ] Staging
- [ ] Production

31. 7.6. What kind of inputs do you use to evaluate a decomposition result? *
Tick all that apply

Marcar tudo o que for aplicável.

- [ ] Functional tests
- [ ] Simulation
- [ ] Production

B.2 Survey Results

B.2.1 Areas respondents have been working on

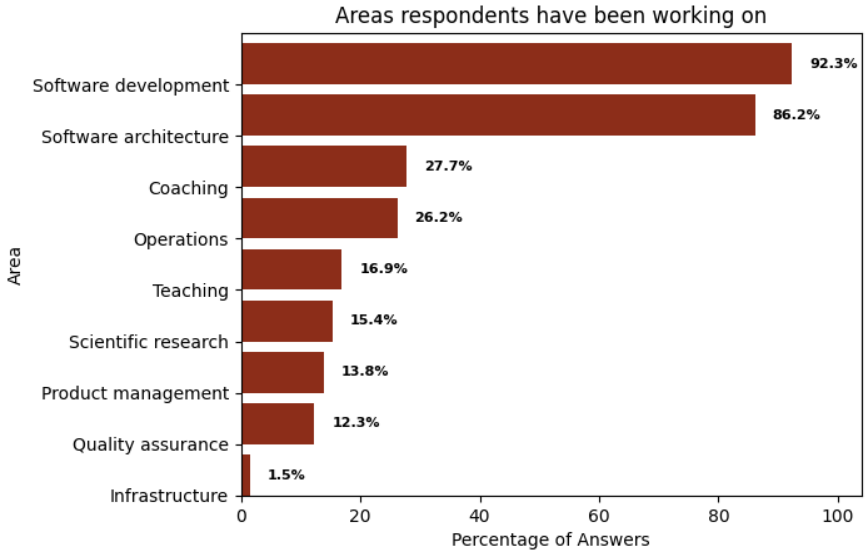


Figure B.1: Areas respondents have been working on the past 5 years

B.2.2 Professional experience in Microservice

Figure B.2: Professional experience in Microservice

B.2.3 Number of migration projects respondents were involved in

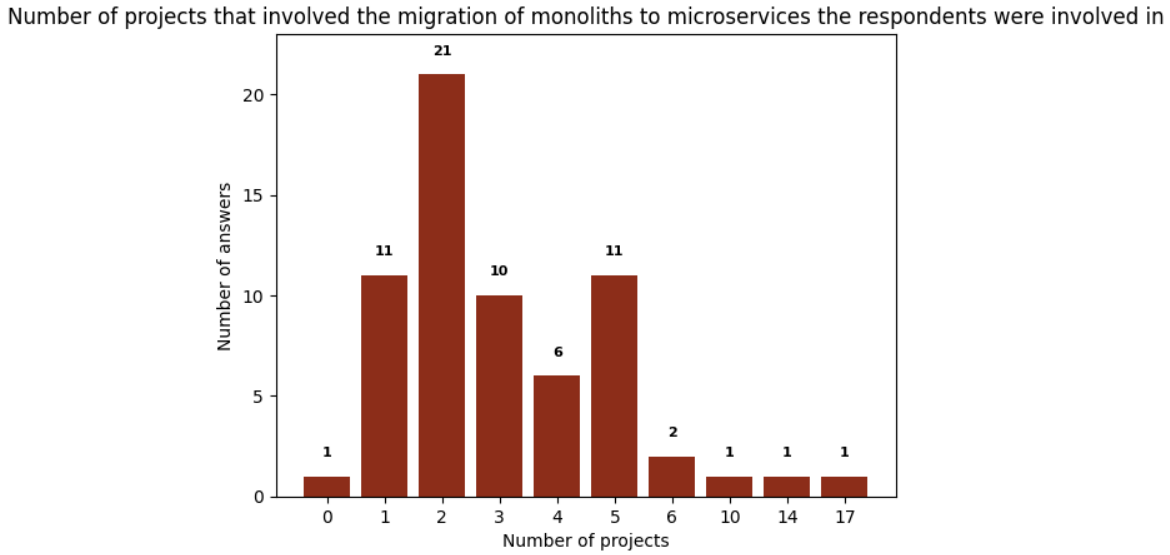


Figure B.3: Number of projects that involved the migration of monoliths to microservices the respondents were involved in

B.2.4 Monthly active users of the migrated systems

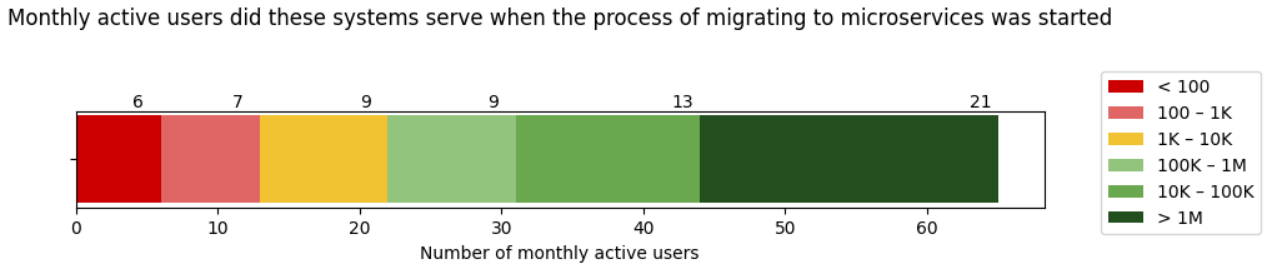


Figure B.4: Monthly active users did these systems serve when the process of migrating to microservices was started

B.2.5 People working on the migrated systems

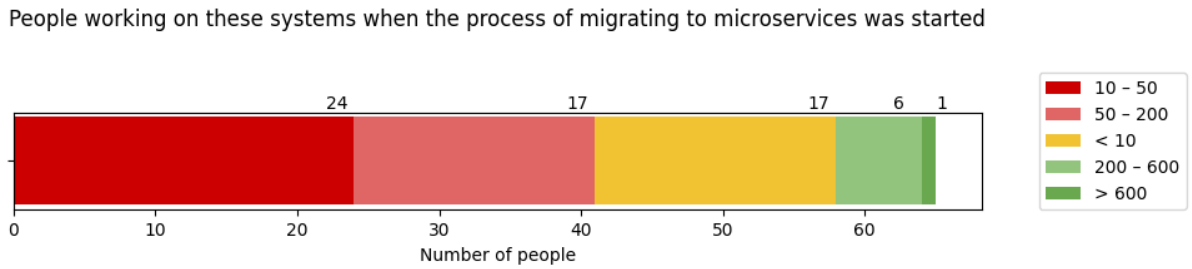


Figure B.5: People working on these systems when the process of migrating to microservices was started

B.2.6 Guidance when migrating a monolith

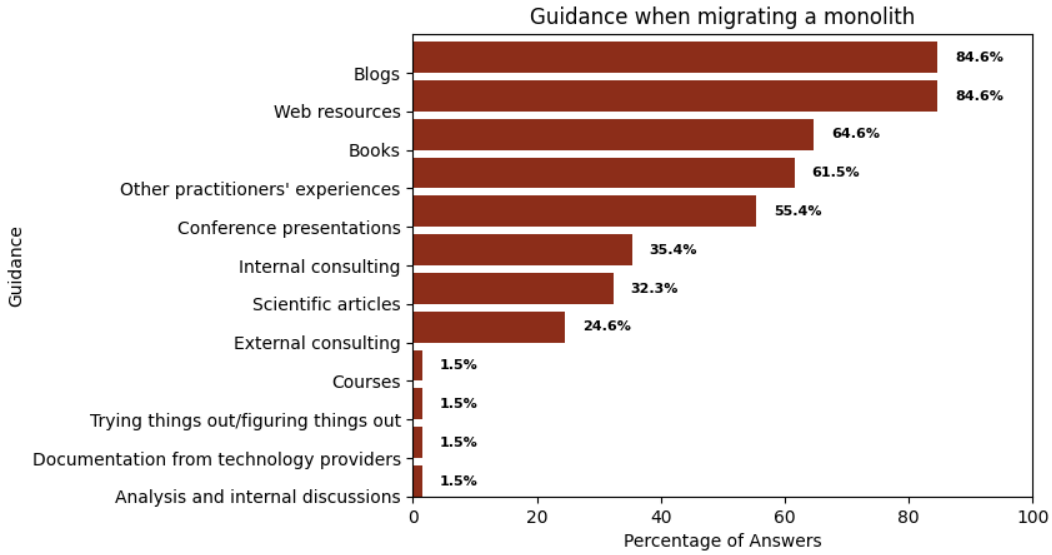


Figure B.6: Where do respondents look for guidance when migrating a monolith

B.2.7 Common strategies to plan the migration regarding the evolution of the product

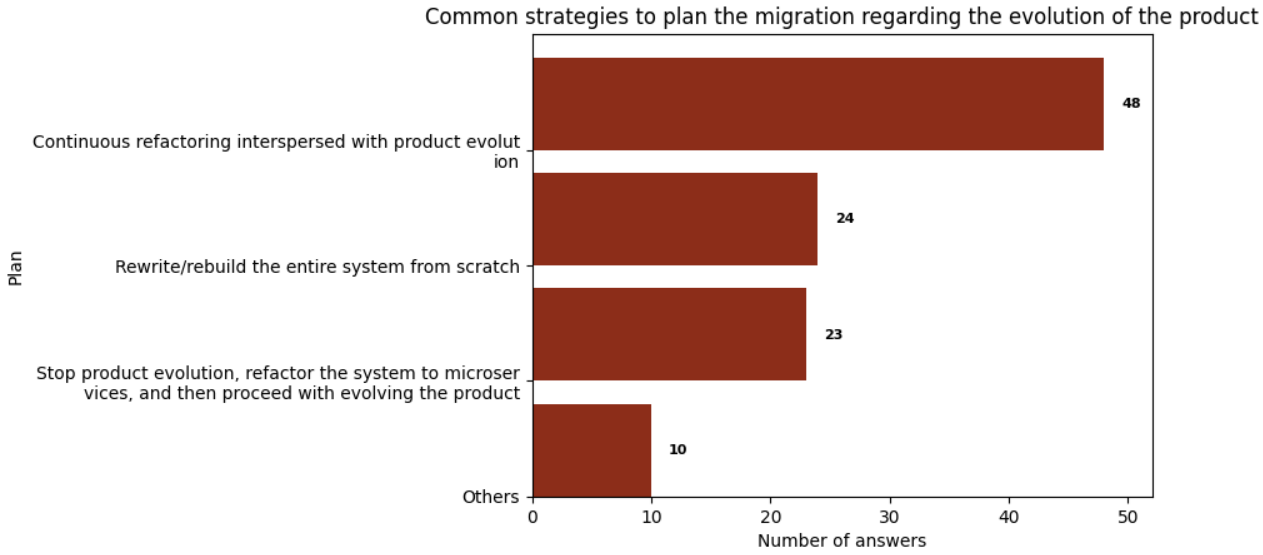


Figure B.7: Common strategies to plan the migration regarding the evolution of the product

B.2.8 Assistance by automated or semi-automated tool

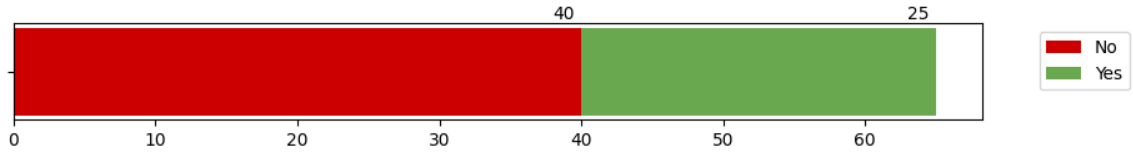


Figure B.8: Assistance by automated or semi-automated tool

B.2.9 Challenged faced in the migration

Figure B.9: Most important challenges faced in the migration process

B.2.10 Environments where the decomposition result is evaluated

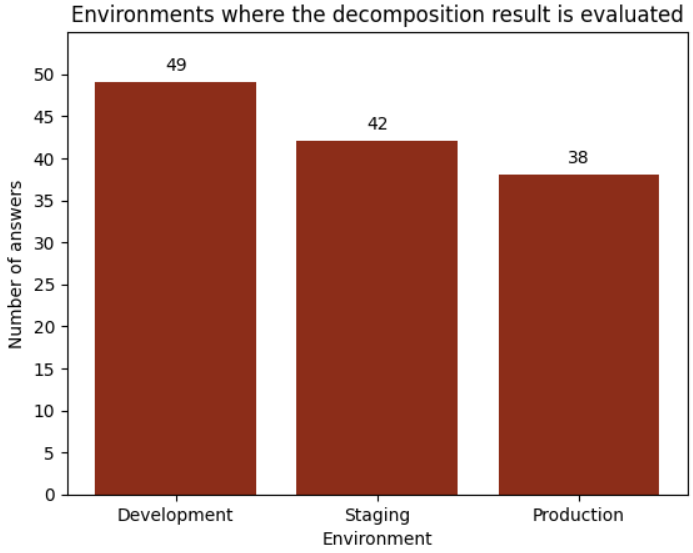


Figure B.10: Environments where the decomposition result is evaluated

B.2.11 Inputs used to evaluate a decomposition result

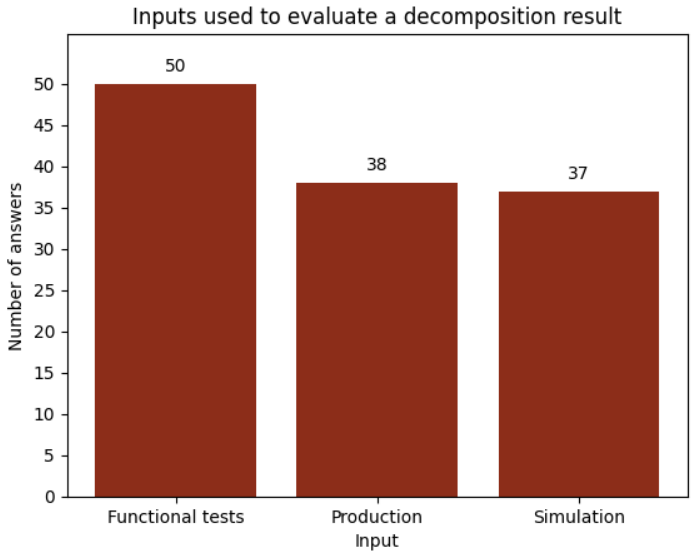


Figure B.11: Inputs used to evaluate a decomposition result

Appendix C

Catalogue of refactorings: A guide for the migration towards a microservices architecture

The following catalogue corresponds to the full catalogue addressed in Chapter 5.

This catalogue guides the refactoring process toward a microservices architecture. It contains large-scale refactorings, the smaller-scale refactorings that compose it, and the steps to implement the smaller-scale ones, like a recipe to follow, touching very different topics of the migration process across eight different chapters:

- Chapter 1 (Appendix C.1) describes refactoring sequences for breaking existing dependencies between microservices.
- Chapter 2 (Appendix C.2) outlines infrastructure changes needed for a microservices design.
- Chapter 3 (Appendix C.3) explores microservice deployment and orchestration strategies.
- Chapter 4 (Appendix C.4) describes the properties of microservices and how the refactorings in this catalogue help to achieve them.
- Chapter 5 (Appendix C.5) outlines different refactoring techniques that lead to intermediate states of refactoring when we have constraints and limitations during our refactoring process.
- Chapter 6 (Appendix C.6) contains sequences of common refactorings.
- Chapter 7 (Appendix C.7) tackles additional challenges when transferring a system to a microservices design.
- Chapter 8 (Appendix C.8) discusses how to prepare for the migration process.

The refactorings are presented using a structure similar to the one proposed by Martin Fowler and Kent Beck in their 2nd edition of the book *"Refactoring: Improving the Design of Existing Code"* [33]. They are composed of four sections:

- **Title** corresponds to the name of the refactoring. This name is designed to be as self-explanatory as in any code refactoring naming.
- **Motivation** describes a scenario, the set of observed conditions that may lead to the need to apply the refactoring. We can see it as the driving cause.
- **Mechanics**, presents a step-by-step guide of the refactoring implementation.
- **Example** exemplifies the refactoring application, either by using some code snippets, descriptions or schemas.

C.1 Breaking Dependencies

To separate the system into microservices, we need to identify the dependencies between the clusters of classes that will make our intended microservices and "break" (cf. Section 5.1) the dependencies between those clusters so that the microservices can function independently.

We can consider multiple types of dependencies. This catalogue chapter (Appendix C.1) describes how to break the different dependencies and how the code should be refactored to achieve that. It is common to find these dependencies types together, so some refactorings described below may link to others and form a sequence of refactorings to solve the dependencies between microservices.

C.1.1 Change Local Method Call Dependency to a Service Call

Motivation

When splitting the monolith into microservices, it is prevalent to have code dependencies in the form of a method invocation between components. As the classes in question will become part of different services, method invocations often have to be refactored to service calls in the microservices architecture. This service call should be made with a protocol that can be synchronous or asynchronous, depending on the requirements and goals.

When we do not want a service waiting for the other to respond and an instant response is not required because we do not need that information immediately, asynchronous calls are usually a good option. This is the case when we accept having eventual consistency when it comes to data operations. It is especially common when a service call triggers other service calls, and we do not want the first service to be busy waiting for these calls to complete. It helps in scalability and availability. It is common to implement it through an asynchronous remote procedure call or messaging with a publisher/subscriber protocol, where services publish messages to a message broker, and the subscriber consumes the messages when available to process them.

Asynchronous calls are not the best solution to ensure we are reading accurate data and need consistency in the moment of the action, as consistency in asynchronous calls is eventual; in this case, a synchronous request/response protocol is usually preferred.

Making this solution asynchronous allows for better scalability and fault tolerance as the services become decoupled.

Mechanics

1. Decide the communication strategy and make the initial configurations to use it.
   (a) Synchronous (using strategies like REST or RPC, for example)
      i. Store the necessary information (e.g. URL) to make the remote calls to the microservice.
   (b) Asynchronous (with technologies like Mosquitto, RabbitMQ, Apache Kafka, etc.): using strategies like Event Sourcing or some form of asynchronous RPC.
      i. Set up a message broker/event bus.
      ii. Create a topic.
2. Configure the microservice that makes the invocation to use the communication strategy.
   (a) Synchronous - Change the method calls to and from local components to be remote calls to a different service:
      i. Create an interface with the declaration of the identified methods.
      ii. Create a class that implements that interface and makes the service calls, a Request Class.
   (b) Asynchronous
      i. This microservice will act like a subscriber, so make it subscribe to the topic you have created.
3. Configure the microservice that has the method to use the communication strategy.
   (a) Synchronous - arrange the microservice owning the method to respond to this communication protocol, creating an API to respond to the service calls. Example:
      i. Create a class that defines the resource paths for the requests and processes them producing a response.
      ii. Add methods to the class to perform the actions required by the service calls.
   (b) Asynchronous
      i. This microservice will act like a publisher, so you make it push the messages it wants to communicate in the topic created.

Note: Make sure to guarantee fault tolerance (check **Chapter 7 (Appendix C.7)**).

Example

Let us imagine we have in the monolith a set of classes for managing orders (candidate to become a new *OrderManagement* microservice) and another set of classes for managing inventory (candidate to become a new *Inventory* microservice). The former includes a class *OrderProcessor* that makes a local method call to the *updateInventory* method defined by the class *InventoryService* of the latter, as shown in Listing C.1.

```
1  public class OrderProcessor {
2      private final InventoryService inventoryService;
3
4      public OrderProcessor(InventoryService inventoryService) {
5          this.inventoryService = inventoryService;
6      }
7
8      public void processOrder(Order order) {
9          inventoryService.updateInventory(order);
10         // Other processing logic
11     }
12 }
13
14 public class InventoryService {
15     private final InventoryManager inventoryManager;
16
17     public InventoryService(InventoryManager inventoryManager) {
18         this.inventoryManager = inventoryManager;
19     }
20
21     public void updateInventory(Order order) {
22         inventoryManager.updateInventory(order);
23     }
24 }
```

Listing C.1: The *OrderProcessor* class from proposed *OrderManagement* microservice makes a local method call to the method *updateInventory* of the *InventoryManager* call of the proposed *Inventory* microservice.

In Listing C.1, we can see that the *OrderProcessor* class performs a local call to the method of *InventoryManager*, and as we want to create two separate microservice, we need to break this dependency. For that, the method calls should become remote service calls. An example for a synchronous solution using REST is shown in Listing C.2.

```
1 //OrderManagement Microservice
2 public interface InventoryService {
3     void updateInventory(Order order);
4 }
```

```
5
6  @Service
7  public class RemoteInventoryService implements InventoryService {
8      private final RestTemplate restTemplate;
9      private final String inventoryServiceUrl;
10
11     public RemoteInventoryService(RestTemplate restTemplate, @Value("${inventory.
           service.url}") String inventoryServiceUrl) {
12         this.restTemplate = restTemplate;
13         this.inventoryServiceUrl = inventoryServiceUrl;
14     }
15
16     @Override
17     public void updateInventory(Order order) {
18         String url = inventoryServiceUrl + "/update";
19         restTemplate.postForObject(url, order, Void.class);
20     }
21 }
22
23 public class OrderProcessor {
24     private final InventoryService inventoryService;
25
26     public OrderProcessor(InventoryService inventoryService) {
27         this.inventoryService = inventoryService;
28     }
29
30     public void processOrder(Order order) {
31         inventoryService.updateInventory(order);
32         // Other processing logic
33     }
34 }
35
36 // Inventory Microservice
37 @RestController
38 @RequestMapping("/api/inventory")
39 public class InventoryController {
40     @PostMapping("/update")
41     public ResponseEntity<Void> updateInventory(@RequestBody Order order) {
42         // Update inventory logic
43         // ...
44         return ResponseEntity.ok().build();
45     }
46 }
```

Listing C.2: *OrderProcessor* uses the *InventoryService* interface to make a synchronous service call to the *Inventory* service using REST.

