UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DIPARTIMENTO DI INGEGNERIA, SCIENZE DELL'INFORMAZIONE E MATEMATICA

Doctoral Program in *Information and Communication Technology*

Curriculum *Emerging computational models, software architectures, and intelligent systems.*

XXXV cycle

Thesis Title

Migration to Microservices: a Quality-Driven Approach

SSD INF/01

Ph.D. Candidate

Roberta Capuano

Doctoral Program Supervisor

Prof. Vittorio Cortellessa

Tutor

Prof. Henry Muccini

A.A. 2021/2022

Abstract

Microservices architecture has become increasingly popular among software practitioners in recent years as an effective approach to building complex applications that are more scalable, maintainable, and resilient. Top companies like Netflix, Amazon, and Uber have all successfully modernized their systems migrating them to microservices architecture. However, this migration process can pose certain challenges that require careful planning and execution to achieve desired outcomes and ensure that both functional and non-functional requirements are met.

One of the most significant challenges in microservices migration is the planning phase, as poor planning can lead to increased complexity, reduced scalability, and degraded performance of the final system. To address these challenges, we propose a quality-driven migration approach that considers software qualities in all the migration stages. Our approach aims to improve software qualities with the migration process by applying architectural refactoring techniques, such as antipatterns detecion analysis. By combining techniques from both migration and refactoring, our approach can help organizations to achieve the desired outcomes of microservices migration while improving or maintaining quality attributes throughout the process. In addition, we created a novel quality-driven (antipatterns-based) refactoring approach to be applied to microservices derived from the migration.

Given the industrial nature of the PhD of which this document represents the final Thesis, the research has been validated by applying it to a real case study from the BIM Italia company. In particular, the refactoring approach has been applied for refactoring two microservices of BIM Italia that suffered from significant performance degradation after migration. Industrial experimentation in BIM Italia has showcased the importance of quality-driven migration approaches in a microservices architecture.

This Thesis makes several contributions, including: i) conducting a thorough analysis of quality-driven migration approaches, ii) proposing a quality-driven migration process for microservices that relies on antipatterns analysis to ensure a successful migration, iii) introducing a graph-based software representation with annotations for antipatterns detection, and iv) developing and validation of a quality-driven refactoring approach for microservices resulting from the migration of a monolithic system.

The output of this Thesis is a set of guidelines for quality-driven migration to microservices that can help practitioners to avoid common pitfalls ensuring that their systems meet the expected quality requirements. Our work demonstrates the importance of considering quality attributes throughout the migration process and how architectural refactoring can help achieve these goals.

Keywords: software architecture, migration to microservices, architectural refactoring, software quality.

Contents

List of Figures

List of Tables

Acronyms

CA *Cache-Aside*

DAL *Template-Method*

DRG *Diagnosis Related Group*

DTS *Data Transformation Services*

DT *Data Taffy*

HDR *Hospital Discharge Records*

ICD-9-CM International Classification of Diseases, Ninth Revision, Clinical Modification

RQs *research questions*

RQ *research question*

SLR *Systematic Literature Review*

SSN Italian National Health Service

ToB *Tower of Babel*

MDC Major Diagnostic Categories

MSA Microservices Architecture

ROI Return on Investment

SOA Service-Oriented Architecture

SOC Service-Oriented Computing

WHO World Health Organization

WS Web Services

Chapter 1

Introduction

This Chapter provides a comprehensive overview of modernization, migration to microservices and qualities. Additionally, it defines the Thesis objectives, highlighting its contribution to the field of software architecture. The Thesis structure provides a roadmap for the reader. By the end of this Chapter, there will be a clear understanding of the scope and focus of the Thesis, as well as the key concepts and ideas that will be explored in detail throughout the document.

1.1 Software Modernization

The term "legacy system" denotes a system characterized by its inherent inability to effectively adjust or accommodate itself in response to dynamic and continuously evolving business requirements [1]. However, according to Lehman's first law [2], software needs to be continuously adapted to changing user requirements and technical environments; otherwise, it will become progressively less suitable for real-world use.

To address the challenges of increasing system flexibility and reducing maintenance costs, many companies adopted modernization processes. Software modernization is defined as *"the process of evolving existing software systems by replacing, re-developing, reusing, or migrating the software components and platforms, when traditional maintenance practices can no longer achieve the desired system properties"* [3]. Currently, different approaches are used to modernize legacy systems: refactoring and reengineering. In the first case, actions are made on the code to improve the software behaviour or meet new functional requirements [4]. The latter consists in the software analysis, re-architecting and re-implementation [5]. Despite three decades of legacy system modernization research, many legacy systems are still in daily operation, and 180-200 billion lines of legacy code are still in active use [6, 7].