In the *OrderManagement* microservice, we created the *InventoryService* interface that defined the *updateInventory* method and created the implementation of this interface, the *RemoteInven-*

toryService class. This class uses *RestTemplate* to send a POST request to the inventory service URL.

The *OrderProcessor* class remains the same, although now it depends on the *InventoryService* interface for updating the inventory.

In the *Inventory* microservice, we created the *InventoryController* class that handles the HTTP POST request for updating the inventory. The *updateInventory* method contains the logic to update the inventory based on the received order.

This refactoring changes the direct local method call dependency to a synchronous RESTful API call, allowing the *OrderManagement* microservice to communicate with the *Inventory* microservice via remote synchronous service calls using HTTP requests.

The asynchronous solution using *Apache Kafka* is shown in Listing C.3.

```
// OrderManagement microservice
    public class OrderEvent {
    private Order order;

    // Constructors, getters, and setters
}

public class OrderUpdatedEvent {
    private Order order;

    // Constructors, getters, and setters
}


@Component
public class KafkaEventProducer {
    private final KafkaTemplate<String, OrderEvent> kafkaTemplate;
    private final String topic;

    public KafkaEventProducer(KafkaTemplate<String, OrderEvent> kafkaTemplate,
        @Value("${kafka.topic}") String topic) {
        this.kafkaTemplate = kafkaTemplate;
        this.topic = topic;
    }

    public void publishOrderEvent(OrderEvent orderEvent) {
        kafkaTemplate.send(topic, orderEvent);
    }
}

@Component
public class OrderEventListener {

    private final InventoryService inventoryService;
```

```
34
35      public OrderEventListener(InventoryService inventoryService) {
36          this.inventoryService = inventoryService;
37      }
38
39      @KafkaListener(topics = "${kafka.topic}")
40      public void handleOrderEvent(OrderEvent orderEvent) {
41          inventoryService.updateInventory(orderEvent.getOrder());
42      }
43  }
44
45  public class OrderProcessor {
46      private final KafkaEventProducer kafkaEventProducer;
47
48      public OrderProcessor(KafkaEventProducer kafkaEventProducer) {
49          this.kafkaEventProducer = kafkaEventProducer;
50      }
51
52      public void processOrder(Order order) {
53          OrderEvent orderEvent = new OrderEvent(order);
54          kafkaEventProducer.publishOrderEvent(orderEvent);
55          // Other processing logic
56      }
57  }
58
59  // Inventory microservice
60  @Service
61  public class InventoryService {
62      public void updateInventory(Order order) {
63          // Update inventory logic
64          // ...
65      }
66  }
```

Listing C.3: *OrderProcessor* uses the *InventoryService* interface to make an asynchronous service call to the *Inventory* service using *Apache Kafka* and Events.

We created the event class, *OrderEvent*. We then created a *Kafka* event producer, implemented an event listener to process the order events and updated the *OrderProcessor* to use the *Kafka* event producer.

The *Kafka Event Producer* publishes the *OrderEvent* to the *Kafka* option, while the *OrderEventListener* listens to the *Kafka* topic and invokes the *updateInventory* method of the *Inventory* service, to perform the actual update. This service contains the logic to update the inventory.

The *OrderProcessor* class starts using the *Kafka Event Producer* to publish the *OrderEvent* so the events can be handled asynchronously instead of directly calling the *Inventory* service.

By making this solution asynchronous, it allows for better scalability and fault tolerance as the services become decoupled.

C.1.2 Move Foreign-key relationship to code

Motivation

When two entities are related and dependent on one another, their relationship is frequently characterised by a foreign-key relationship between the database tables representing each entity. One-to-one, Many-to-One, and One-to-Many relationships are possible.

When we extract a service and realise that these entities should be in different microservices, the database tables that describe them should belong to different database schemas, as each microservice should have its database.

Because one service will own the table containing the foreign key and another will own the table from which the foreign key comes, we must break the database and eliminate the foreign-key relationship. Therefore, as the constraint is no longer in the database, we must move this relationship to the code itself.

We have to keep in mind that sometimes the migration needs to carry more information than a single database table. When it happens, it is not always clear that those dependencies need to be migrated. Besides, integration layers can sometimes be required for functional decoupling.

Mechanics

The following steps can either be performed after breaking the code or with the code breakage in mind. Whenever services are mentioned, they are mentioned as what will be the end service division, but they do not have to be already implemented.

1. Remove the foreign-key constraint from the table that stores it.

2. In the class of entity (database table) that used to have the foreign-key constraint, create an instance variable that represents the other entity involved in the said relationship and create a column for that variable in this entity table. This variable will no longer be a foreign key but a filter of the select query to retrieve data.

3. Separate the tables into the databases of the different owners (at this moment, this might be more conceptual, but in the future, this will represent the databases of the different microservices).

4. Create an interface for each of these databases that implements the methods of data manipulation.

5. Identify the methods that use/manipulate data from different databases and change them to use the newly created interfaces.

6. When you separate the services, don't forget to use the previous refactoring to "Change local method call dependency to a service call"(C.1.1) to change these local methods to service calls using the primary key as a parameter.

Note: We may need to remove code annotations when using specific programming languages, frameworks or ORMs that use it. The way we join the information of these two entities is no longer through a join query, so data consistency has to be a concern. Do not forget to implement mechanisms to guarantee data integrity and consistency (check **Chapter 7 (Appendix C.7)**). Additionally, we should be aware that the latency of requests increases as we transform the database calls into service calls.

Example

Suppose we have two entities, *Order* and *Customer*, with a foreign-key relationship where an *Order* belongs to a *Customer*. The *Order* entity has a *ManyToOne* relationship with the entity *Customer*, using the *customer_id* foreign-key column for the association, as can be seen in Listing C.4.

```
@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "customer_id")
    private Customer customer;

    // Other properties, getters, and setters
}

@Entity
@Table(name = "customers")
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Other properties, getters, and setters
}

@Service
public class OrderService {
    private final OrderRepository orderRepository;

    public OrderService(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }

```

```
33      public void processOrder(Order order) {
34          // Perform business logic
35
36          Customer customer = order.getCustomer();
37          // Use customer data for processing
38
39          orderRepository.save(order);
40      }
41  }
42
43  @Repository
44  public interface OrderRepository extends JpaRepository<Order, Long> {
45      // Order-related methods for data manipulation
46  }
```

Listing C.4: *Order* has a foreign-key constraint with *Customer*

The *OrderService* class depends on the *OrderRepository* class for data access and manipulation. In the *processOrder* method, the *Customer* entity is accessed directly through the *getCustomer* method.

Listing C.5 shows the code after applying the refactoring.

```
1  @Entity
2  @Table(name = "orders")
3  public class Order {
4      @Id
5      @GeneratedValue(strategy = GenerationType.IDENTITY)
6      private Long id;
7
8      private Long customerId;
9
10     // Other properties, getters, and setters
11 }
```

Listing C.5: The foreign key constraint between *Order* and *Customer* is now in the code.

We removed the foreign-key constraint from the *Order* table referencing the *Customer* table. In the *Order* class, we created an instance variable *customerId* to represent the association with the *Customer* entity. In the *Order* table, we added a column for the *customerId*. Each table goes to a separate database; the *Order* table goes to *orders_db*, and the *Customer* table goes to *customers_db*.

We created the interfaces for data manipulation. *OrderRepository* defines the methods for manipulating the *Order* entity in the *orders_db* database. *CustomerRepository* defines the methods for manipulating the *Customer* entity in the *customers_db* database. We identified that the method *processOrder* interacted with both entities, so we modified it to use the respective interfaces.

C.1.3 Replicate Data Across Microservices

Motivation

Sometimes, different microservices need the same data. Still, if they access the same data source, they won't be totally independent, as one microservice will also be managing the data of another.

To solve this, each service can have its own dedicated database with replication to the shared data source. Ideally, in a microservices architecture, this should occur with no distributed transaction, assuming eventual consistency; however, synchronous replication is also possible. One of the services is the data owner and source of truth. The other simply holds a copy of the data it needs to access and manipulate.

Mechanics

1. Split the database by deciding which service will be the owner of the shared data.
2. There are a few strategies to replicate the data:
   (a) Using mechanisms of the database engine, you can create one or more replication channels between it and the shared data source.
   (b) Using event sourcing, a method of storing (or communicating) data which facilitates data replication because events may be easily repeated. It is a way to keep eventual consistency. It holds events that are frequently objects, and because event sourcing does not need to know its consumers, other technologies can be utilised concurrently (for more on event sourcing, check Martin Fowler's article here).
   (c) Using "Change Data Capture" refactoring (check refactoring C.5.12)

Example

In this example, we will show how to use Event Sourcing to replicate data across microservices.

We will use the example of the entity *Order* containing a relationship with the entity *User* of the type ManyToOne and these two entities will belong to different microservices. Since the entity *User* needs to know its order, one solution is to replicate the data of the entity *Order* for the *User* using Event Sourcing.

To apply event sourcing, we have to identify the entity for data replication, which is the *Order* in this case; implement event sourcing; publish events; subscribe to events and update the local data. Listing C.6 shows the *OrderManagement* microservice code to implement event sourcing.

```
1 @Service
2 public class OrderService {
3     private final EventPublisher eventPublisher;
4     private final OrderRepository orderRepository;
5
```

```
    public OrderService(EventPublisher eventPublisher, OrderRepository
        orderRepository) {
        this.eventPublisher = eventPublisher;
        this.orderRepository = orderRepository;
    }

    public Order createOrder(OrderDTO orderDTO) {
        // Create the order entity
        Order order = new Order(orderDTO);

        // Save the order in the repository
        Order savedOrder = orderRepository.save(order);

        // Publish the order created event
        OrderCreatedEvent event = new OrderCreatedEvent(savedOrder.getId());
        eventPublisher.publish(event);

        return savedOrder;
    }

    // Other methods for updating and deleting orders
}
```

Listing C.6: The *OrderManagement* microservice publishes *OrderCreatedEvent* events every time a new order is created

The *OrderManagement* microservice is responsible for creating and managing orders, so every time a new order is created, it publishes an *OrderCreatedEvent* that contains the newly created order using the *EventPublisher*. The microservices interested in replicating the *Order* data can subscribe to the *OrderCreatedEvent* and update their local order records accordingly. Listing C.7 shows the code in the *User* side, the microservice that wants to replicate the data.

```
@Service
public class UserService {
    private final EventSubscriber eventSubscriber;
    private final UserRepository userRepository;

    public UserService(EventSubscriber eventSubscriber, UserRepository
        userRepository) {
        this.eventSubscriber = eventSubscriber;
        this.userRepository = userRepository;
        eventSubscriber.subscribe(OrderCreatedEvent.class, this::
            handleOrderCreatedEvent);
    }

    private void handleOrderCreatedEvent(OrderCreatedEvent event) {
        // Retrieve the order details based on the event
```

```
14          Order order = getOrderDetails(event.getOrderId());
15
16          // Update the user associated with the order
17          User user = userRepository.findById(order.getUserId()).orElse(null);
18          if (user != null) {
19              user.addOrder(order);
20              userRepository.save(user);
21          }
22      }
23
24      private Order getOrderDetails(String orderId) {
25          // Retrieve the order details from the appropriate source (e.g., call the
                Order service)
26          // ...
27          return new Order(orderId, "User123", /* other order details */);
28      }
29
30      // Other methods for user management
31  }
```

Listing C.7: The *User* microservice listens to *OrderCreatedEvent* events to update its own data storage

As the *User* microservice is interested in replicating *Order* data, it subscribes to the *OrderCreatedEvent* using *EventSubscriber* and the *handleOrderCreatedEvent* method is invoked whenever a new order is created.

Inside this method, the *Order* details are retrieved based on the event. The associated *User* is fetched, and if it exists, the *Order* is added to its list of orders, and this way, it keeps the user's order data updated.

This way, each microservice can independently handle the received events and update its own data storage, ensuring data consistency.

C.1.4 Split Database Across Microservices

Motivation

When extracting a service, we commonly find that some monoliths aggregate in the same database table data that will further need to be split across different microservices, as different microservices access the same database table. Therefore, splitting a monolithic database is not so trivial.

When deciding to choose which service is going to be the owner of the data, we should keep in mind the business requirement and what makes sense for the system in our hands. Also, we should be able to clearly see which data fields remain master/slave on each service.

Mechanics

1. Separate into each microservice database only the tables that are only accessed/manipulated by that microservice.
2. Find the columns in the table used by the different newly defined microservices.
3. Analyse which is the case of each table and what solution better fits your system's requirements:
   (a) Two microservices access the same database table but do not update the same columns
      i. You can replicate the data for both microservices using **"Replicate Data Across Microservices"** (C.1.3) and use a data replication mechanism to keep it consistent or
      ii. Decide to which of these microservices each column belongs.
      iii. Split this table inside the monolith schema between the two tables belonging to the different components that will soon be microservices.
      iv. In each component, include the corresponding table and adapt the code to use that table.
      v. If the different microservices interact with what used to be foreign keys on the monolith schema, use the **"Move Foreign-key relationship to code"** refactoring (C.1.2).
   (b) Two microservices use the same database table and update the same columns
      i. You can replicate the data for both microservices using **"Replicate Data Across Microservices"** (C.1.3) and use a data replication mechanism to keep it consistent or
      ii. Decide which microservice should own this data.
      iii. Make the other microservice make a service call to update this column.
         A. To perform this incrementally, we can first make the change in the monolith to the soon microservice that does not own the data, make the update through a method call, and then, when creating the microservices, use the refactoring **"Change local method call dependency to a service call"** (C.1.1).

Note: Guarantee data consistency (check **Chapter 7 (Appendix C.7)**)

Example

We will present an example for option (b) of the Mechanics without data replication.

Suppose we have two microservices, the *Inventory* microservice and the *OrderManagement* microservice and that both use the same database table called *Product*. If first need to identify the microservice that should own the data, and in this case, we believe the *Inventory* microservice is more suitable. Then we need to define a service API for the *Inventory* microservices with methods

to update the specific columns related to inventory management. Then we must remove the direct database updates from *OrderManagement* microservice to the shared columns in the *Product* table and change them to make service calls to the API provided by *Inventory* microservice, whenever updates to the shared columns related to inventory management are required. Listing C.8 shows the code on the *Inventory* microservice side.

```
@RestController
@RequestMapping("/api/inventory")
public class InventoryController {
    private final InventoryService inventoryService;

    public InventoryController(InventoryService inventoryService) {
        this.inventoryService = inventoryService;
    }

    @PutMapping("/product/{productId}/stock")
    public ResponseEntity<String> updateProductStock(@PathVariable("productId")
        Long productId, @RequestParam("stock") int stock) {
        // Call the service method to update the stock of the product
        inventoryService.updateProductStock(productId, stock);
        return ResponseEntity.ok("Product stock updated successfully");
    }
}
```

Listing C.8: *Inventory* microservice code - the owner of the data)

Inventory microservice owns the shared data related to product inventory and exposes and API endpoint *'api/inventory/product/productId/stock'* to update the stock of a product.

Listing C.9 shows the code on the *OrderManagement* microservice side.

```
@Service
public class OrderService {
    private final RestTemplate restTemplate;
    private final String inventoryServiceUrl = "http://localhost:8082/api/inventory
        /product";

    public OrderService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public void placeOrder(Order order) {
        // Perform order placement logic

        // Make a service call to the Inventory Service to update the product stock
        String updateUrl = String.format("%s/%s/stock?stock=%s",
            inventoryServiceUrl, order.getProductId(), order.getQuantity());
```

```
15          restTemplate.put(updateUrl, null);
16
17          // Continue with the remaining order placement logic
18      }
19  }
```

Listing C.9: *OrderManagement* microservice code - the service that uses data owned by Inventory microservice

The *OrderManagement* microservice is responsible for placing orders, and whenever it places an order, it makes a service call to the *Inventory* microservice API endpoint *'api/inventory/product/productId/stock'* to update the stock of the ordered product.

With this refactoring, the *Inventory* microservice has control over the inventory-related data, while the *OrderManagement* microservice interacts with the *Inventory* microservice through service calls to update the shared inventory columns. This way, each microservice focus on its specific responsibilities.

C.1.5 Create Data Transfer Object

Motivation

This refactoring is commonly necessary when we extract a service and there is a relationship between entities that will belong to different microservices. It suggests transferring more data in each call through a data transfer object that will hold all the call's data. Hence, this object will contain all the data that must be shared between the microservices to reduce the number of service calls performed, as service calls are expensive regarding latency.

It decouples presentation from the service layer and the domain model.

Note: This data transfer object must be serialisable to be sent through the connection.

Mechanics

1. Create an entity (Data Transfer Object) to hold the data necessary in a call between those services.

Example

In this example, we must create a Data Transfer Object to hold the information about an *Order* to be transferred between microservices during order-related communication. An example of the *Order* object being sent in the communication can be seen in Listing C.10.

```
1  @Entity
2  public class Order {
3      @Id
4      @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
5     private Long id;
6
7     private String customerName;
8
9     // Other fields and relationships
10
11    // Constructors, getters, and setters
12 }
13
14 @Service
15 public class OrderService {
16     private final OrderRepository orderRepository;
17
18     public OrderService(OrderRepository orderRepository) {
19         this.orderRepository = orderRepository;
20     }
21
22     public Order getOrderDetails(Long orderId) {
23         return orderRepository.findById(orderId);
24     }
25 }
```

Listing C.10: *Order* object is being sent through the communication channel.

In the *getOrderDetails* method from *Order* microservice class, an object of type Order is being sent through the communication. However, we want to create a Data Transfer Object that can hold the necessary data in a call to this method that contains more than information only present in the *Order* class. This way, the services will not have to share the same entity because we are encapsulating the specific data for communication, creating an abstraction. Listing C.11 shows the creation of a DTO for *Order*.

```
1  public class OrderDTO {
2      private Long orderId;
3      private String customerName;
4      private List<String> products;
5      // Other fields as needed
6
7      // Constructors, getters, and setters
8  }
9
10 @Service
11 public class OrderService {
12     private final OrderRepository orderRepository;
13
14     public OrderService(OrderRepository orderRepository) {
15         this.orderRepository = orderRepository;
16     }
```

```
17
18      public OrderDTO getOrderDetails(Long orderId) {
19          Order order = orderRepository.findById(orderId);
20          // Transform Order entity into OrderDTO
21          OrderDTO orderDTO = new OrderDTO();
22          orderDTO.setOrderId(order.getId());
23          orderDTO.setCustomerName(order.getCustomer().getName());
24          orderDTO.setProducts(order.getProducts().stream().map(Product::getName)
25              .collect(Collectors.toList()));
26          // Set other fields as needed
27
28          return orderDTO;
29      }
30  }
```

Listing C.11: Creation of a DTO to be sent through the communication

We defined a new class representing the DTO and declared the necessary fields to hold the data. In the future, more fields can be added to this DTO as they correspond to the transferred data. Then, we transform the data being transferred into the DTO. The *getOrderDetails* method in the *Order* microservice will no longer retrieve an *Order* object but a *OrderDTO* object.

The DTO provides a standard format for transferring the data of orders between services.

C.1.6 Break Data Type Dependency

Motivation

A data type dependency is common between different microservices. When we separate the microservices by business capabilities, some microservices might still need information about some entities in specific parts of their operations that are part of a different business capability. This dependency can appear on instance variables types, parameter types, return types and even method variables types.

We must identify where this data type is used and break this dependency to separate the microservices smoothly.

Mechanics

1. Identify where the data type is used (for example, the method invocations from the data type class, variable, parameters, or return types of the data type).
2. There are three ways of doing this:
   (a) Assuming it belongs only to the microservice where it was first defined:
      i. if there are method invocations:

A. Create an interface with the same name as the data type that defines the methods invocations identified for use through the data transfer object to make service calls to the data owner.

B. The method invocations will change from local calls to calls to the service that owns the data types and its methods, using the refactoring **"Change local method call dependency to a service call"** (C.1.1).

ii. The return types, variables and parameters shall use a Data Transfer Object to create the data type in the microservice because it will be sent through the service calls. Use the refactoring **“Create Data Transfer Object”** (C.1.5).

iii. Make the necessary changes in the code to use the new data type and the right interface for the method calls.

(b) Keep it in both microservices

i. Replicate the data type in both microservices and use event sourcing to ensure eventual consistency. Check the refactorings **"Change local method call dependency to a service call: asynchronous"** (C.1.1) and **"Replicate Data Across Microservices"** (C.1.3).

(c) Keep it in both microservices, but one of them is a proxy.

Example

We will focus this example on the first way of solving this, assuming it belongs only to the microservice where it was first defined.

Imagine we have two microservices *OrderManagement* microservice and *Inventory* microservice, where the *OrderManagement* microservice depends on a data type called *Product*. We want to break the dependency of the *Product* data type in the *OrderManagement* microservice and let the data type be owned by *Inventory* microservice. The *OrderManagement* microservice before the refactoring using the *Product* data type as a variable type, as can be seen in Listing C.12.

```
1  public class OrderService {
2      private ProductService productService;
3
4      public OrderService(ProductService productService) {
5          this.productService = productService;
6      }
7
8      public void createOrder(Order order) {
9          // Perform order creation logic
10
11         // Directly access the ProductService to get product information
12         Product product = productService.getProductById(order.getProductId());
13
14         // Use the product to complete the order creation process
```

```
15      }
16  }
```

Listing C.12: *OrderManagement* microservice before the refactoring

To resolve it, we create a *ProductInterface* that defines the necessary methods invocations to interact with *Product* data in the *Inventory* microservice. The *ProductService* implements this interface and makes the requests to the Inventory microservice that owns the data type *Product*. This way, We then replace the local method invocations in the *Order* service that involves the *Product* data type with calls to the *ProductService* interface, which will make service calls to the *Inventory* microservice to retrieve or manipulate the *Product* data.

We create a *ProductDTO* to use for transferring the *Product* data between the microservices through service calls, and we modify the return types, variables and parameters in the service's communications to use the DTO. Lastly, we update the *Order* service to use the new data type and the *ProductService* interface for method invocations. All changes performed to the *Inventory* microservice can be found in Listing C.13 and all changes performed to the *OrderManagement* microservice can be found in Listing C.14.

```
1  @Service
2  public class InventoryService {
3      public ProductDto getProductById(Long productId) {
4          // Logic to fetch product information from the inventory database or any
               other source
5          // ...
6
7          // Assume product information is retrieved and stored in the 'product'
               variable
8          ProductDto product = new ProductDto();
9          product.setId(productId);
10         product.setName("Example Product");
11         product.setPrice(BigDecimal.valueOf(9.99));
12
13         return product;
14     }
15 }
16
17 public class ProductDto {
18     private Long id;
19     private String name;
20     private BigDecimal price;
21
22     // Getters and setters
23 }
```

Listing C.13: *Inventory* microservice after the refactoring

```
public class ProductDto {
    private Long id;
    private String name;
    private BigDecimal price;

    // Getters and setters
}

public interface ProductInterface {
    ProductDto getProductById(Long productId);
}
@Service
public class ProductService implements ProductInterface {
    private final RestTemplate restTemplate; // or any HTTP client

    public ProductService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public ProductDto getProductById(Long productId) {
        // Make an HTTP request to the InventoryService to fetch the product
        String inventoryServiceUrl = "http://inventory-service/api/products/" +
            productId;
        ResponseEntity<ProductDto> responseEntity = restTemplate.getForEntity(
            inventoryServiceUrl, ProductDto.class);
        return responseEntity.getBody();
    }
}

@Service
public class OrderService {
    private final ProductService productService;

    public OrderService(ProductService productService) {
        this.productService = productService;
    }

    public void createOrder(OrderDto orderDto) {
        // Process the order details
        // ...

        // Retrieve product information from the ProductService
        Long productId = orderDto.getProductId();
        ProductDto product = productService.getProductById(productId);

        // Perform other order-related operations
        // ...
    }
}
```

Listing C.14: *OrderManagement* microservice after the refactoring

C.1.7 Duplicate file Across Microservices

Motivation

When extracting a microservice from the monolith, it is common to find the need to have an interface or an abstract class in different microservices.

Mechanics

In this case, duplicating the file for each microservice is okay, as these classes do not handle business logic.

Example

Imagine you have a file called *Utils.java* that defines multiple functions useful for this domain but does not handle any business logic. If the microservice *Order* and the microservice *Inventory* both use multiple functions from that file, we can just simply add the *Utils.java* file to both microservices repositories. Suppose the microservice *Order* and the microservice *Inventory* both use multiple functions from that file. In that case, we can just simply add the *Utils.java* file to both microservices repositories.

C.2 Infrastructure Improvement

When we migrate a monolithic application into a microservices architecture, we must make one significant change to get the full advantage of this new architecture: improving the infrastructure. This chapter introduces the changes to implement to the system's infrastructure to accommodate the microservices paradigm.

C.2.1 Introduce the circuit breaker

Motivation

Services make requests to each other to collaborate. Our system should be able to gracefully handle faults with an unknown recovery time, bringing resilience and stability to the application. For example, when a service makes a synchronous call to another service and is unavailable, it may block waiting for a response if nothing is prepared. Therefore, the failure of one service can potentially cascade to other services.

Adding a circuit breaker will provide a wrapper over remote invocations and a barrier that will shut off additional attempts to invoke a remote service based on a set threshold, allowing the system to recover from problems rather than propagate them throughout the system.

Mechanics

1. Add a Circuit Breaker to the system (either implement it or use a third-party library).
2. Configure and parametrise the Circuit Breaker's thresholds so that when the number of consecutive failures crosses them, the circuit breaker trips and during the timeout period, all attempts to invoke the remote service will fail immediately. After the timeout expires, the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes regular operation. Otherwise, if there is a failure, the timeout period begins again.
3. Configure the service client to invoke a remote service via the circuit breaker.

Note: There are various ways to implement this refactoring. These steps explain that it can be done in a separate service, within an API Gateway, on the client or server side. On the client's side, it stops making the calls after a threshold of failing calls. The services implement the circuit breaker logic on the server's side, so the calls still hit the service.

Example

The schema in Figure C.1 represents an example of the implementation of a circuit breaker.

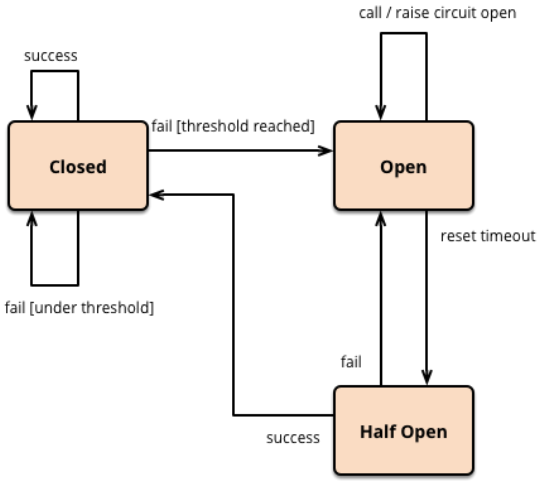


Figure C.1: Circuit Breaker implementation schema
From Fowler [66]

It starts in the close state. If the service is up and running, the circuit breaker passes the requests to services. However, when it fails to reach the service for a threshold of times, it enters the open state. In this state, the circuit breaker returns an error without processing the request. Additionally, it has a timeout setting that, once it is over, will make it enter the half-open state. In the half-open state, the circuit breaker tries once again to pass the calls to the service that was previously failing. If it fails again, it goes back to the open state. If it succeeds, it will return to the closed (default) state, where the calls are passed to the services.

C.2.2 Introduce service registry

Motivation

Each service can have one or more instances in production that can be modified dynamically. The introduction of a Service Registry enables services to locate one another dynamically.

Mechanics

1. Set up a service registry, which stores service instances' addresses.
2. Make each service register itself during the initiation.
3. You can unregister a service when they terminate or fail their health check.

Note: Implement this correctly, as failure to do so can create a single point of failure. Remember that the service registry directly impacts the system's availability; therefore, replication strategies should be implemented.

Example

A service requests the load balancer to communicate with a service. The load balancer queries the service registry for the existing instances of the requested service and load balance, choosing the endpoint of one of the less busy instances.

Figure C.2 demonstrates the communication between services using the load balancer and the service registry.

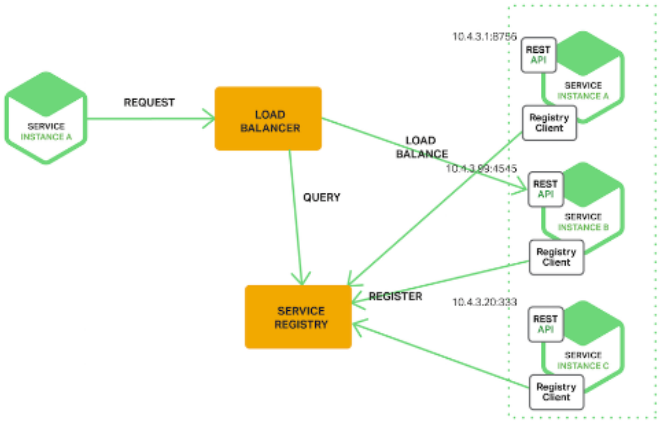


Figure C.2: Service Registry Implementation schema
Source: "Service Discovery in a Microservices Architecture", [17]

C.2.3 Introduce internal/external load balancer

Motivation

Assume you've already established a service registry. This service registry contains information about each operating microservice's many instances. It is feasible to balance the load on a service amongst its instances using this information. Its focus is to balance the load based on the client's requirements without needing an external load balancer.

Internal

Its focus is to balance the load based on the client's requirements without needing an external load balancer, allowing different load-balancing mechanisms to be used in different services.

Mechanics

1. Implement on each service an internal load balancer. This load balancer queries the service registry for the existing available instances for that service.
2. Make it balance the load between the available instances using the appropriate metrics for that service.

External

The goal is to balance the load with as few changes to the service code as possible and to create a centralised load-balancing approach for all services.

Mechanics

1. Set up an external load balancer that queries the service registry for the available instances of a given service and uses an algorithm to balance the load between the different services depending on the requests.
2. This load balancer can act as a proxy or an instance address (recommended).

Note: The load balancer can be built-in the service proxy, being the requests made to the service registry that return the address if the less busy instance of the requested service.

Example

The example provided in the previous refactoring (Section C.2.2) works perfectly for this case. It shows how a load balancer works in the systems infrastructure, whether internal or external. In the case of being internal, the requesting service would query the service registry directly for the existing instances and make the load balancer itself.

C.2.4 Introduce configuration server

Motivation

When breaking an architecture in microservices, each microservice might have numerous instances in production. Although, as mentioned in Section C.2.2, the list of available instances is accessible through the service registry, it may be necessary to update some configurations while they are operating without redeploying them. By using a configuration server, we can propagate the changes in the configuration to all running instances.

Mechanics

1. Create a separate repository for the software configurations.
2. Separate the source code and the software configurations into different repositories.
3. Every time a change happens to configuration keys, the configuration repository has to be synchronised.
4. When changes occur to the configuration repository, these should be propagated to the running instances that should adapt to the new configuration.

Note: If many programming languages are utilised, each programming language must implement the configuration propagation and adapting strategy.

Example

When a new configuration is committed to git, it updates the configuration server that propagates to the apps. The user then refreshes the apps. Figure C.3 shows an example of the application of this technique.

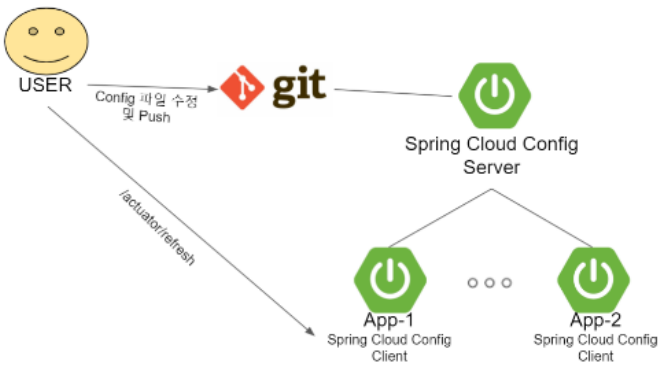


Figure C.3: Configuration Server implementation schema
Source: https://joomn11.tistory.com/79?category=936835

C.2.5 Introduce edge server or API Gateway

Motivation

With this new architecture, new services can be simply introduced, and existing ones can be rearchitected based on new requirements. On the other hand, the user does not need to be aware of this, and these actions should be tracked. Monitoring the status and usage of a service in production allows for the dynamic rerouting of external requests to internal services.

The API Gateway enables services to leverage several communication methods and form a single API with calls to many services.

Mechanics

Implementing an edge server or an API Gateway is quite similar. We create a new layer between the outside world and the set of microservices. This layer can be an API gateway or an edge server and perform dynamic routing based on its configuration. This layer can interact with the service registry to discover the addresses of the instances of each service. This layer becomes an excellent place to monitor the usage of services.

Note: As the communication is all performed through this layer, this layer should also be replicated through load balancing mechanisms to avoid becoming a single point of failure.

Example

Figure C.4 represents the layer of communication. The client requests to the API Gateway (or edge server), and this layer performs dynamic rerouting to the requested service to the most favourable instance.

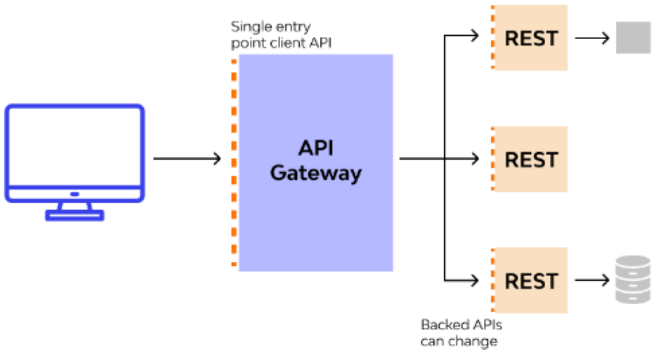


Figure C.4: API Gateway implementation schema
Source: "API Gateway", https://www.wallarm.com/what/the-concept-of-an-api-gateway

C.2.6 Configure service discovery

Motivation

It is common in a microservices architecture to use cloud solutions to host the systems; in these cases, the addresses where the instances are hosted are frequently dynamically allocated. Because the number of instances running changes dynamically, it is difficult to predict each service's address. It is usual in these scenarios to set up a service discovery with a service registry to keep track of the locations of existing systems.

Mechanics

1. Configure and run a service registry.
2. Whenever a service starts running, it registers itself on the service registry; whenever it shuts down, it de-registers itself from the service registry.
3. Add a service discovery client to the services to be used. Whenever they need to communicate with another service, they call service discovery to obtain the service location.
4. Replace previous configurations and remote calls to use the discovery client instead of a static address.

Note: It is common to have a replica of the service registry registered within itself so it does not become a single point of failure.

Example

Figure C.5 shows how the Service Discovery Pattern is implemented.

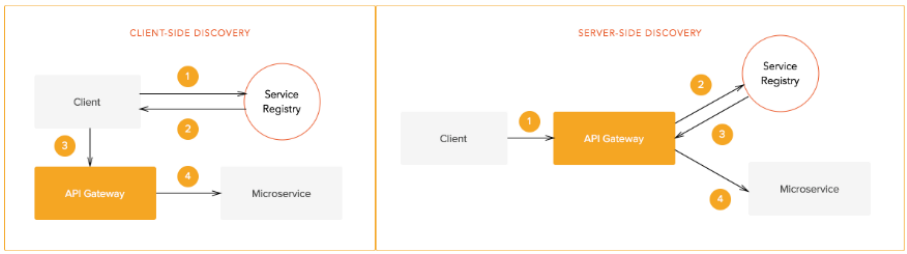


Figure C.5: Service Discovery implementation schema
Source: [86]

C.2.7 Configure health-check

Motivation

Microservices must be notified when a service fails to avoid routing requests to that microservice. This is often accomplished by sending a health check request to the microservice to determine

whether or not it is operational. It should either respond positively or return the specific error that is causing it to malfunction.

Mechanics

1. Decide what type of health check verification you want to perform, which can be a simple 200 status code response.
2. Implement the health check verification in the microservice. This can be verifying all the resources it needs, like the database, or simply returning a positive response.
3. Add the endpoint to the microservice.

Example

In Figure C.6, we can see the mechanics of implementing the health check. The load balancer periodically makes a health-check request to the services' instances it knows and waits for a positive response. If it does receive a positive response, it will take note of that and not forward the requests to that instance.

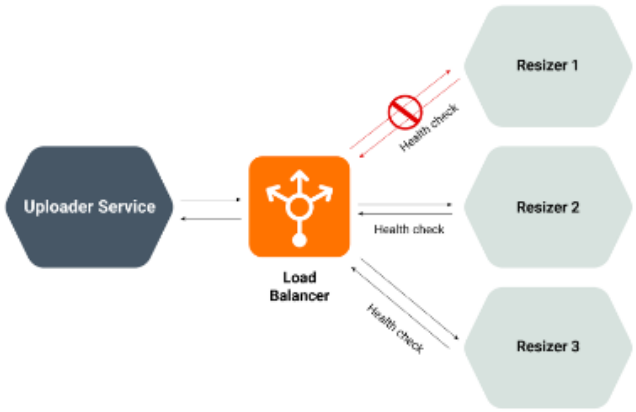


Figure C.6: Health Check implementation schema
Source: "In Brief: The Circuit Breaker Pattern", https://wso2.com/blogs/thesource/in-brief-the-circuit-breaker-pattern/

C.3 Deployment and Orchestration

Another major difference between monoliths and microservices is their deployment. As seen in the previous chapter, besides each microservice having to be deployed, each service usually has multiple running instances. Each of these instances has to be deployed, configured and monitored.

In addition, having multiple instances running simultaneously, orchestration becomes more demanding. This chapter approaches the strategies of deployment and orchestration to adapt to a microservices architecture.

C.3.1 Enable continuous integration

Motivation

The number of services expands due to the migration to microservices, yet we want the system to be "production-ready". We want to build and test the microservices automatically and verify their availability. Because we execute minor and frequent changes, continuous integration shortens the time it takes to validate and release software. It is a step to achieve continuous delivery/deployment.

Mechanics

1. Place each service in a separate repository, having its history and separating its build life cycle.
2. Create an artifact repository.
3. Create a continuous integration server.
4. Create a continuous integration job for each service, where the new code is fetched, all the tests are run in the new code, the corresponding artifacts are built, and these artifacts are pushed into the artifact repository.
5. Create a trigger so the job runs each time the code changes in that repository. The development team should be informed of the errors if the job fails.
6. When the job fails, resolving it becomes the priority, as new changes should pass all the tests so it does not break the existing system.

Example

Figure C.7 represents how continuous integration should work. Every time a developer commits to a git repository, this action should trigger the CI pipeline to run the job that builds the code, runs the unit tests and produces the result: either pass or fail and this result should be notified to the engineering team.

Figure C.8 shows how we connect continuous integration with continuous deployment. After merging the new code to the system and successfully passing all tests, we build and create an image, push it to the registry and after manual approval, it is deployed to production.

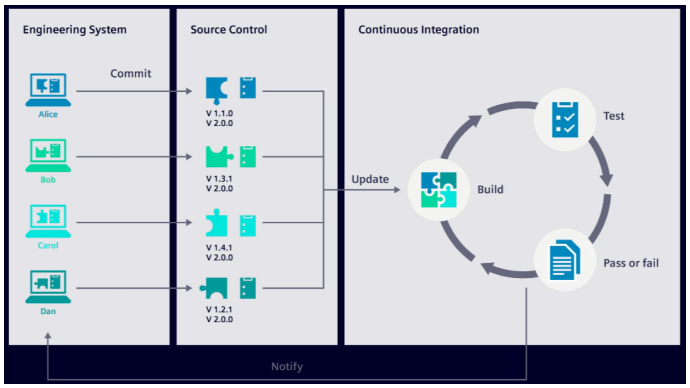


Figure C.7: Continuous Integration implementation schema
Source: "Continuous Integration with TIA Portal", https://www.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal/highlights/continuous-integration.html

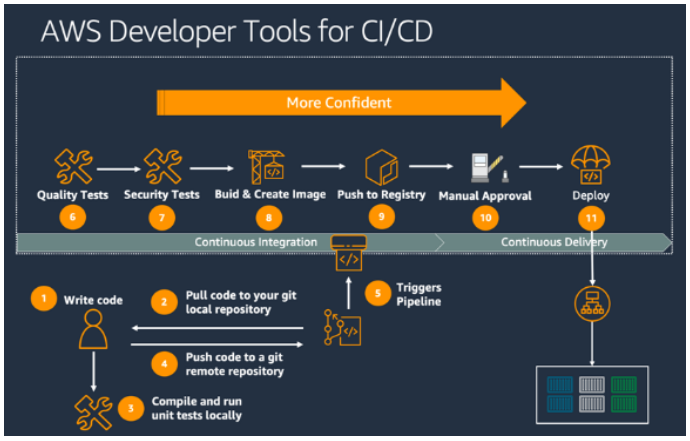


Figure C.8: Aws Developer Tools for CI/CD
Source: "AWS Prescriptive Guidance Patterns", https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/welcome.html

C.3.2 Containerize services

Motivation

One of the goals of breaking down a system into microservices is to make them totally independent of one another. This means that each microservice can be designed with a different technology, will have different configurations and different versions of the same packages, etc. and, as a result, each microservice will require a distinct environment for deployment.

Containerisation is a type of virtualisation in which a service is deployed into a single container image and runs isolated from the other services with its own desired environment. Besides this advantage, containers are lightweight and portable and facilitate automation.

Mechanics

1. Choose a containerisation technology. Docker is a widespread technology to use in these cases.
2. Analyse the system's dependencies.
3. In each service's code repository, create a script with the container image configuration and the script to make it run.
4. Have a list of the required environment variables for running the service in its code repository and the values it takes in a different environment (development, production, etc.). It is common to see this in a .env file.
5. Deploy the container.

Note: Containerising the services adds two more tasks for the continuous integration pipeline: building the container images and storing them in a private repository. Also, the development environment should adapt to embrace containers.

C.3.3 Orchestrate service

Motivation

The more microservices that are created, the more difficult it is to manage all of them and their dependencies since they all have different characteristics. Each microservice may require different initialisation strategies or specific configurations to run correctly. When availability, provisioning, scaling, and other configurations are added, it becomes overwhelming to accomplish this manually. Fortunately, orchestration tools/frameworks allow you to describe all of these behavioural preferences in a configuration file that establishes a distributed cluster with the appropriate nodes, builds the system, and operates it automatically.

Mechanics

1. Choose an orchestration tool/framework (e.g. docker-compose, Kubernetes).
2. Create a configuration file that describes how you want the system to work. This file is expected to be similar to the service's configuration file.
3. Start your system using the configuration files.

Example

Figure C.9 represents an example of container orchestration.