Web-based technologies have played a significant role in encouraging legacy system modernization over the last 20 years [8–11]. In particular, Service-Oriented Architecture (SOA) has been a popular target architecture for legacy system modernization [3].

1.2 Microservices

Around two decades ago, many businesses were captivated by SOA, Web Services (WS), and Service-Oriented Computing (SOC) [12]. Most organizations claimed to have adopted SOA and web services as essential enablers for their projects' success. However, the lack of a uniform definition of SOA among companies made it difficult to recognize its actual value [13].

Today, Microservices Architecture (MSA), are generating the same excitement [14]. MSA is a way of designing and building software applications as a suite of independently deployable, small, modular services. Thus, each microservice runs a unique process and communicates through a well-defined, lightweight mechanism such as a RESTFul API to serve a business goal [15]. The MSA enables organizations to develop, deploy, and scale applications faster and more efficiently by focusing on each service's distinct responsibility addressing the so-called *Single Responsibility Principle*. MSA share the same advantages as SOA, including dynamism, modularity, distributed development, and heterogeneous system integration. Unlike SOA, however, microservices emphasize independence, replaceability, and autonomy [15]. The services should be conceived, implemented, and deployed independently, and different versions can even coexist while allowing the system's topology to change at runtime as needed. Additionally, each microservice component should be changeable without affecting the performance of others. Similar to SOA, microservices are not a panacea. With them, new challenges emerged, and old ones regained attention. Microservice architectures face various non-trivial design challenges intrinsic to any distributed system, including data integrity and consistency management, service interface specification and version compatibility [16]. MSA [17] is based on a few simple principles:

- *Bounded Context*: Introduced in [18], this principle highlights the importance of focusing on business capabilities in a MSA. Related functionalities are grouped together as a single business capability, which is then implemented as a service.
- *Size*: Size is a critical aspect of microservices and offers significant benefits in terms of service maintainability and extensibility. The recommended approach in a MSA is to split a service into two or more services if it is too large, preserving granularity and focusing solely on providing a single business capability.
- *Independence*: This principle promotes loose coupling and high cohesion by asserting that each service in MSA operates independently of others, with communication between services only through their published interfaces.

1.3 Migration to Microservices

In the past ten years, companies started migrating their legacy, most of the time monolithic-based software systems, to MSA to improve scalability, resilience, time to market, maintainability, and technology alignment [19]. This results in an improved ability to deliver value to customers, increased agility, and faster adaptation to changing business needs [20]. Thus, migration to microservices is defined as *the process of refactoring a monolithic application to adopt MSA*. The goal of migration to microservices is to achieve better scalability, and agility for the application, as well as improved maintainability and faster development cycles. The migration process involves breaking down the large, monolithic codebase into smaller, independent services that can be developed, tested, and deployed separately [21]. In this context, techniques such as service decomposition, API-first development, and containerization are commonly used [22, 23].

The migration to microservices can be complex and requires careful planning and execution to minimize disruption to the existing application and ensure a smooth transition to the new architecture. Properly managed migration to microservices can result in improved performance and faster innovation, increasing competitiveness for businesses [24]. In [20], San Newman provides a guideline to practitioners to migrate their legacy systems to MSA. The three crucial phases of migration to microservices can be summarized as follows:

- *Planning*: the aim of this phase, is to understand the migration goal. The objective is to analyse why to migrate to microservices. The most impacting reasons for companies are: i) time-to-market improvement, ii) embrace new technology, and iii) get a better business value. In this context, it is essential to consider that the migration to microservices has an impact on the overall organization structure. This consideration derives from Conway's Law which assesses that any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure [25].

- *Splitting the Monolith*: in this context, a company may decide to refactor the monolith using different patterns. The most efficient one is figured out to be the strangler fig pattern [26]. The idea is that the old and the new system can coexist. Thus, the new system can slowly grow and potentially replace the old system supporting its incremental migration. The techniques that can be used to decompose the monolithic system will be discussed in Chapter 3.

- *Database decomposition*: Microservices perform best when practising information hiding, encapsulating the data storage and retrieval mechanisms. Thus, when migrating toward a microservice architecture, the monolith's database splitting shall be run separately to best out of the transition.