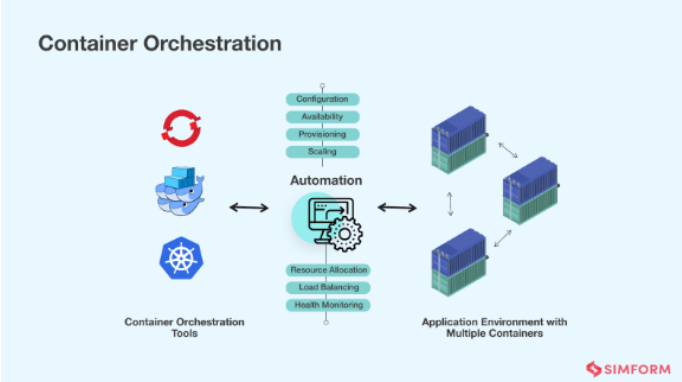


Figure C.9: Container Orchestration schema
Source: [17]

C.3.4 Deploy into a cluster and orchestrate containers

Motivation

We want to deploy all service instances into a cluster and orchestrate them with the least effort possible.

Mechanics

1. Choose a cluster management tool. Ideally, this tool should allow the definition of the deployment architecture of the services declaratively. This way, any additional effort for deployment, for instance, auto-scaling, monitoring, name resolution, and failure management of the deployed services, should be delegated to the cluster management tool.

2. Integrate this tool into your system.

Note: Pay close attention to single points of failure. Choose the management tool wisely, for example, one that provides means for auto-scaling the services.

Example

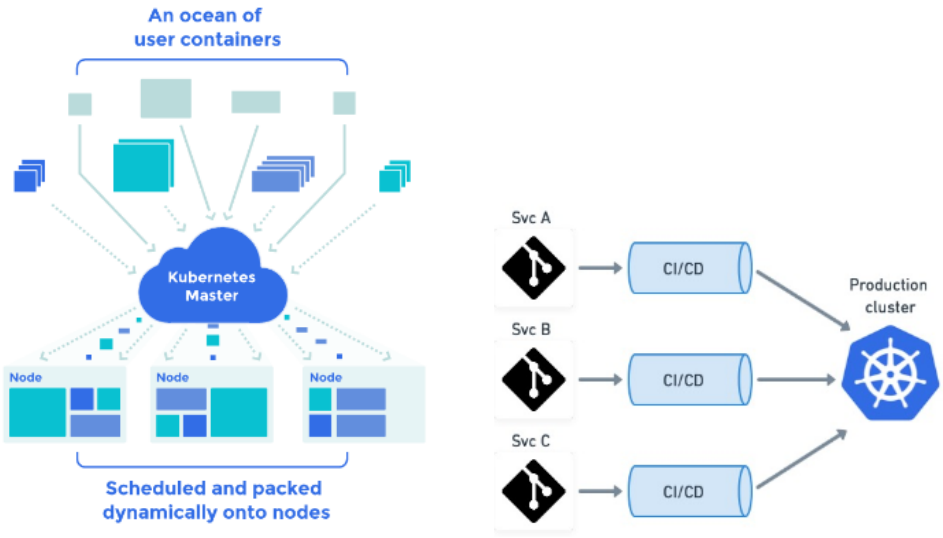
Figures C.10 show how it works to deploy services into clusters and its orchestration.

C.3.5 Centralize logging

Motivation

Each microservice can or not have its logging infrastructure. Logs aid in the diagnosis, troubleshooting, and tracking of issues. They usually get stored in the source directory in a log file.

If each microservice had its log file, the analysis of the execution of the entire system and debugging would be more challenging, as we couldn't easily track interaction between microservices or a global view of what is happening in the system.



(a) Kubernetes Master , Source: [28] (b) Production cluster, Source: [101]

Figure C.10: Deployment into clusters

This is simplified by having a centralised logging infrastructure to view the logs produced by each microservices. This centralised logging allows the addition of tags to the logs or even preferentially arrange them.

Mechanics

1. Configure the centralised logging platform.
2. Configure logging for your applications if it doesn't already exist.
3. Redirect the logs from the application to the centralised logging platform.

C.3.6 Centralize Configuration

Motivation

Similarly to the logs, each microservice may have its own set of configuration files. Because there are system-wide configurations, some of these configurations can be duplicated for all microservices. When one of these global configurations changes, all microservices must be updated, which is inefficient.

Extracting these configurations into a configuration server centralises the configurations, making modification easier.

Mechanics

1. Extract the information in the configuration files from each microservice into a source code repository to be versioned.

2. Configure and run a configuration server that reads the configuration from the repository and serves it to existing microservices.

3. Configure existing microservices to read the configuration from the configuration server instead of using the local configuration file.

C.4 Building Microservices

Following James Lewis and Martin Fowler's description of the common characteristics of microservices [62], we will dive into their definition of each characteristic and identify the refactorings offered in this catalogue that help to achieve them. As they said, "[...] not all microservice architectures have all the characteristics, but we expect that most microservice architectures exhibit most characteristics.".

C.4.1 Componentization via Services

The authors argue that we should treat services as components since they can be deployed independently. Thus, if a service changes, only that service needs to be redeployed, not the entire application.

"A service may consist of multiple processes that will always be developed and deployed together, such as an application process and a database that's only used by that service." [62]

Useful refactorings: All refactorings of chapters 1, 2, 3 and 6 contribute to having services as components.

C.4.2 Organized around Business Capabilities

There are numerous ways to break a system down into different services. However, the authors state that *"The microservice approach to division is different, splitting up into services organized around business capability. Such services take a broad-stack implementation of software for that business area, including user interface, persistent storage, and any external collaborations. Consequently, the teams are cross-functional, including the full range of skills required for the development: user experience, database, and project management"*. [62]

Useful Refactorings: This is not a refactoring issue but rather a concern when determining the boundaries of each service, which is not covered in this catalogue. With the company and project characteristics in mind, this decision should always be made to enhance project productivity and efficiency.

C.4.3 Products, not Projects

Following up the line of thought of organising the microservices around business capabilities, the authors believe that we should shift to the products over projects mentality. This way, you don't see the software development as completing a set of functionalities but rather as a way to make the

most of the business capability. With the typical small granularity of the services and each team owning a product over its lifetime, a closer connection is built with the user.

Useful refactorings: 3.1, 3.2., 3.3, and 3.4 are useful for letting a team own a product, and 6.1 to create these products teams will be working on.

C.4.4 Smart endpoints and dumb pipes

This trait is related to the shift in communication paradigms that occurs when we migrate to microservices. This structure must be carefully considered since calls between microservices can cause increased latency.

The basis of communication in microservices is to make a request, which is then processed and applied some logic by the service receiving the request, which produces a response.

According to the authors, communication is most typically implemented via HTTP request-response protocols with resource APIs or lightweight messaging. The first makes use of web principles and protocols. The second, a lightweight message bus, often only works as a message router ("dumb") and leaves the "smart" part to the endpoints. These services will design a strategy to operate without introducing additional latency, which is common in service calls.

Useful refactorings: 1.1., 1.5, 6.4, 6.5., and 6.7 are useful refactorings to create smart endpoints as well as the concerns mentioned in Chapter 7.

C.4.5 Decentralized Governance

As we split the monolith into microservices, each microservice can have its own stack of technologies, standards, and so on. It also opens the door to employing tools to handle the demands of each microservice, as well as using and developing open-source code that other developers can use to solve similar challenges. Decentralised governance allows each service to have its own governance strategy, contributing to the reality that each team should own their product.

Useful refactorings: 2.1 to 2.7, 3.1, 3.2, 3.3 and 3.4.

C.4.6 Decentralized Data Management

Microservices decentralise both conceptual models and data storage decisions.

"Microservices prefer letting each service manage its own database, either different instances of the same database technology or entirely different database systems". [62]

This creates implications for managing updates, it can be solved by using transactions (even distributing transactions, when needed), but usually, it is mitigated by assuming eventual consistency to answer the demand quickly and through compensating operations to deal with mistakes.

Useful Refactorings: 1.2, 1.3, 1.4, 1.6, 1.7, 6.4, 6.5, 6.6, 6.7 and 6.8.

C.4.7 Infrastructure Automation

Infrastructure automation solutions have grown substantially with the emergence of the cloud. Systems designed with microservices use Continuous Integration and Continuous Delivery strategies, which benefit greatly from infrastructure automation techniques.

The most popular applications of infrastructure automation are automated testing, deployments, and microservices management in production.

Useful refactorings: 2.1 to 2.7 and 3.1 to 3.6.

C.4.8 Design for failure

"A consequence of using services as components is that applications need to be designed to tolerate the failure of services. Any service call could fail due to the unavailability of the supplier, and the client has to respond to this as gracefully as possible". [62]

Services fail frequently, and we must be able to detect and automatically restore them. This can be accomplished through real-time monitoring of the application and logging configurations for each service. How we achieve this is irrelevant, but maintaining the system available is critical.

Useful refactorings: 2.1, 2.2, 2.3, 2.6, 2.7, 3.3, 3.4, 3.5, 6.5, 6.7, 6.8, and all concerns mentioned in chapter 7.

C.4.9 Evolutionary Design

When decomposing a monolith into microservices, we must keep the concepts of independent replacement and upgradeability in mind.

Code that changes together should be in the same microservice, and code that changes rarely should be in a distinct microservice than code that changes regularly. If two services tend to change simultaneously, they should generally be merged.

It is also useful to use microservices when we know that something will be temporary; in this manner, we build and deploy it fast, and when it is no longer necessary, we remove it, speeding up the release process. However, this raises the concern of one service breaking its consumers, so the services should be designed to be as tolerant as possible of the changes that their suppliers may suffer.

Useful refactorings: All refactorings of chapters 2 and 3 contribute to an evolutionary design and refactorings 6.1, 6.2, 6.4, 6.5, 6.7, 6.8, 6.9 and the concerns mentioned in chapter 7.

C.5 Intermediate refactoring states (common strategies)

The migration should be a step-by-step process as it is complex. Therefore, multiple ways order these steps, leading to a microservices architecture. In this chapter, we present some common refactorings that lead to intermediate states of the refactoring that contribute to the end goal of migrating to a microservices architecture but do not directly apply refactorings that go along with microservices characteristics. For instance, not always database splitting is the first step, and there

are numerous states the database can be refactored as intermediate steps to allow the evolution of the migration.

C.5.1 Shared Database

Motivation

When we start breaking the monolith, we may not want to split the database straight away to the different microservices. We may want to avoid performing all these changes in the first moment and focus on the functionality migration. This state is usually an initial state, usually a state of the system's evolution, but it does not represent a refactoring. Situations where using a shared database might be a good solution are:

1. In a microservices architecture is on read-only static reference data because there aren't multiple services trying to change the same data, only reading it;
2. When a service exposes its database as a defined endpoint that was thought to handle multiple consumers;
3. Or when it is an intermediate step before refactoring it.

Mechanics

The database is shared among multiple microservices. Each service can access data owned by other services using local ACID transactions. If, at some point, two microservices share a purely read-only database part, it may not be necessary to keep copies of these in multiple places, and it may be kept that way.

Note: This is not ideal in the microservices paradigm as it brings inconsistencies, no possibility of keeping things hidden, etc., so splitting the database is preferred.

Example

In Figure C.11, we can see that all three services, Dispatch, Finance and Order Processing, use the same database and can change its information directly.

C.5.2 Database Wrapping Service

Motivation

When multiple services depend on the same datastore, it is hard to consider pulling apart the underlying schema. The database wrapping service uses a service as a wrapper of the database schema, hiding the schema complexity behind a service. This refactoring is seen as a step toward more fundamental changes. It is common in a many-to-many relationship between entities in different microservices, where there usually is a join table. Instead of using the move foreign-key

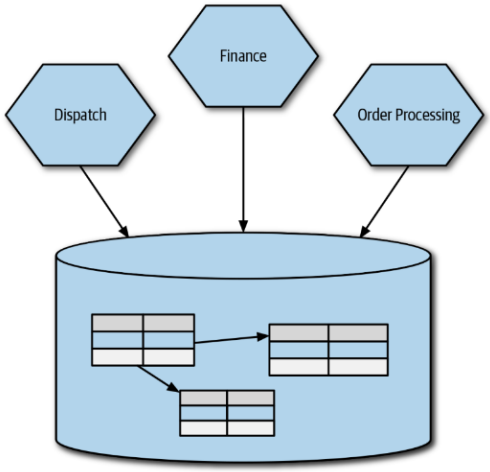


Figure C.11: Shared Database Example
Source: From S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly, 2019. [75]

connection to code refactor and lose the table, this refactor is applied, and the relationship is kept the way it is.

Mechanics

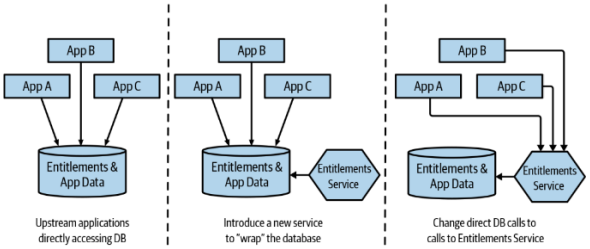
1. Create a new service to hide the database that acts as a thin wrapper.
2. Extract the classes mapped to the involved entities to the new service.
3. Replace clients' requests to the database with requests for this new service.

Note: Encourage developers writing the different clients to think of this new service as someone else's and start storing their data locally. Ensure that the database schema is unchanged and that there is low coupling.

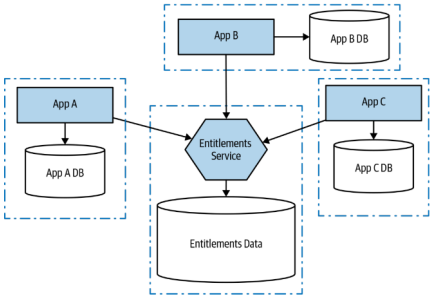
Example

The new service will act as an interface to consumers while changes are being made in the database to improve the schema. This way, it can later adapt to our end goal of breaking the tables, so each service owns its data, and there is loose coupling.

In this example, Figure C.12, all apps must access the database. This database stores each app's data and entitlements. The entitlements are the accounts each individual can access and the operations they can do. As the entitlements are very complex and have a lot of logic behind them, we create the Entitlements Service that owns the entitlements and app data and hides this complexity. Then we create a database for each service, so each service stores its app data, and the entitlement service only stores the entitlements data in its database. This way, other services think of the entitlements as owned by another service.



(a) Using a service to wrap a database



(b) Using the database wrapping service pattern to reduce dependence on a central database

Figure C.12: Database Wrapping Service Example
Source: From S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly, 2019. [75]

C.5.3 Database-as-a-Service

Motivation

When clients need a database to query, this could be because they need to query large amounts of data or because external parties already use toolchains that require a SQL endpoint to work against.

Mechanics

1. Create a dedicated database to be exposed as a read-only endpoint and have this database populated when the data in the underlying database changes.
2. Separate the database we expose from the one we use inside our service boundary.
3. In the external database, make a mapping engine that takes changes in the internal database and determines what changes.
4. The mapping engine must change when the internal database changes structure to ensure the public-facing database remains consistent.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.4 Database view

Motivation

We want a single data source for multiple services, but changing all clients to point to the new service(s) is impractical. For clients considering the monolith as their data source, we want to

mitigate concerns regarding coupling to specific data schema parts. There are more clients reading data than writing it. We can use database views to show only the relevant data for each service.

Mechanics

1. Create a dedicated schema that hosts views like the old schema assembled from data now owned by the new service(s).
2. Have clients that only read data points at those views instead of the original tables.
3. Change the clients that need to write data to use the new services directly instead of the monolith.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.5 Synchronize data in the application

Motivation

When we want to split the schema of the monolith into two separate data stores in preparation for extracting a new service.

Mechanics

1. Create the new data store and bulk-import data from the monolith—only the part of the schema we are interested in splitting needs to be imported.
2. Make the monolith write the same data to both data stores, ensuring they are kept in sync, and that writes to the new data store work well.
3. Make the new data store the source of truth and ensure that the reads also work. As we still write to both data stores, we can easily fall back to reading from the old data store if any issue is found.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.6 Tracer writer

Motivation

When we need to move the ownership of some data from the monolith to a new service in an incremental fashion, tolerating there being two sources of truth during the migration.

Mechanics

1. Identify the part of the monolith data schema whose ownership we want to change and the service that will host it.
2. Ensure this data syncs between the monolith and the new service. This can be achieved in several ways:
   (a) Allowing writing only on one of the sources of truth and making it in charge of replicating the change in the other one;
   (b) Having clients send writes to both sources; or
   (c) Allow clients to send writes to either source and synchronise data behind the scenes.
3. Once all clients use the new source of truth, the old source of truth can be retired.

Note: *Synchronize Data in Application* focuses on splitting the schema, and this technique focuses on moving a part of the schema to the ownership of a previously created service.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.7 Separating libraries from their dependents

Motivation

Martin Fowler defines libraries as "components that are linked into a program and called using in-memory function calls", distinguishing them from services that are "out-of-process components who communicate with a mechanism such as a web service request or remote procedure call". In this situation, there is code used by multiple services using a method call, but it is not business related.

Scalability needs between a shared library and the dependent services can create the need for the library to become a service that can scale independently.

Mechanics

1. Extract the code used by multiple services into a library.
2. Extract the library as a service creating an API or other types of interface to access it.

Note: Sometimes, separating libraries from their dependents and using a service call instead of a method call may introduce performance issues. Although performance issues can be handled using careful caching mechanisms, it adds another layer of complexity to the dependent service.

C.5.8 UI Composition

Motivation

As we incrementally migrate a monolith to microservices, the user interface's functionality must be served partly by an existing monolith and partly by the new microservice architecture.

Mechanics

Different variants of UI Composition may be used:

1. Page-based Composition consists of having different services serve different pages, migrating each one at a time.
2. Widget Composition consists of embedding a piece of the user interface provided by a new service into the monolith-provided user interface.
3. Micro Frontends consist of taking a microservice-based approach to frontend development. The idea is to break down a user interface into different components that can work independently.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.9 Branch by Abstraction

Motivation

Changes to the existing codebase will likely take time, and we want to avoid any disruption. It assumes we can change the code of the current system.

Mechanics

1. Develop, in the monolith, an alternative implementation for a module so that the new and old implementations comply with the same interface/abstraction.
2. The new implementation typically calls a new external service, whereas the old implements the functionality internally.
3. At some point, switch from the old to the new one.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.10 Parallel Run

Motivation

When the failure of the functionality being worked implies a high risk for the business.

Mechanics

1. With a parallel run, rather than calling either the old or the new implementation, we call both, allowing us to compare the results to ensure they are equivalent.
2. Despite calling both implementations, only one is considered the source of truth at any given time. Typically, the old implementation is considered the source of truth until the ongoing verification reveals that we can trust the new implementation.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.11 Decorating Collaborator

Motivation

We use it to trigger some behaviour based on something happening inside the monolith, but we want to avoid changing it. This technique assumes we can intercept responses returned by the monolith before they reach the clients.

Use the decorator pattern to make it appear from the outside that we have added new logic to the monolith without actually changing it.

Mechanics

1. Route client requests to a proxy, allowing the call to reach the monolith as usual.
2. Based on the result of this call, call out to a new microservice and possibly modify the result before returning it to the client.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.12 Change Data Capture

Motivation

When we need to react to a change in data in the monolith but cannot intercept this change at the perimeter of the monolith (e.g., using a decorator) or change its implementation.

Mechanics

Rather than trying to intercept and act on calls made into the monolith, we react to changes made in a data store. The underlying capture system will be coupled to the monolith's datastore. It can rely on database triggers, transaction log pollers (usually a file into which is written a record of all the changes that have been made) or batch delta copier (a program that regularly scans the database in question for what data has changed).

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.5.13 Aggregate Exposing the Monolith

Motivation

When the monolith database still owns the data we want to access, we need a way for our new service to access it. Calling back to the monolith to access the data is harder than accessing the database directly; however, it is better in the long run. So we can expose a service endpoint (an API or a stream of events) in the monolith to access this data.

Mechanics

1. Create an endpoint that exposes the data the microservice needs to access.
2. This endpoint can limit the level of access the microservice can have to that data.

3. Make the microservice use this endpoint to access the data.

Example

S. Newman's book, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith* [75], presents a good example of application of this technique.

C.6 Common Sequences of Refactorings

This chapter introduces common sequences of refactorings found when migrating from monoliths to microservices.

C.6.1 Extract Service

Motivation

We may need to extract a service from the rest for multiple reasons, like scalability, ease of deployment, etc. We have multiple functionalities when we have a monolith system, but not all are used as much as the others. As one functionality is used more frequently than the other, scalability can become an issue. This refactoring will transform one or more regular class(es) into a remote service.

Mechanics

1. Analyse all the dependencies of the functionality we will extract to a service. Both the dependencies to the rest of the monolith and that the monolith has with the service to be extracted.

2. Resolve these dependencies. This usually involves refactoring foreign keys and changing method calls to other files/classes/etc. to remote calls (synchronous or asynchronous). Appendix C.1 describes interesting refactorings to resolve these dependencies.

   (a) If external libraries are dependencies, they will later need to be added to the new service.

   (b) All its methods used by other components inside the monolith had to be ready to be called over service calls, like services communication, etc.

   (c) Its original clients have to access it through remote service calls.

3. Create a new project folder with the files identified to be owned by this functionality with the same characteristics and the changes made during the previous steps.

4. Make it capable of running independently, installing the necessary dependencies and preparing its production environment.

Note: Make sure to guarantee fault tolerance and data consistency and to solve performance issues (check **Chapter 7 (Appendix C.7)**).

Example

We have seen in the previous section (cf. Section C.1) many refactorings that lead to the extraction of the services *Inventory* and *Order*. To apply this refactoring, the same way we identified those dependencies, we identify all dependencies these microservices have between themselves and to the rest of the monolith and use similar refactorings to the ones identified in that section to extract them. When all dependencies are resolved, we create a new project for each of these microservices with the designated files that belong to them; we build them and deploy them after correctly preparing them with the necessary dependencies for the production environment.

C.6.2 Strangler Fig

Motivation

When we have decided to evolve a system to a microservices architecture, we want to take incremental steps toward the new architecture and ensure that each step is easily reversible, reducing risks.

Mechanics

1. You can start by deciding if you want to wrap the monolith with an API that allows us to access the new system in the old way. If so, perform the necessary implementations (Branch by Abstraction refactoring, **Chapter 5 (Appendix C.5)**).
2. Start small, with the macro, then micro mindset and identify the functionalities you want to extract and their boundaries.
3. Identify the order by which you want to extract the functionalities. And program how you want to do the extraction.
4. Gradually move functionality over to the new microservices architecture. This technique usually uses as the main refactoring of the extract service refactoring (C.6.1).
5. Reroute calls from the monolith over to the new microservice using the change local method call dependency to a service call (C.1.1).
6. If the new extracted functionality uses functionalities that remain inside the monolith, then the monolith can expose this functionality (C.5.13). Iteratively extract all functionalities.
7. Write new code as microservices.

   Note: Do this by replacing or rewriting existing features parallel to the old architecture, one at a time, until the old architecture has been entirely replaced. Usually, we must create a proxy or façade that provides a stable API for old clients throughout the migration.

Example

As an example, we are going to use the same example given by Sam Newman on *"Monolith to Microservices: Evolutionary Patterns to Transform your Monolith"* [75].

In Figure C.13, we can see that the *InventoryManagement* functionality is self-contained and, therefore, has no dependencies. So, we can simply extract this service, rerouting existing calls regarding this functionality to this new service instead of to the monolith.

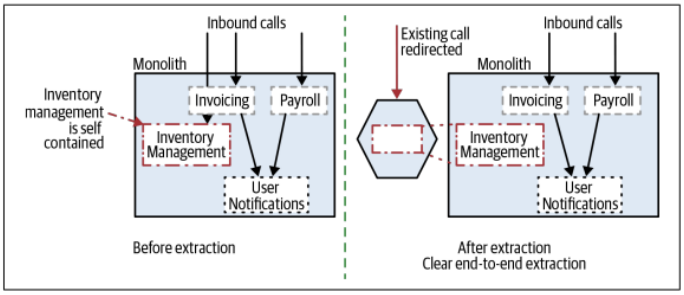


Figure C.13: Example of application of the Strangler Fig refactoring to the InventoryManagement refactoring
Source: From S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly, 2019. [75]

We gradually plan the extraction of the functionalities to services according to an order we define and at our own pace, and all the new code is written as microservices. These changes can be rolled back whenever we find it is not suitable. If the functionality were to be used by the functionalities inside the monolith, we would have to redirect those too.

C.6.3 Typical functionality library pattern

Motivation

When many services have functionalities in common that do not have memory or state.

Mechanics

1. Move the common functionalities into a repository with common code libraries.
2. Import this library into the services that used the functionalities.
3. Change the service calls for these functionalities to code dependencies.

C.6.4 API Composition

Motivation

Each microservice has its database, and it is no longer straightforward to implement queries that join data from multiple services. Creating an aggregator service allows one to gather data from the different services, merge it and show it to the end user.

It is very common to use this refactoring when there are a lot of reads.

Mechanics

1. Define an API Composer.
2. Make the composer identify the services it needs to invoke to answer the user query.
3. After identifying, it queries the microservices.
4. Perform an in-memory join of the results.
5. Return the response to the user.

C.6.5 SAGA

Motivation

Each service has its own database, but some business transactions involve multiple services. This refactoring corresponds to a distributed transaction and increases data consistency across services. This represents another case where we expect eventual consistency because each transaction may take some time. It is common when multiple writes to the database are triggered by one action.

Mechanics

1. Implement each business transaction as a SAGA - sequence of local transactions.
2. Each transaction updates the database and publishes a message or event to trigger the next local transaction. Or an orchestrator tells the participants what local transactions to execute.
3. If a local transaction fails because some business rule was violated, then a series of transactions are executed to undo the changes made by the previous transactions.

Example

Figure C.14 shows an example of the SAGA works.

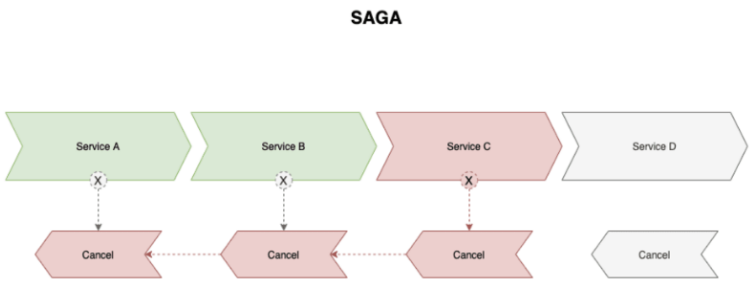


Figure C.14: SAGA example
Source: [44]

C.6.6 Change Data Ownership

Motivation

When extracting a new service that encapsulates the business logic of some data, that data should belong to that service and, therefore, should be moved into the new service. The new service has to be the new source of truth of that data.

Mechanics

To do this, we have to break the dependencies the monolith may have with this data through refactorings like: C.1.2, C.1.3, C.1.4, C.5.2, and C.6.5.

Example

As an example, we are going to use the same example given by Sam Newman on *"Monolith to Microservices: Evolutionary Patterns to Transform your Monolith"* [75].

In Figure C.15, we can see that we want to move invoice-related data into the new *Invoice* service. Therefore, we need to change the monolith to take the *Invoice* service as the source of truth for invoice-related data and change all calls to read or change the invoice-related data to be made directly to the *Invoice Service*. This can initiate other refactorings like "Move Foreign-key relationship to code" to make this work.

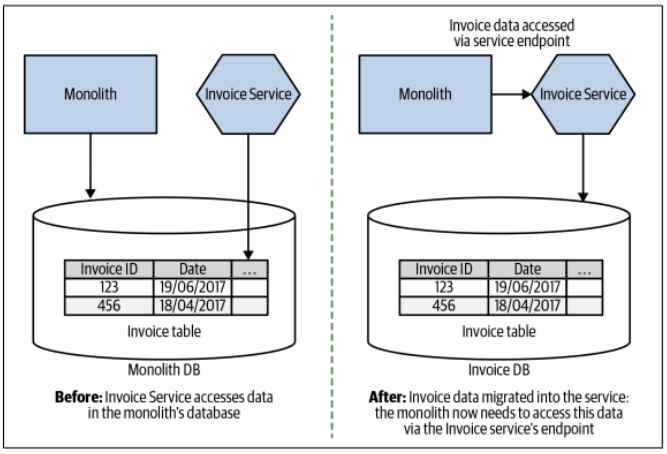


Figure C.15: Example of application of the Change Data Ownership refactoring of the Invoice table to the Invoice Service
Source: From S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly, 2019. [75]

C.6.7 Tolerant Reader

Motivation

Being tolerant when reading data from a service is an excellent advantage because it prevents issues from happening whenever the format changes. The idea is to read only the parts of the message it needs to read and tolerate changes as long as the required fields are present and readable. Even if some fields are added or removed, the ones the reader needs are expected to remain.

Mechanics

1. Take the minimum assumptions about the structure of the message.
2. Make sure there's only one bit of code that reads data payloads like this.
3. For example, if you use a DTO, use generic collections to make it serialisable, allowing the technologies to differ from service to service.

C.6.8 Anti-corruption Layer

Motivation

We want to avoid data corruption during any communication at all costs. Therefore, creating an anti-corruption layer is often a good solution when the services use different communication technologies.

Mechanics

1. Create a layer between the two services as a proxy to allow them to communicate.
2. This layer should verify the data flowing in the communication and transform it to fit the receiver, acting like a translator if needed.

Example

Figure C.16 shows how typically an Anti-corruption layer is added to the system.

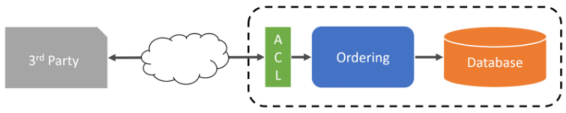


Figure C.16: Anti-corruption Layer schema
Source: [23]

C.6.9 Adapt Service Interface

Motivation

The availability of business functionality across different providers is quite typical in cloud-based services. Multiple service implementations may be required for various business and technical reasons, including fault tolerance and a better division of responsibilities (as less similar code is repeated).

The issue is that it is also common for this multiple services implementation to have different service interfaces. When this happens, there are two ways of solving it: you either treat the invocation of the similar services functionalities independently, and each service has a replica of that code, or you can systematically adapt the services' interface to each other, so we can use the same interface no matter which service we invoke.

Mechanics

These mechanics explain how to implement the second methodology to solve this issue.

1. Create an adapter service that translates the requested method to the method on the adaptee interface.
2. Incrementally start with two services interface, identify the differences and map the names to create one interface.
3. Solve the differences by reordering the parameters, providing missing parameters or omitting extra parameters. In some cases where different types for the same parameter must be solved manually, adjusting to each one seems better.
4. It ends when you can create one single interface for all services.

C.7 Extra Concerns

In most circumstances, we should have additional considerations in addition to the procedures indicated in each refactoring to design a resilient system. Numerous inputs exist on these topics, so we leave links to helpful resources to address these concerns.

More concerns can be added in this chapter, as the change in architectural paradigm raises many new concerns to keep in mind. Two sources we want to add to this catalogue because it comprises a lot of information on microservices architecture are the following Source 1, Source 2.

C.7.1 Data consistency

Ensures the data format or the data concerning other data. It basically refers to the usability of the data.

This is ensured in a monolith by the ACID transactions of the relational database. ACID means:

- Atomicity;
- Consistency;
- Isolation;
- Durability.

Even though, in a microservices architecture, the data is distributed across the microservices, it still needs the ACID properties.

To achieve this, three common strategies are distributed transaction (check SAGA refactoring, C.6.5), Two-Phase commit and Eventual Consistency.

For instance, in cases of moving the foreign key relationship to code, we can check before deletion, handle deletion gracefully or don't allow deletion when the data the foreign key refers to is being deleted so we can ensure the data consistency on the service that uses that data through the foreign-key.

Some useful sources to look for information about this are: Source 1, Source 2, Source 3, Source 4, Source 5, Source 6 and Source 7.

C.7.2 Resilience and Fault Tolerance

Refers to the system's ability to continue operating without interruption when one or more components fail.

Some good strategies to create a resilient and fault-tolerant system are timeouts, retries, circuit breaker, chaos testing, etc.

Some valuable sources to look for information about this are: Source 1, Source 2, Source 3, Source 4, Source 5 and Source 6.

C.7.3 Performance

Performance is a common bottleneck for microservices. In this section, we want to provide some sources to identify and solve the performance challenges.

Some recommended sources are: Source 1, Source 2, Source 3 and Source 4.

It is also common to apply the CQRS pattern as it allows to maximize performance, scalability and security. Check this Source for more about it.

C.7.4 Security and Network

In distributed systems, network and security problems are very likely to rise. The following sources provide good insights on this theme: Source 1, Source 2, Source 3, Source 4, Source 5, Source 6 and Source 7.

C.8 Migration Notes

Preparing well is the best way to successfully journey from a monolithic architecture into a microservices architecture. This means:

- Having a good understanding of the monolith and of the reasons why you are performing the migration;
- Give training to the teams that will handle the microservices;
- Creating a good plan for the migration and how you will handle the synchronisation of the monolith and the microservices;
- Using feature flags is a good technique to turn on and off portions of the monolith when they are migrated;
- Have a good set of automated tests implemented, from unit to integration and end-to-end;
- Many times, putting everything in a mono repo and creating a shared CI pipeline is recommended;
- Sometimes, modularising the monolith before the migration is a good strategy;
- Creating a monitoring server that monitors, parses the information and aggregates it into structured information that can be pooled timely to update some visualisation tool to help each service team improve performance and detect anomalies.
- Going Macro and then Micro. Start with larger services around a logical domain concept, then break them into multiple services.

A piece of advice is to start small and analyse well if the partitions being performed are productive. Picking something with the fewer dependencies possible, for instance, can be a good starting point. The strangler fig refactoring is one of the most used approaches.

C.9 References

Table C.1 shows the refactorings in the catalogue and the main references we used.

Table C.1: Refactorings references of the complete catalogue

Refactoring Name	**Chapter**	**References**
Change local method call dependency to a service call		[9], [112], [79]
Move Foreign-key relationship to code		[37], [75], [79]
Replicate Data Across Microservices		[35]

Continued on next page

Table C.1 – continued from previous page

Refactoring Name	Chapter	References
Split Database Across Microservices	1	[112], [75]
Create Data Transfer Object		[37]
Break data type dependency		[37]
Duplicate file Across Microservices		-
Introduce the circuit breaker		[112], [79], [82]
Introduce service registry		[9]
Introduce internalexternal load balancer		[9]
Introduce configuration server	2	[9]
Introduce edge server or API Gateway		[9]
Configure service discovery		[79]
Configure health-check		[79]
Enable continuous integration		[9]
Containerize services		[79]
Orchestrate service		[79]
Deploy into a cluster and orchestrate containers		[9]
Centralize logging	3	[79]
Centralize Configuration		[79]
Shared Database		[75]
Database Wrapping Service		[75]
Database-as-a-Service		[75]
Database view		[75]
Synchronize data in the application		[75]
Tracer writer		[75]
Separating libraries from their dependents	5	[75, 9]
UI Composition		[75]
Branch by Abstraction		[75]
Parallel Run		[75]
Decorating Collaborator		[75]
Change Data Capture		[75]
Aggregate Exposing the Monolith		[75]
Extract Service		[108], [79]
Strangler Fig		[75], [108]
Typical functionality library pattern		[112]
API Composition		[68]
SAGA		[68]
Change Data Ownership		[75]
Tolerant Reader	6	[112], [32]

Continued on next page

Table C.1 – continued from previous page

Refactoring Name	**Chapter**	**References**
Anti-corruption Layer		[112], [58], [23]
Adapt Service Interface		[61], [59], [100]

Appendix D

MicroOnion: an Assisted and Incremental Refactoring Tool

D.1 Source code representation example

Listing D.1 presents an example of the JSON file with the source code representation's content.

```
{
    "pl.edu.wat.wcy.pz.restaurantServer.entity.User": {
        "full_name": "pl.edu.wat.wcy.pz.restaurantServer.entity.User",
        "constructor": [],
        "package": "pl.edu.wat.wcy.pz.restaurantServer.entity",
        "short_name": "User",
        "annotations": [
            "@AllArgsConstructor",
            "@NoArgsConstructor",
            "@Entity",
            "@Builder"
        ],
        "instance_variables": [
            {
                "annotations": [
                    "@Id",
                    "@GeneratedValue(strategy = GenerationType.IDENTITY)",
                    "@Column(name = \"USER_ID\")"
                ],
                "modifier": "private ",
                "identifier": [],
                "type": "Long",
                "variable": "userId",
            },
            ...
        ],
        "methods": {
            "setPassword": {
                "name": "setPassword",
                "annotations": [],
                "returnDataType": [
```

```
                    "void"
                ],
                "identifier": [],
                "parametersDataType": [
                    {
                        "type": "String",
                        "variable": "password",
                        "qualifedType": ""
                    }
                ],
                "variables": [],
                "body": "{\n    this.password = password;\n}",
                "methodInvocations": []
            },
            ...
        },
        "implements": [],
        "extends": [],
        "imports": [
            "lombok.AllArgsConstructor",
            "lombok.Builder",
            ...
        ],
        "methodsInvocations": [],
        "isInterface": false,
    },
    ...
}
```

Listing D.1: Source Code Representation Example

D.2 Output of Brito et al. tool example

Listing D.2 represents the resulting file of the Brito et al. [13] tool for the RestaurantServer project.

```
[
    {
        "id": "12_02_15_08_13",
        "name": "asledziewski__restaurantServer",
        "rootPath": "/home/mbrito/git/thesis-web-applications/monoliths",
        "relativePath": "asledziewski__restaurantServer",
        "clusterString": "{0: ['pl.edu.wat.wcy.pz.restaurantServer.security.
            WebSecurityConfiguration', 'pl.edu.wat.wcy.pz.restaurantServer.security.jwt.
            JwtAuthEntryPoint', 'pl.edu.wat.wcy.pz.restaurantServer.security.jwt.
            JwtAuthTokenFilter', 'pl.edu.wat.wcy.pz.restaurantServer.security.jwt.JwtProvider
            '],
        ...
        6: ['pl.edu.wat.wcy.pz.restaurantServer.form.response.JwtResponse', 'pl.edu.wat.wcy.
            pz.restaurantServer.email.MailService', 'pl.edu.wat.wcy.pz.restaurantServer.
            controller.AuthController', 'pl.edu.wat.wcy.pz.restaurantServer.entity.Role', 'pl
            .edu.wat.wcy.pz.restaurantServer.repository.RoleRepository', 'pl.edu.wat.wcy.pz.
            restaurantServer.form.LoginForm', 'pl.edu.wat.wcy.pz.restaurantServer.form.
            SignUpForm']}",
```

```
        "commitHash": "",
        "Modularity": 0.6102473225290658
    }
]
```

Listing D.2: Output of Brito et al. tool for RestaurantServer project

D.3 File representing the system's evolution

This example represents part of the file representing the system's evolution that is of the type *RefactoringRepresentation* from the domain model. In this example, Listing D.3, we can see that service 1 has already been extracted, as it is independent as well as the changes that occurred in its representation to implement the refactorings.

```
{
    "project_name": "restaurantServer",
    "snapshot_number": 1,
    "services": [
        {
            "id": "0",
            "files": [
                "pl.edu.wat.wcy.pz.restaurantServer.security.WebSecurityConfiguration",
                "pl.edu.wat.wcy.pz.restaurantServer.security.jwt.JwtAuthEntryPoint",
                "pl.edu.wat.wcy.pz.restaurantServer.security.jwt.JwtAuthTokenFilter",
                "pl.edu.wat.wcy.pz.restaurantServer.security.jwt.JwtProvider"
            ],
            "new_classes": [
                "UserDetailsServiceImplRequestInterfaceImpl",
                "UserPrincipleRequestInterfaceImpl",
                "JwtProviderHandleRequest"
            ],
            "service_calls": [
                {
                    "type": "synchronous",
                    "protocol": "HTTP",
                    "target": "loadUserByUsername",
                    "target_service": "1",
                    "requester": "JwtAuthTokenFilter",
                    "owner": "pl.edu.wat.wcy.pz.restaurantServer.security.service.
                        UserDetailsServiceImpl"
                },
                {
                    "type": "synchronous",
                    "protocol": "HTTP",
                    "target": "getUsername",
                    "target_service": "1",
                    "requester": "JwtProvider",
                    "owner": "pl.edu.wat.wcy.pz.restaurantServer.security.service.
                        UserPrinciple"
                }
            ],
            "dtos": [
```

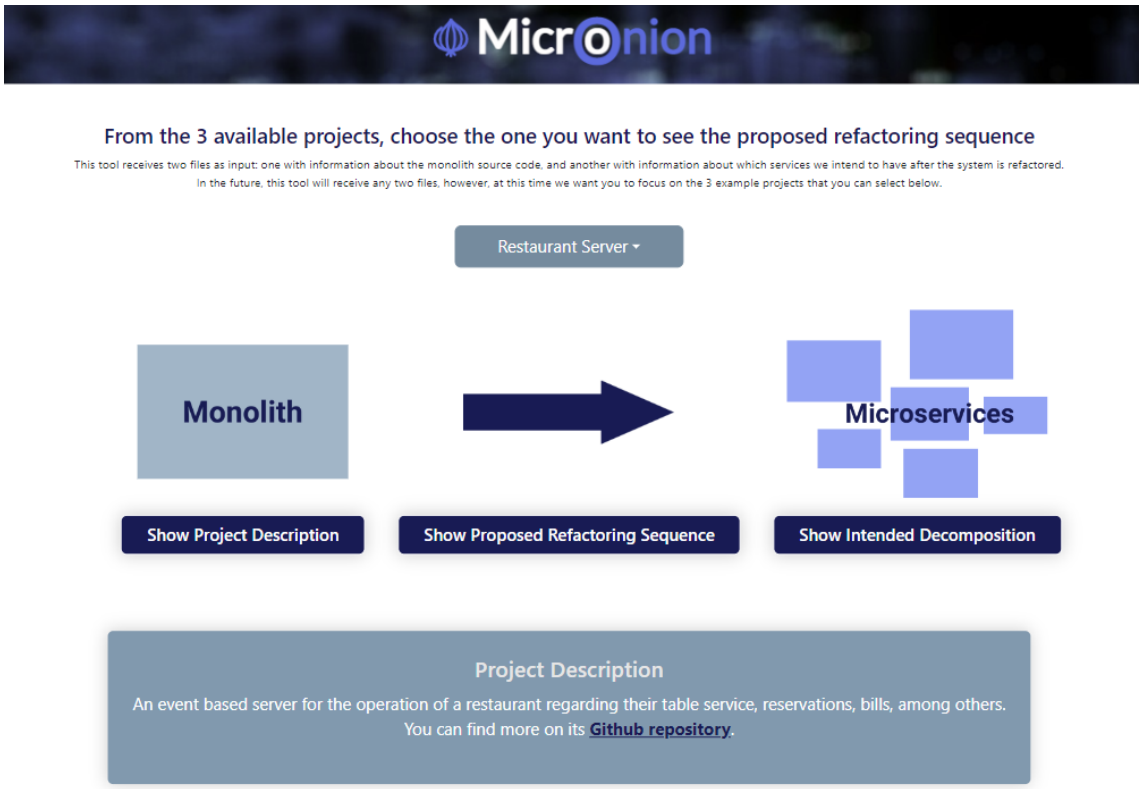
```
                "UserDetailsServiceImplDTO"
            ],
            "interfaces": [
                "UserDetailsServiceImplDTOInterface",
                "UserDetailsServiceImplRequestInterface",
                "UserPrincipleRequestInterface"
            ],
            "dependencies": {},
            "independent": true
        },
        {
            "id": "1",
            "files": [
                "pl.edu.wat.wcy.pz.restaurantServer.security.service.UserDetailsServiceImpl",
                "pl.edu.wat.wcy.pz.restaurantServer.repository.UserRepository",
                "pl.edu.wat.wcy.pz.restaurantServer.entity.User",
                "pl.edu.wat.wcy.pz.restaurantServer.security.service.UserPrinciple",
                "pl.edu.wat.wcy.pz.restaurantServer.service.UserService",
                "pl.edu.wat.wcy.pz.restaurantServer.controller.UserController"
            ],
            "new_classes": [
                "UserDetailsServiceImplHandleRequest",
                "UserPrincipleHandleRequest"
            ],
            "service_calls": [],
            "dtos": [],
            "interfaces": [],
            "dependencies": {
                "5": {
                    "User": [
                        [
                            "pl.edu.wat.wcy.pz.restaurantServer.entity.Reservation",
                            "methodVariable",
                            "databaseDependency"
                        ]
                    ],
                    "UserService": [
                        [
                            "pl.edu.wat.wcy.pz.restaurantServer.entity.Reservation",
                            "methodVariable"
                        ]
                    ],
                    "UserController": [
                        [
                            "pl.edu.wat.wcy.pz.restaurantServer.entity.Reservation",
                            "methodVariable"
                        ]
                    ]
                },
                "6": {
                    "User": [
                        [
                            "pl.edu.wat.wcy.pz.restaurantServer.entity.Role",
                            "methodVariable",
                            "databaseDependency"
                        ]
                    ],
```

```
                    "UserController": [
                        [
                            "pl.edu.wat.wcy.pz.restaurantServer.email.MailService",
                            "methodInvocation",
                            "variableType"
                        ]
                    ]
                }
            },
            "independent": false
        },
        ...
    ]
}
```

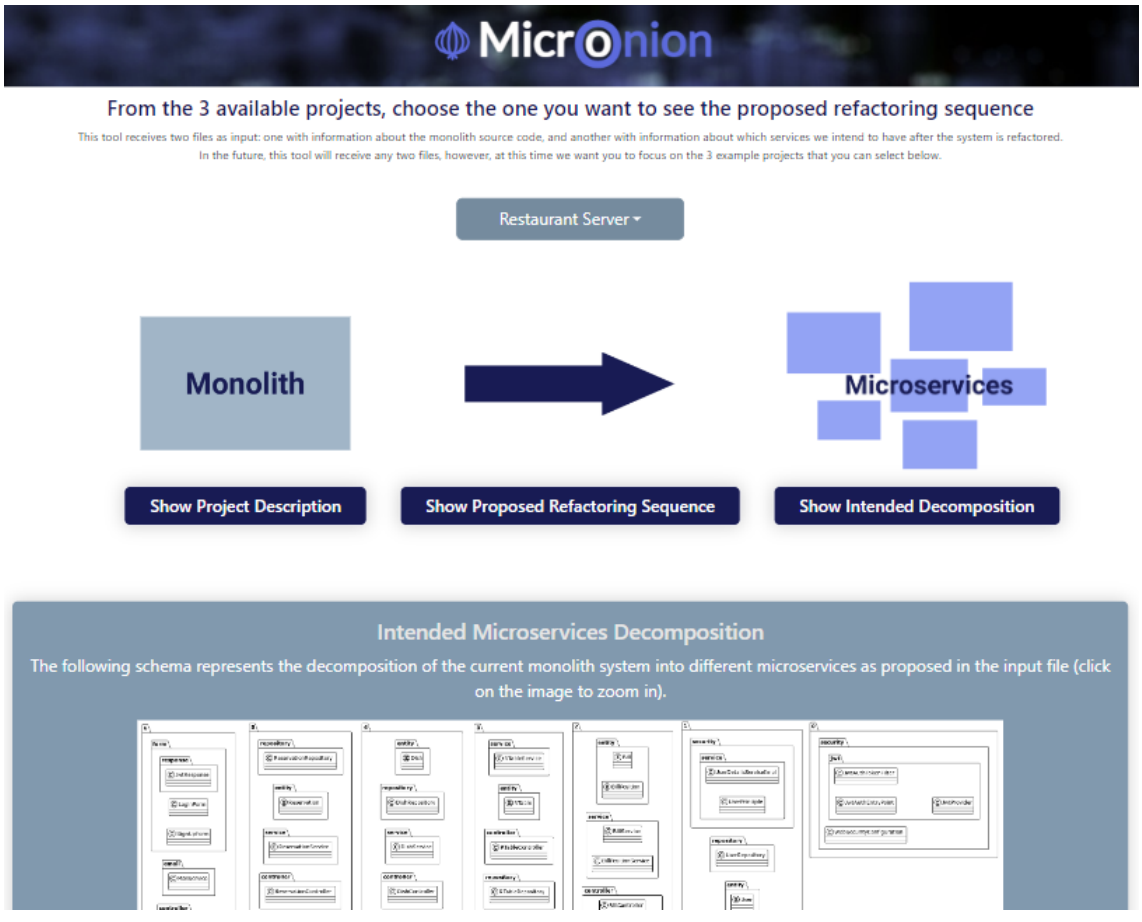
Listing D.3: Example output file schema representing the system's evolution

D.4 Choose Project Page

This section shows the Choose Project Page's screenshots on a larger scale in Figure D.1.



(a) Project's Description



(b) Intended Decomposition

Figure D.1: Choose Project Page

D.5 *Proyecto UNAM*'s services extraction order

This section shows the *Proyecto UNAM*'s services extraction sequence screenshot on a larger scale in Figure D.2.

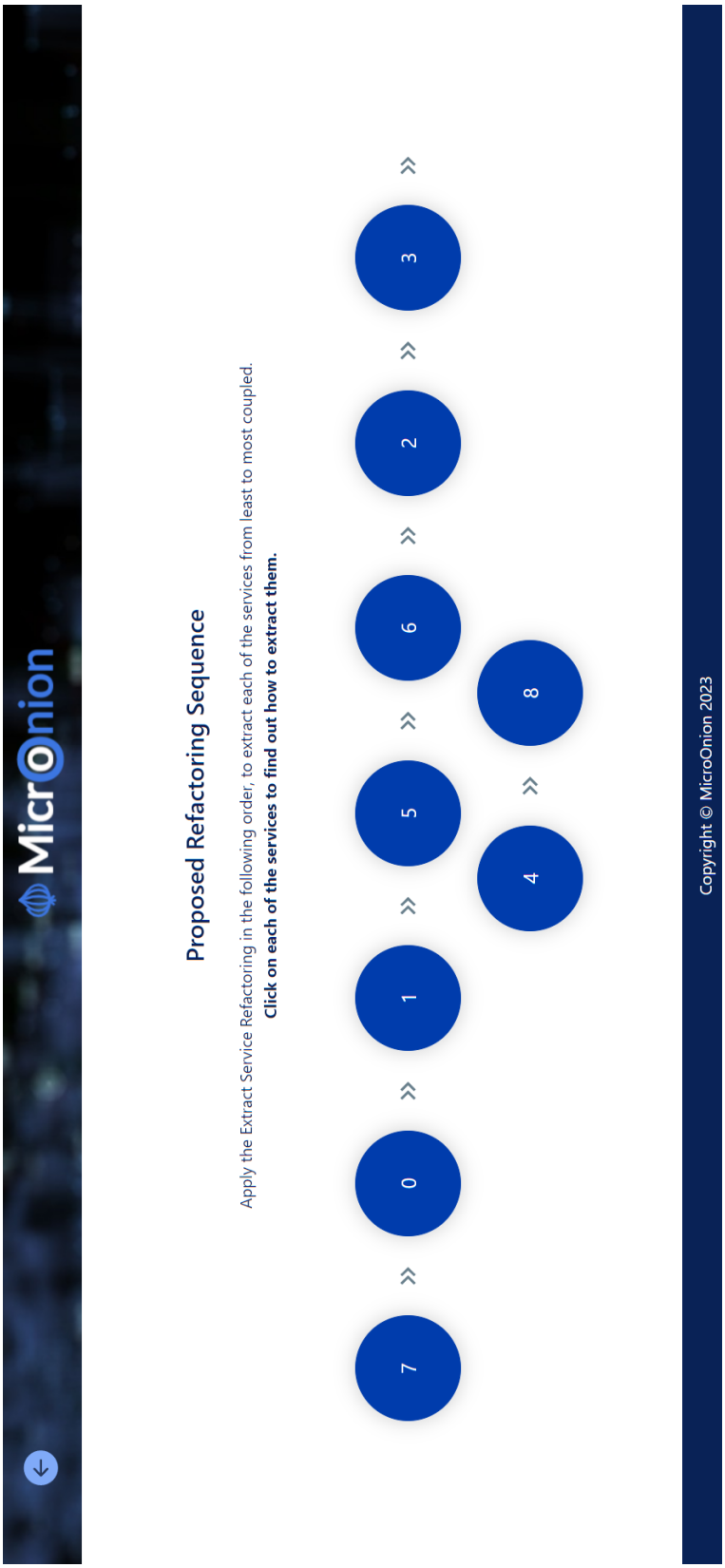


Figure D.2: Project's services extraction sequence

D.6 Extract Service Page

This section shows the Extract Service Page screenshots on a larger scale in Figures D.3 and D.4.

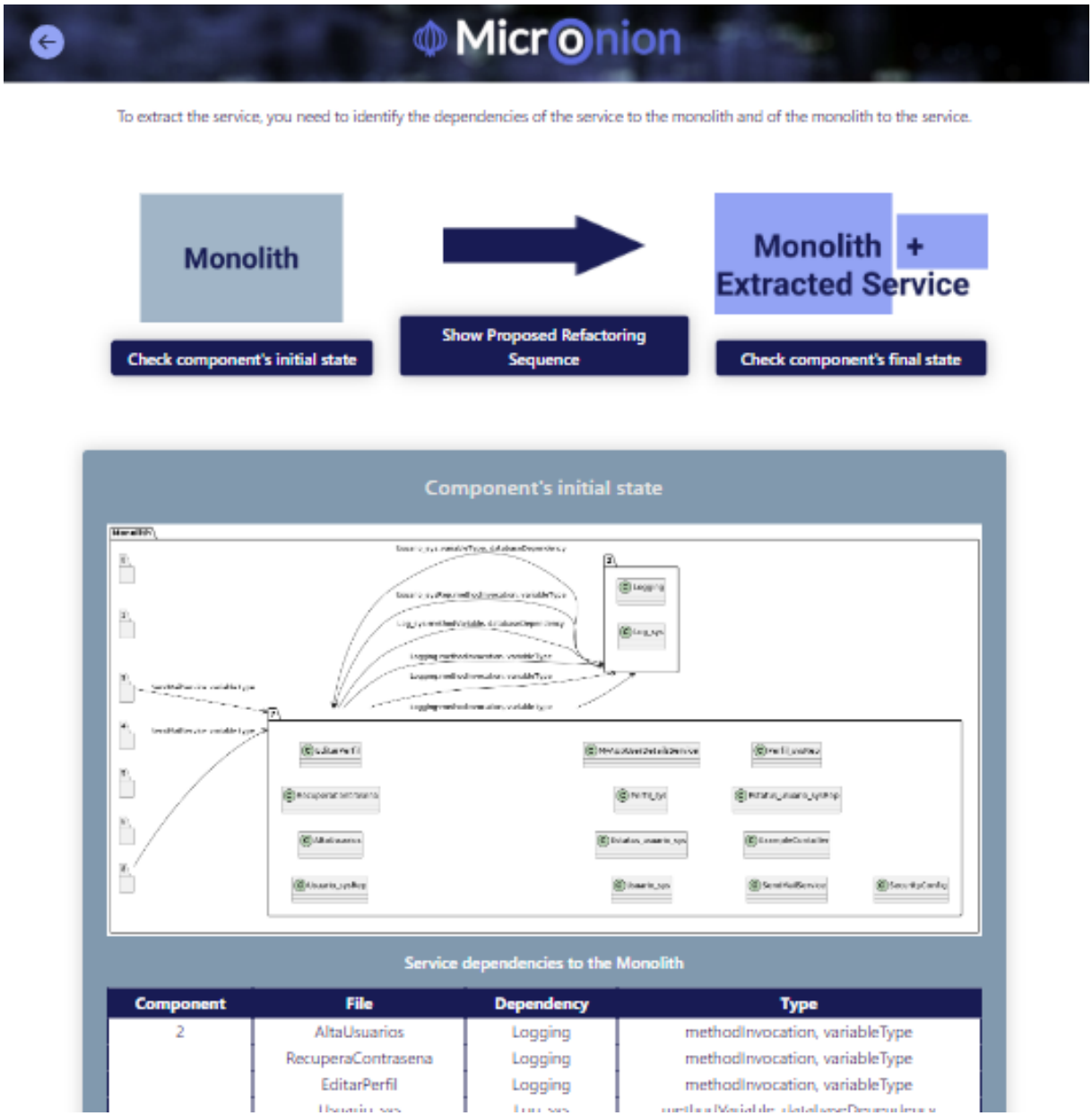


Figure D.3: Extract Service Page - Component's initialstate

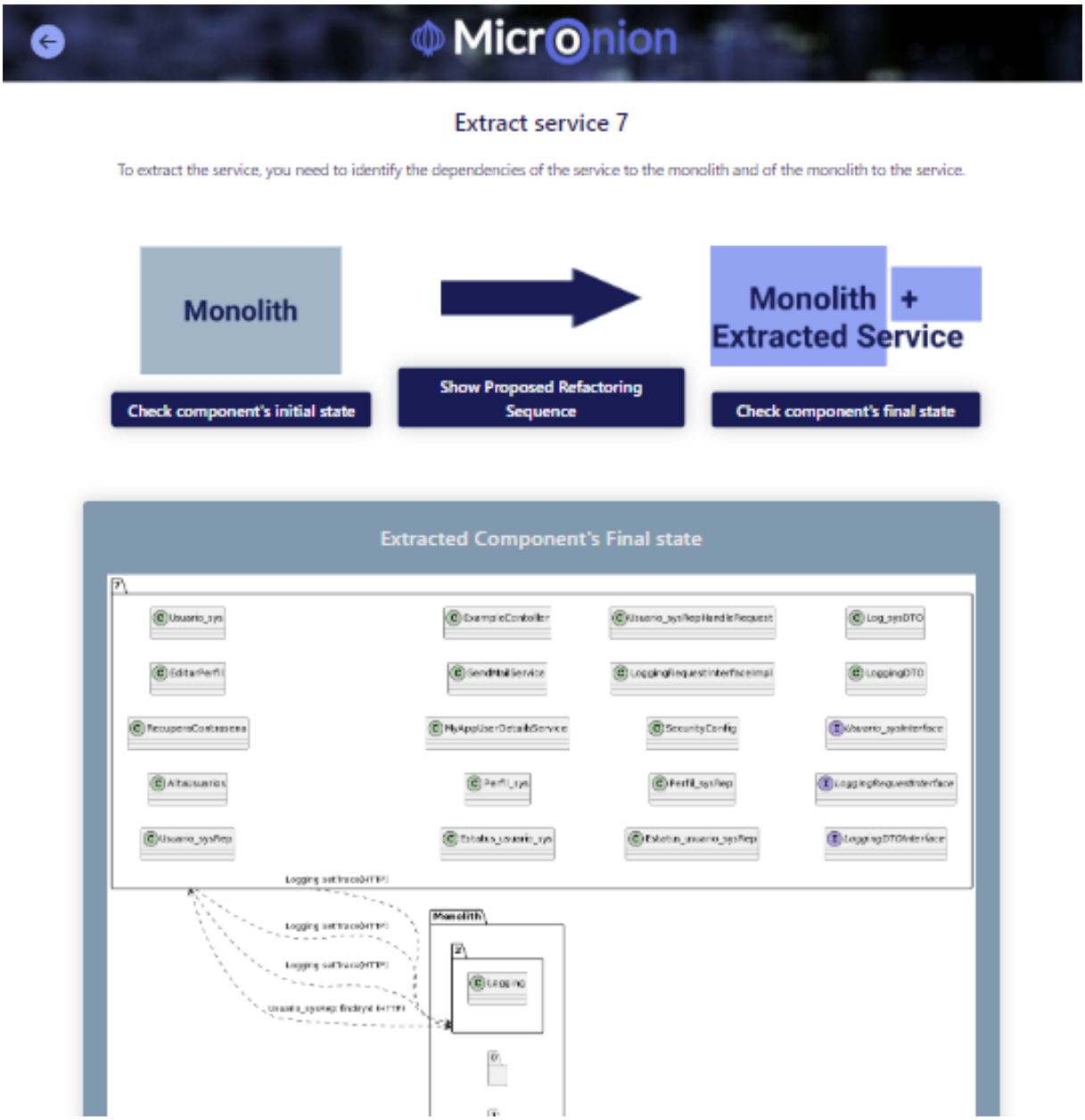


Figure D.4: Extract Service Page - Component's final state

D.7 Choose Project Page

This section shows the Choose Project Page screenshots on a larger scale in Figures D.5 and D.6.

Figure D.5: Choose Project Page - Project's Description

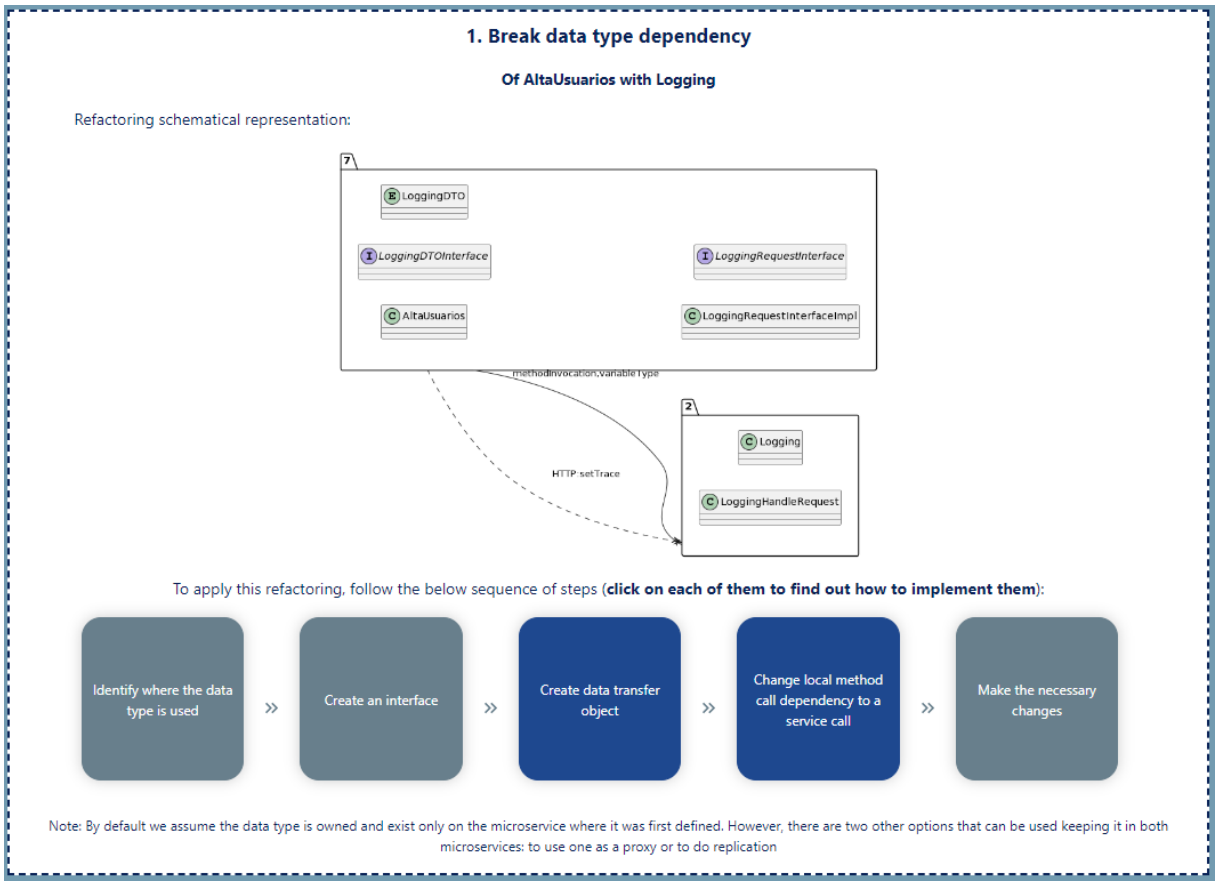


Figure D.6: Choose Project Page - Intended Decomposition

D.8 System before service 1 extraction

This section shows the System before service 1 extraction screenshot on a larger scale in Figure D.7.

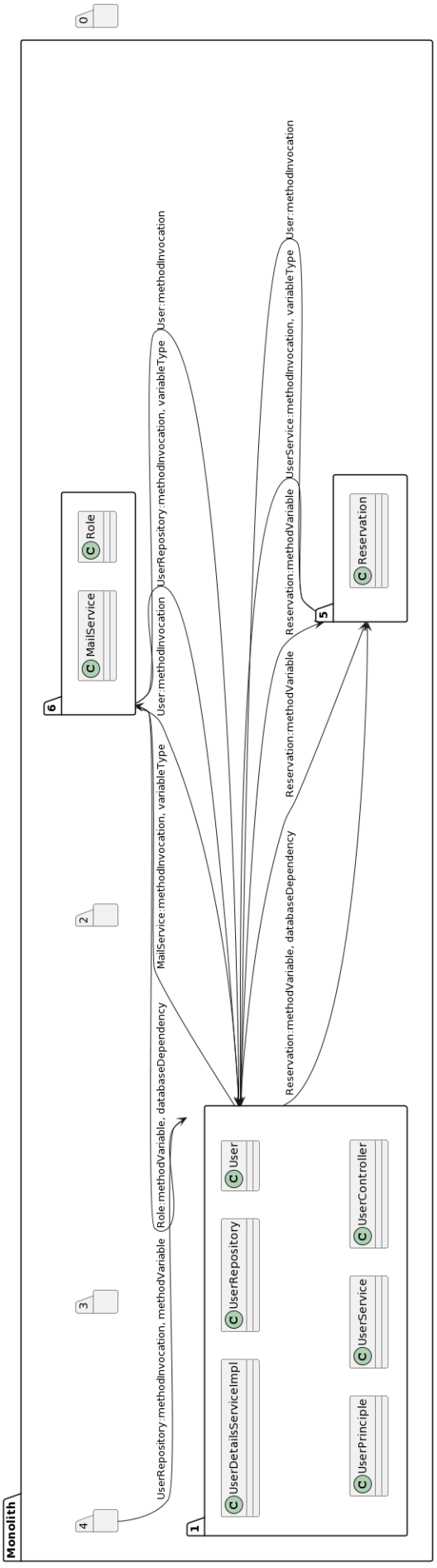


Figure D.7: System before service 1 extraction

D.9 System after service 1 extraction

This section shows the System after service 1 extraction screenshot on a larger scale in Figure D.8.

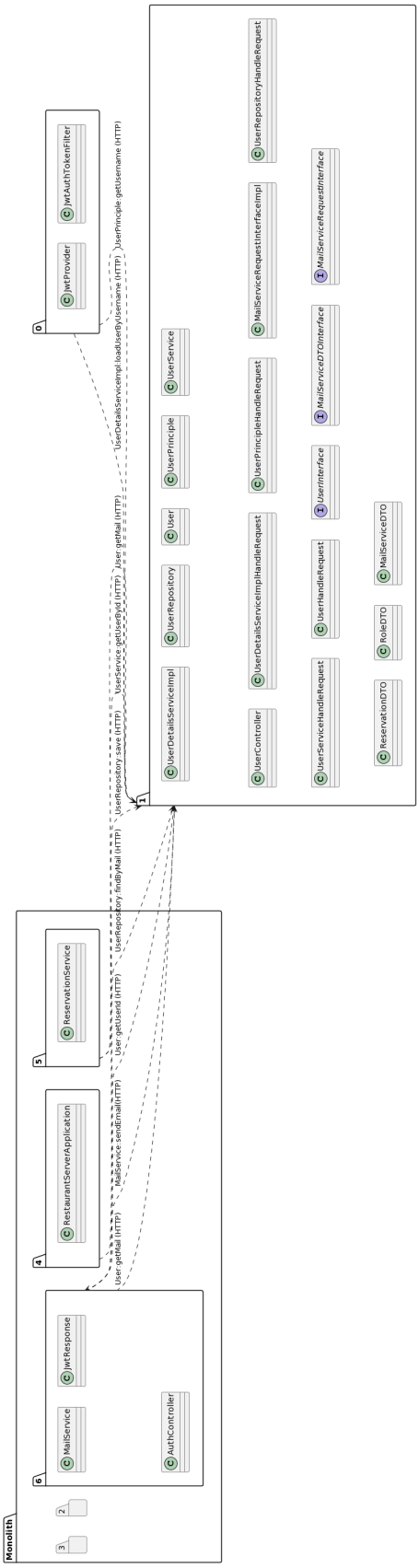


Figure D.8: System after service 1 extraction

D.10 Change local method call dependency to service call schematic representation

This section shows the Change local method call dependency to service call schematic representation screenshot on a larger scale in Figure D.9 and the corresponding PlantUML code that generates it in Listing D.4.

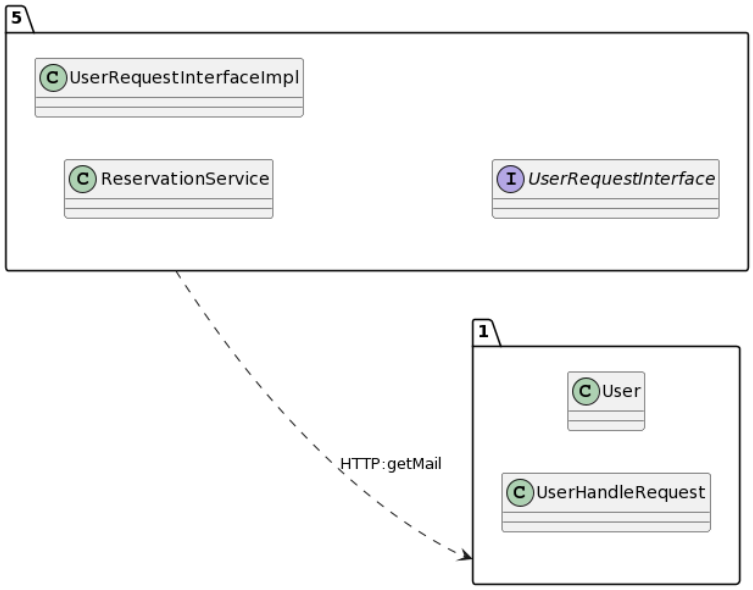


Figure D.9: Change local method call dependency to service call schematic representation

```
@startuml
allow_mixing
left to right direction
package "5"{
class ReservationService
class UserRequestInterfaceImpl
interface UserRequestInterface

}
package "1"{
class User
class UserHandleRequest

}
"5" ..> "1":HTTP:getMail
@enduml
```

Listing D.4: PlantUML code to generate diagram in Figure D.9

D.11 Move foreign-key relationship to code and Change data ownership schematic representation

This section shows the Move foreign-key relationship to code and Change data ownership schematic representation screenshot on a larger scale in Figure D.10 and the corresponding PlantUML code that generates it in Listing D.5.

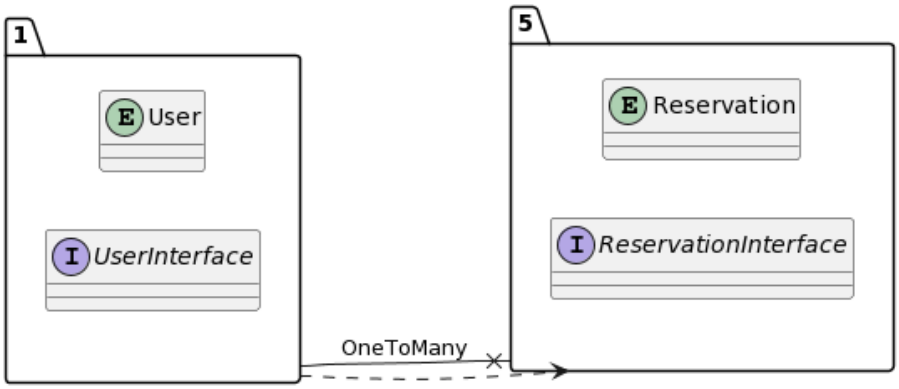


Figure D.10: Move foreign-key relationship to code and Change data ownership schematic representation

```
@startuml
allow_mixing
left to right direction
package "1"{
entity User
interface UserInterface

}
package "5"{
entity Reservation
interface ReservationInterface

}
"1" --x "5":OneToMany
"1" ..> "5"
@enduml
```

Listing D.5: PlantUML code to generate diagram in Figure D.10

D.12 Break data type dependency schematic representation

This section shows the Break data type dependency schematic representation screenshot on a larger scale in Figure D.11 and the corresponding PlantUML code that generates it in Listing D.6.

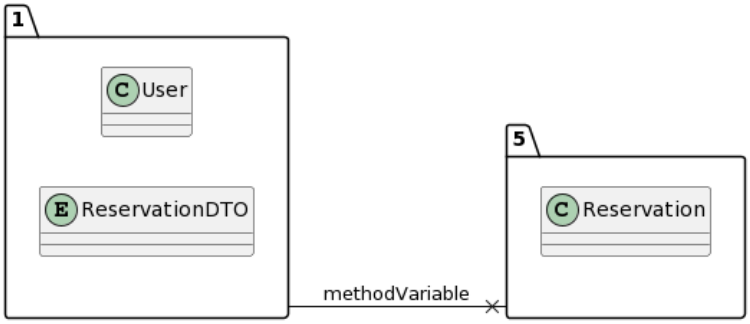


Figure D.11: Break data type dependency schematic representation

```
@startuml
allow_mixing
left to right direction
package "1"{
class User
entity ReservationDTO
}
package "5"{
class Reservation
}
"1" --x "5":methodVariable
@enduml
```

Listing D.6: PlantUML code to generate diagram in Figure D.11

D.13 Create data transfer object schematic representation

This section shows the Create data transfer object schematic representation screenshot on a larger scale in Figure D.12 and the corresponding PlantUML code that generates it in Listing D.7.

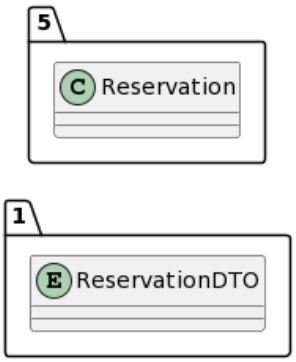


Figure D.12: Create data transfer object schematic representation

```
@startuml
allow_mixing
left to right direction
package "1"{
entity ReservationDTO

}
package "5"{
class Reservation
}
@enduml
```

Listing D.7: PlantUML code to generate diagram in Figure D.12

D.14 File dependency schematic representation

This section shows the File dependency schematic representation screenshot on a larger scale in Figure D.13 and the corresponding PlantUML code that generates it in Listing D.8.

```
@startuml
allow_mixing
left to right direction

folder 1{
  class UserController
}

folder 0{
  class UserController
}
@enduml
```

Listing D.8: PlantUML code to generate diagram in Figure D.13

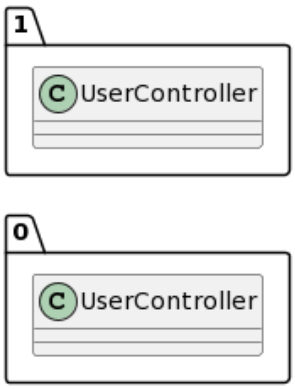


Figure D.13: File dependency schematic representation

Appendix E

Empirical Evaluation Survey

The following survey is a copy of the survey handed to developers that participated in the interviews of our empirical evaluation (cf. 8).

MicroOnion - Assisted Refactoring Towards a microservice architecture

This survey is part of ongoing research on refactoring towards a microservice architecture.

We are validating the output of our research work: a **catalogue of refactorings** and an **assisted refactoring tool (MicroOnion)**.

Please answer it you have been involved in at least one migration of a monolith to a microservices architecture. The expected time to answer is around 30 minutes.

* Indica uma pergunta obrigatória

Confidentiality Statement

By collaborating on this research, you understand that the data collected through this form:
1. Includes no personally identifying information.
2. Will be used by the researchers exclusively for the purpose of scientific inquiry.
3. May be published by the researchers (e.g., in journals, conferences, or blog posts).
4. Will go through additional anonymization and aggregation steps before being published.
5. Will be kept by the researchers in perpetuity and can be used for future academic studies.
6. Is collected using Google Forms and, therefore, its collection and use is subject to Google's Privacy Policy (policies.google.com/privacy).

1. Experience and Background (~3min)

In this section, we want to understand your experience and background.

1. 1.1 What areas have you been working on these past 5 years? *
Tick all that apply.

Marcar tudo o que for aplicável.

- [ ] Software Development
- [ ] Software Architecture
- [ ] Operations
- [ ] Quality Assurance
- [ ] Product Management
- [ ] Coaching
- [ ] Teaching
- [ ] Scientific Research
- [ ] Infrastructure
- [ ] Outra: _____

2. 1.2. What is your title? *
E.g., Senior Developer, Architect, Principal Software Engineer, Tester, etc.

_____

3. 1.3. Which country are you working from? *

_____

4. 1.4. What is your professional experience in Microservices (in years)? *

_____

5. 1.5. How many projects that included the migration of a monolith to microservices have you been involved in? *

_____

6. 1.6. What were the domain areas of these projects? *

Marcar tudo o que for aplicável.

- [ ] Healthcare
- [ ] Finance
- [ ] Retail (including E-commerce)
- [ ] Social Media & Marketing
- [ ] Security
- [ ] Manufacturing
- [ ] Education
- [ ] Travel and Tourism
- [ ] Insurance
- [ ] Construction
- [ ] Real Estate
- [ ] Supply Chain Management
- [ ] Automotive
- [ ] Media & Entertainment
- [ ] IT services
- [ ] Mobility and Transport
- [ ] Government
- [ ] Outra: _____

7. 1.7. Roughly how many monthly active users did these systems serve when the process of migrating to microservices was started? *

Marcar apenas uma oval.

- ◯ < 100
- ◯ 100 - 1K
- ◯ 1K - 10K
- ◯ 10K - 100K
- ◯ 100K - 1M
- ◯ > 1M

8. 1.8. Roughly how many people were working on these systems when the process of migrating to microservices was started? *

Marcar apenas uma oval.

- ◯ < 10
- ◯ 10 - 50
- ◯ 50 - 200
- ◯ 200 - 600
- ◯ > 600

9. 1.9. Are you familiar with any tool that provides assistance in the refactoring towards a microservice architecture? If yes, provide us the name of the tool (if you used multiple tools separate their names by commas) *

Marcar apenas uma oval.

- ◯ No
- ◯ Outra: _____

2. Refactorings Implementation (~20min)

The main focus of this section is on refactorings for **breaking dependencies** with the purpose of extracting services from a monolith.

Here we are going to present you 5 refactorings: when to apply it, how to apply it and a simple example of it.

What do we mean by *breaking dependencies*? For example, when **Component A** uses a method from **Component B**, we will break this dependency by changing the local method call to a service call, making B a remote service. These refactorings do **not** handle the additional problems of breaking this dependency, such as A being able to function properly when B is not available.

*Note that we refer to a **component** here as a unit that has been recognized to become a microservice when the architecture is eventually decomposed to microservices but has yet to be extracted, and so remains a part of the monolith but will become independent of the others during the migration.*

Change local method call dependency to service call

Motivation

When we split the monolith into microservices, it is prevalent to have code dependencies in the form of a method invocation between components. As the classes in question will become part of different services, method invocations often have to be refactored to service calls in the microservices architecture. This service call should be made with a protocol that can be synchronous or asynchronous, depending on the requirements and goals.

When we don't want a service waiting for the other to respond and an instant response is not required because we don't need that information immediately, asynchronous calls are usually a good option. This is the case when we accept having eventual consistency when it comes to data operations. It is especially common when a service call triggers other service calls, and we don't want the first service to be busy waiting for these calls to complete. It helps in scalability and availability. It is common to implement it through an asynchronous remote procedure call or messaging with a publisher/subscriber protocol, where services publish messages to a message broker, and the subscriber consumes the messages when available to process them.

Asynchronous calls are not the best solution when we want to make sure we are reading accurate data and need consistency in the moment of the action, as consistency in asynchronous calls is eventual; in this case, a synchronous request/response protocol is usually preferred.

Mechanics

1. Decide the communication strategy and make the initial configurations to use it
    1. **Synchronous** (using strategies like REST or RPC, for example)
        1. store the necessary information (e.g. URL) to make the remote calls to the microservice.
    2. **Asynchronous** (with technologies like Mosquitto, RabbitMQ, Apache Kafka, etc.): using strategies like Event Sourcing or some form of asynchronous RPC.
        1. Set up a message broker/event bus.
        2. Create a topic.
2. Configure the microservice that makes the invocation to use the communication strategy.

1. **Synchronous** - Change the method calls to and from local components to be remote calls to a different service:
    1. Create an interface with the declaration of the identified methods.
    2. Create a class that implements that interface and makes the service calls, a Request Class.
2. **Asynchronous**
    1. This microservice will act like a subscriber so make it subscribe to the topic you have created.

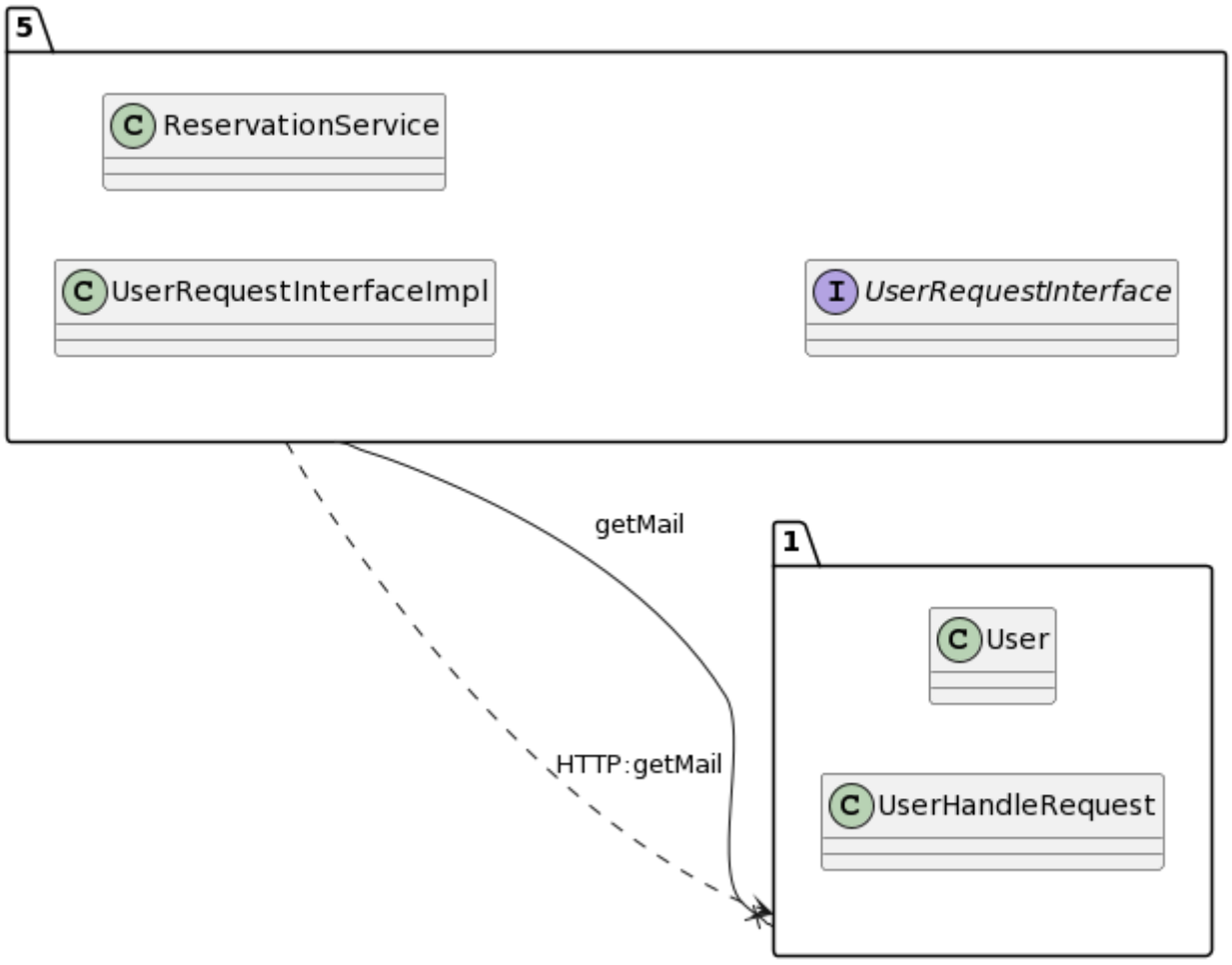
3. Configure the microservice that has the method to use the communication strategy.
    1. **Synchronous** - arrange the microservice owning the method to respond to this communication protocol, creating an API to respond to the service calls. Example:
        1. Create a class that defines the resource paths for the requests and processes them producing a response.
        2. Add methods to the class to perform the actions required by the service calls.
    2. **Asynchronous**
        1. This microservice will act like a publisher so you make it push the messages it wants to communicate in the topic created.

Example

Change local calls to method User.getMail() to a remote service call
ReservationService performs a method call of **getMail** from **User**. As these two files now belong to different microservices, this call needs to be remote.

Therefore, a **UserRequestInterface** and its implementation (**UserRequestInterfaceImpl**) are created on the side of the requester and the **UserHandleRequest** is created on the side of the method owner. This way, the method call is performed remotely via **HTTP**, as is expressed in the following schema.

Note: the call could be asynchronous or synchronous, but as the ReservationService needs the email to send the booking confirmation, it is better for it to be synchronous.



10. 2.1. Have you previously applied this refactoring? *
Either manually or through the use of a refactoring tool.

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

11. 2.2. Do you agree with the above steps (mechanics) for applying this refactoring? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

12. 2.3. Can you think of any circumstances where you would want to apply this refactoring but the above mechanics is not suitable or sufficient to do so? *

13. 2.4. Does the figure illustrate clearly what happens to the system in this refactoring? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

14. 2.5. What information do you think is crucial for the figure to convey yet is missing? *

Move Foreign-key relationship to code

Motivation

When two entities are related and dependent on one another, their relationship is frequently characterized by a foreign-key relationship between the database tables that represent each entity. One-to-one, Many-to-One, and One-to-Many relationships are possible.

When we extract a service and we realize that these entities should be in different microservices, the database tables that describe them should belong to different database schemas, as each microservice should have its own database.

Because one service will own the table containing the foreign key and another will own the table from which the foreign key comes, we must break the database and eliminate the foreign-key relationship. Therefore, as the constraint is no longer in the database, we need to move this relationship to the code itself.

Mechanics

The following steps can either be performed after breaking the code or with the code breakage in mind. Whenever services are mentioned, they are mentioned as what will be the end service division, but they do not have to be already implemented.

1. Remove the foreign-key constraint from the table that stores it.
2. In the class of entity (database table) that used to have the foreign-key constraint, create an instance variable that represents the other entity involved in the said relationship and create a column for that variable in this entity table. This variable will no longer be a foreign key but a filter of the select query to retrieve data.
3. Separate the tables into the databases of the different owners (at this moment, this might be more conceptual, but in the future, this will represent the databases of the different microservices).
4. Create an interface for each of these databases that implements the methods of data manipulation.
5. Identify the methods that use/manipulate data from different databases and change them to use the newly created interfaces.
6. When you separate the services, don't forget to use the previous refactoring to **"Change local method call dependency to a service**

call”, to change these local methods to service calls using the primary key as a parameter.

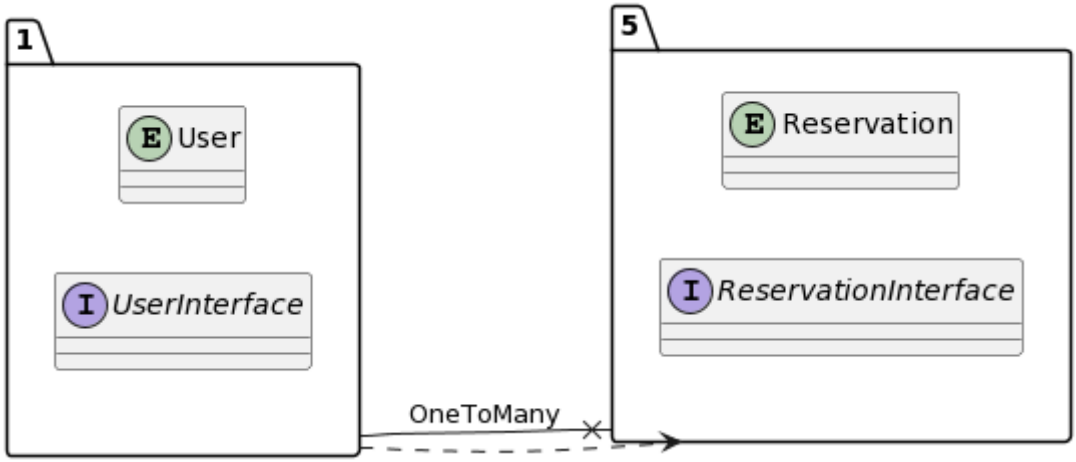
Note: We may need to remove code annotations when using specific programming languages, frameworks or ORMs that use it. The way we join the information of these two entities is no longer through a join query, so data consistency has to be a concern. Do not forget to implement mechanisms to guarantee data integrity and consistency. Additionally, we should be aware that the latency of requests increases as we transform the database calls into service calls.

Example

Move foreign-key relationship to code of the OneToMany relationship between User and Reservation

User and **Reservation entities(E)** have a OneToMany relationship, however there are no method calls between the entities.

Therefore, you remove the constraint (step 1) and create a user variable on the Reservation table (step 2). The User table goes to the database of microservice 1 and the Reservation table to the database of microservice 5 (step 3). Created a interface on each of the microservices for their entities (**UserInterface** and **ReservationInterface**) (step 4) and the **OneToMany** relationship is moved to code, making the existing and future method **calls remote**, as is expressed in the following schema (step 5 and 6).



15. 2.6. Have you previously applied this refactoring? *
Either manually or through the use of a refactoring tool.

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

16. 2.7. Do you agree with the above steps (mechanics) for applying this refactoring? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

17. 2.8. Can you think of any circumstances where you would want to apply this refactoring but the above mechanics is not suitable or sufficient to do so? *

18. 2.9. Does the figure illustrate clearly what happens to the system in this refactoring? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

19. 2.10. What information do you think is crucial for the figure to convey yet is missing? *

Replicate data

Motivation

Sometimes, different microservices need the same data, but if they access the same data source, they won't be totally independent, as one microservice will be also managing the data of another.

To solve this, each service can have its own dedicated database with replication to the shared data source. Ideally, in a microservices architecture, this should occur with no distributed transaction, assuming eventual consistency; however, synchronous replication is also possible. One of the services is the owner and source of truth of the data, the other simply holds a copy of the data it needs to access and manipulate.

Mechanics

1. Split the database by deciding which service will be the owner of the shared data.
2. There are a few strategies to replicate the data:
    1. Using mechanisms of the database engine, you can create one or more replication channels between it and the shared data source.
    2. Using event sourcing, a method of storing (or communicating) data, which facilitates data replication because events may be easily repeated. It is a way to keep eventual consistency. It stores events that are frequently objects, and because event sourcing does not need to know its consumers, other technologies can be utilized concurrently (for more on event sourcing, check Martin Fowler's article on it here).
    3. Or by using **"Change Data Capture"** refactoring.

Example

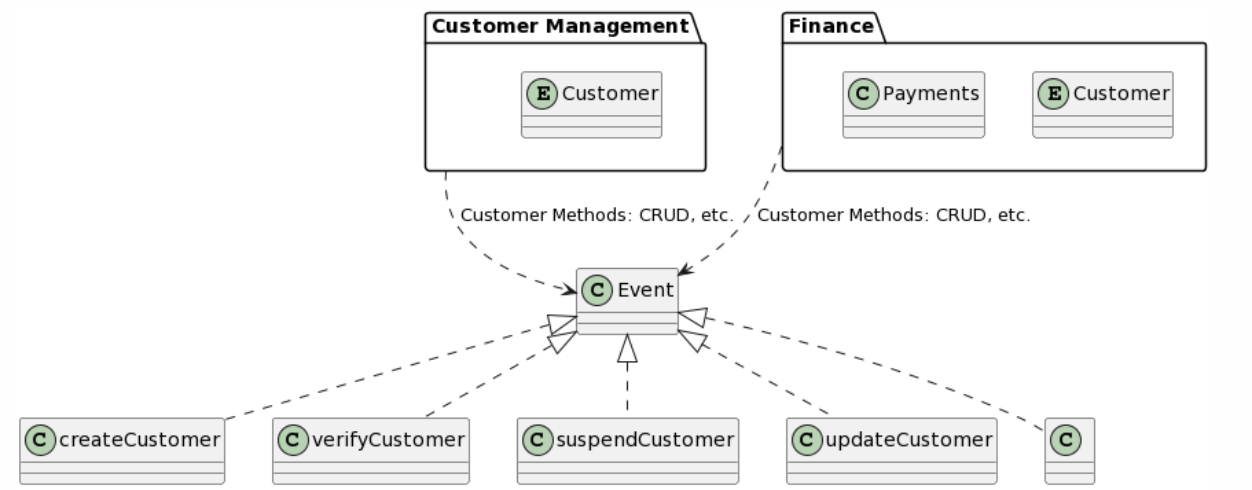
Replicate Customer data to Customer Management and Finance

Imagine having a database table "**Customer**" that has three columns: id, name and status (can have three values: NOT_VERIFIED, VERIFIED, SUSPENDED).

This table is updated by two microservices: **Customer Management**, that verifies the customers accounts and **Finance** that changes the customer status to SUSPENDED whenever a customer has unpaid bills.

Here, one solution is to replicate the data and use event sourcing to keep data consistency.

In this case, both the service keep the "**Customer**" entity (E) and whenever they perform actions on this data, an event for that action is created, so that the other service can replicate it to update its data on the "Customer" entity, as is expressed in the following schema



20. 2.11. Have you previously applied this refactoring? *
Either manually or through the use of a refactoring tool.

Marcar apenas uma oval.

- Yes
- No

21. 2.12. Do you agree with the above steps (mechanics) for applying this refactoring? *

Marcar apenas uma oval.

- Yes
- No

22. 2.13. Can you think of any circumstances where you would want to apply this refactoring but the above mechanics is not suitable or sufficient to do so? *

23. 2.14. Does the figure illustrate clearly what happens to the system in this refactoring? *

Marcar apenas uma oval.

- Yes
- No

24. 2.15. What information do you think is crucial for the figure to convey yet is missing? *

Split Database

Motivation

When extracting a service, we commonly find that some monoliths aggregate in the same database table data that will further need to be split across different microservices, as different microservices access the same database table. Therefore, splitting a monolithic database is not so trivial.

Mechanics

1. Separate into each microservice database only the tables that are only accessed/manipulated by that microservice.
2. Find the columns in the table used by the different newly defined microservices.
3. Analyze which is the case of each table and what solution better fits your system's requirements:
    1. Two microservices access the same database table but do not update the same columns
        1. You can replicate the data for both microservices using **"Replicate Data"**. and use a data replication mechanism to keep it consistent or
        2. Decide to which of these microservices each column belongs.
        3. **Split this table** inside the monolith schema between the two tables belonging to the different components that will soon be microservices.
        4. In each component, include the corresponding table and adapt the code to use that table
        5. If the different microservices interact with what used to be foreign keys on the monolith schema, use the **"Move Foreign-key relationship to code" refactoring**.
    2. Two microservices use the same database table and update the same columns
        1. You can replicate the data for both microservices using **"Replicate Data"**. and use a data replication mechanism to keep it consistent or
        2. Decide which microservice should own this data.
        3. Make the other microservice make a service call to update this column.

1. To perform this incrementally, we can first make the change in the monolith to the soon microservice that does not own the data, make the update through a method call, and then, when creating the microservices, use the refactoring **“Change local method call dependency to a service call”**.

Note: Guarantee data consistency.

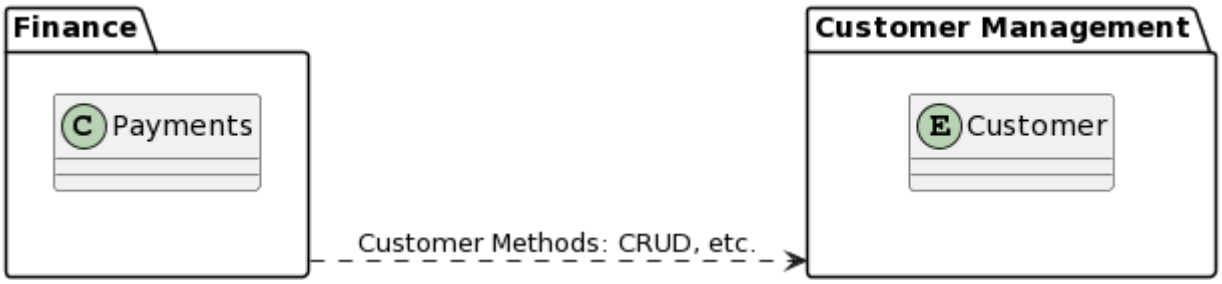
Example

Split database – Customer table

Using the same example as in the previous refactoring:

Imagine having a database table "**Customer**" that has three columns: id, name and status (can have three values: NOT_VERIFIED, VERIFIED, SUSPENDED)
This table is updated by two microservices: **Customer Management**, that verifies the customers accounts and **Finance** that changes the customer status to SUSPENDED whenever a customer has unpaid bills.

Another solution is to decide which microservice should own the data, in this case the Customer Management as it is the responsible for that data and make the other microservice make service calls anytime it wants to interact with that data , either for CRUD methods or other methods implemented in that entity, as is expressed in the following schema.



25. 2.16. Have you previously applied this refactoring? *
Either manually or through the use of a refactoring tool.

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

26. 2.17. Do you agree with the above steps (mechanics) for applying this refactoring? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

27. 2.18. Can you think of any circumstances where you would want to apply this refactoring but the above mechanics is not suitable or sufficient to do so? *

28. 2.19. Does the figure illustrate clearly what happens to the system in this refactoring? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

29. 2.20. What information do you think is crucial for the figure to convey yet is missing? *

Break data type dependency

Motivation

A data type dependency is common between different microservices, as usual, when we separate the microservices by business capabilities, some microservices might still need information about some entities in specific parts of their operations that are part of a different business capability. This dependency can appear on instance variables types, parameter types, return types and even method variables types.

We must identify where this data type is used and break this dependency to separate the microservices smoothly.

Mechanics

1. Identify where the data type is used (for example, the method invocations from the data type class, variable, parameters or return types of the data type).
2. There are three ways of doing this:
    1. Assuming it belongs only to the microservice where it was first defined:
        1. if there are method invocations:
            1. Create an interface with the same name as the data type that defines the methods invocations identified to be used through the data transfer object to make service calls to the data owner.
            2. The method invocations will change from local calls to calls to the service that owns the data types and its methods, using the refactoring **“Change local method call dependency to a service call”**.
        2. The return types, variables and parameters shall use a Data Transfer Object to create the data type in the microservice because it will be sent through the service calls. Use the refactoring “**Create Data Transfer Object” (DTO)**.
        3. Make the necessary changes in the code to use the new data type and the right interface for the method calls.
    2. Keep it in both microservices
        1. Replicate the data type in both microservices and use event sourcing to ensure eventual consistency. Check the

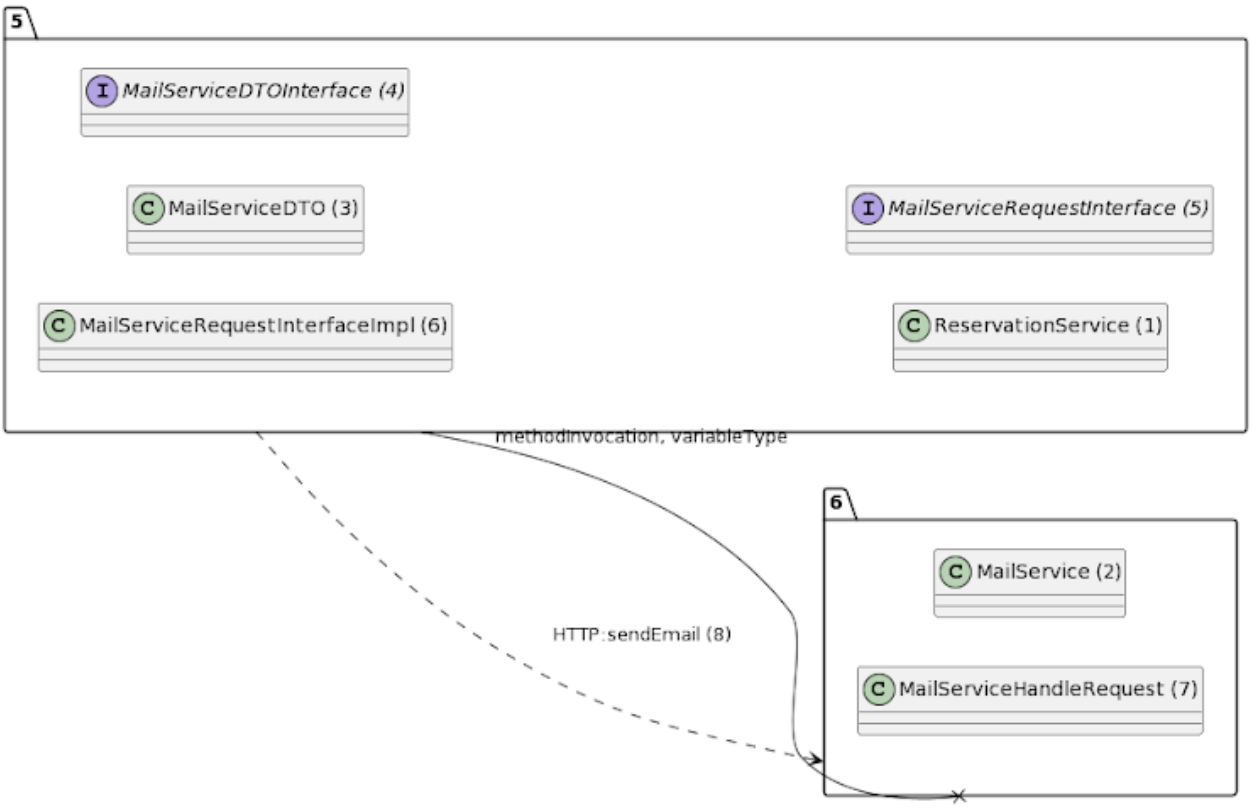
refactorings **“Change local method call dependency to a service call: asynchronous”** and **“Replicate Data”**.
    2. You can create a cache on this replication if needed.
3. Keep it in both microservices, but one of them is a proxy.
    1. One of the microservices is the source of truth, so the proxy one always has to update the data on the other microservice using some communication channel that suits the needs of the data (asynchronous or synchronous).

Example

Break Data Type Dependecy Of ReservationService with MailService

As you can see in the schema the **ReservationService (1)** had a data type dependency with **MailService (2)** because it used its variable type and made method invocations of that type.

Therefore, a **MailServiceDTO (3)** (the Data Transfer Object) was created as well as its interface (**MailServiceDTOInterface (4)**) and because there was a method invocation, the previous refactoring (Change Local Method Call Dependency to Service Call) was also applied as you can see by the existance of the files: **MailServiceRequestInterface (5)**, **MailServiceRequestInterfaceImpl (6)** and **MailServiceHandleRequest (7)** and by the remote call made from microservice 5 to microservice 6 via **HTTP** of the method "**sendEmail**" of **MailService (8)**, as is expressed in the following schema.



30. 2.21. Have you previously applied this refactoring? *
Either manually or through the use of a refactoring tool.

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

31. 2.22. Do you agree with the above steps (mechanics) for applying this refactoring? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

32. 2.23. Can you think of any circumstances where you would want to apply this refactoring but the above mechanics is not suitable or sufficient to do so? *

33. 2.24. Does the figure illustrate clearly what happens to the system in this refactoring? *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

34. 2.25. What information do you think is crucial for the figure to convey yet is missing? *

35. 2.26. We follow the premise that to extract a service at the first stage of a migration, it's necessary to *break* its dependencies to the monolith and of the monolith to this service.
Do you agree with this premise? *

Marcar apenas uma oval.

- Yes
- No
- Maybe

36. 2.27. If no, why?

37. 2.28. Do you think that is any other type of dependency that can occur that will lead to a refactoring that was not mentioned? *

Marcar apenas uma oval.

- Yes
- No

38. 2.29. If yes, please elaborate.

3. MicroOnion: A tool for assisting the refactoring to microservices (~10min)

MicroOnion aims to guide developers in migrating a monolith to microservices by graphically conveying and guiding how to carry out the migration. The main page outlines four significant concerns when doing this kind of migration. For this study, we focus specifically on the refactorings related to first concern, which we name **Extract Service**.

Explore MicroOnion

To answer the following questions, explore MicroOnion, available at this link.

Enter the "**Extract Services**" concern (/chooseProject) and choose one of the three available projects on the dropdown. To learn more about the project, you can see its description (to the left) and the intended microservices decomposition (to the right). *Be aware that, in the future this tool will be able to receive as input a decomposition for any project, and produce the steps needed to migrate it*.

By clicking on the button "Show Proposed refactoring sequence" (in the middle), you are presented with the suggested refactorings sequence for the selected project. Each circle represents an "Extract Service" refactoring of the service whose id is in the circle's center. Click on each to further explore how to apply the refactoring in the system.

Now, when you click on each of them, you are on the "Extract Service" (/extractService) page, where you can:

- Click on the button "Check component's initial state" (to the left) and check the component's initial state while it was still part of the monolith, the dependencies the service had to the monolith and its dependencies to the service.
- Click on the button "Check component's final state" (to the right) and check the component's state after the refactoring, now as an extracted service.
- Click on the "Show Proposed Refactoring Sequence" (in the middle) and check the sequence of smaller-scale refactorings you must apply to extract this service. By clicking on each smaller-scale refactoring, you can see the steps to perform it, how the system changes at each smaller-scale refactoring and even some lower-level refactorings to implement the smaller-scale ones. In all these refactorings, you have footnotes of variations you can apply that the catalogue explains in more detail.

The representation of the initial and final states of the component provides you with a high-level view of the changes applied to the system.

The Catalogue is available here.

39. **3.1. Tell us to which degree you agree with the following statements in the scope of MicroOnion:** *

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
I am familiar with the refactorings suggested by the tool.	◯	◯	◯	◯	◯
The tool in general helps me understand the process of refactoring the system to microservices.	◯	◯	◯	◯	◯
The visual representations help me understand the suggested refactorings.	◯	◯	◯	◯	◯
The tool offers sufficient information to carry out the suggested refactorings	◯	◯	◯	◯	◯
I trust the tool's recommendations.	◯	◯	◯	◯	◯

40. **3.2. Tell us to which degree you agree with the following statements in the scope of MicroOnion:** *
"My trust in the tool's recommendations is improved by ..."

Marcar apenas uma oval por linha.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Its ability to show the overall sequence of refactorings from monolith to microservices	◯	◯	◯	◯	◯
Its ability to show how the system evolves with each step.	◯	◯	◯	◯	◯
Its ease of use	◯	◯	◯	◯	◯

41. 3.3. Do you have any improvement suggestions for MicroOnion?
*Keep in mind that the tool's goal is to assist the refactoring towards a microservice architecture; not to decide on **which** services to extract, but **how** to extract them.*

42. 3.4. Currently, we order the services to extract by the number of dependencies it has with other components, starting with the one with the least dependencies. Which different strategies do you think fit this sorting? *
Please make sure to tick all that apply and add all your suggestions.

Marcar tudo o que for aplicável.

- [ ] Most dependencies first
- [ ] The ones with more database dependencies first
- [ ] Perform a brute force algorithm trying all possible orders and choosing the one that leads to the least number of refactorings
- [ ] Outra: _____

43. 3.5. Why?

_____

_____

_____

_____

_____