1.4 Thesis Context

When transitioning from a monolith to a MSA, the planning phase is crucial for the process's success. During this stage, a roadmap is developed that outlines the migration's scope, goals, and potential obstacles, while also determining the optimal approach. Proper planning is necessary to evaluate whether microservices are suitable for a company's needs. Before committing to this architecture, it is essential to analyze the current systems comprehensively and weight the benefits and drawbacks of the migration [20]. The planning process involves evaluating the current infrastructure, anticipating possible issues during the migration, and designing a roadmap for moving to a MSA. This process can help identify any technical or organizational issues and ensure a seamless transition without disrupting the company's operations. To this aim, it is crucial to consider both functional and non-functional requirements during the planning phase. Neglecting non-functional aspects may result in additional refactoring, which can be time-consuming and expensive.

In [27], Li et al. found that despite the rapid adoption of MSA in the software industry, there is a significant lack of understanding about the quality attributes of MSA. To address this, they conducted a *Systematic Literature Review* (SLR) to provide a comprehensive overview of existing research on the quality attributes related to a MSA for practitioners and researchers. The authors suggest that caution should be exercised when considering migrating from monolithic systems to MSA. Practitioners should evaluate the Return on Investment (ROI) and consider the additional efforts and costs needed to implement tactics for specific quality attributes identified in the SLR or other important ones not obtained due to time constraints. Moreover, practitioners must consider the complex relationships among quality attributes during the migration, including dependencies and trade-offs. Improving one quality attribute may positively affect other quality attributes, while addressing one quality attribute may negatively impact certain quality attributes, creating trade-offs between different quality aspects.

In [28], Bogner et al. investigated how companies perceive the impact of adopting microservices on the quality of their systems, as defined by the *ISO 25010 standard* [29]. They conducted 17 interviews with representatives from 10 companies and focused on eight software qualities: maintainability, portability, reliability, compatibility, performance, usability, security, and functional sustainability. The findings revealed that some interviewees perceived a negative impact on certain aspects of software quality. Specifically, 3 out of 17 participants reported issues with maintainability, while 2 out of 17 had concerns about reliability, 1 out of 17 expressed dissatisfaction with usability, and 4 out of 17 had negative views on security. Therefore, it is necessary to determine strategies that can be implemented to achieve or improve a predefined set of software qualities during the migration to microservices.

It is challenging to find real-world case studies where a migration to MSA had

a negative outcome, resulting in a degradation of system quality. However, several scientific articles address the topic of *microservices migration not being for everyone* [30][31][32]. These articles emphasize the complexities and challenges associated with migrating to a MSA and highlight the importance of careful planning and evaluation before embarking on such a project. Therefore, to enhance the overall success of the transition and lead to better long-term outcomes for organizations, it is essential to employ strategies that ensure the migration to microservices enables the achievement or improvement of a predefined set of software qualities.

Unfortunately, as will be shown in Chapter 3, few of the approaches presented in the literature currently consider software qualities from the migration planning to its implementation. As a result, the following research question has been formulated: *What strategies can be employed to ensure that the migration to microservices enables the achievement or improvement of a predefined set of software qualities?* This approach can enhance the overall success of the transition and lead to better long-term outcomes for organizations.

1.5 Thesis Objectives and Contributions

The main Thesis objective is the definition of a *quality-driven migration to the microservices approach*. The approach aims to help organizations achieve the desired outcomes of microservices migration while improving or maintaining quality attributes throughout the process. The proposed approach takes into account non-functional requirements as a first-class entity in all the defined steps. Given the industrial nature of this Thesis, most of the research activities has focused on the validation of the proposed methods on a real-world case study which will be explored in Part III of this document. The following bullet list reports the main research objectives summarizing the Thesis contribution:

- **RO1: analysis of the state-of-the-art in migration to microservices.** The aim is twofold: i) understand how companies perceive the migration to microservices, and ii) analyse migration phases and techniques, to investigate if and how quality constraints are considered during the migration to microservices. The findings will be presented in Chapters 2 and 3. Part of the study has been published in the *IEEE 19th International Conference on Software Architecture Companion (ICSA-C)* [33].
- **RO2: definition of a *quality-driven migration to microservices approach* aiming to satisfy non-functional requirements.** The approach considers the antipatterns analysis on the monolith ensuring that none of the antipatterns will reflect on the derived microservices. A first attempt of the approach has been presented in the Doctoral Symposium at the European Conference on Software Architecture [34]. This study represents the preliminary idea of the quality-driven

migration approach and it is shown in Chapter 4. In addition, in the same Chapter, a revised version of the approach is presented. The proposed *quality-driven refactoring approach* is divided into different phases and tasks. Two of those tasks have been investigated during the three-years of research and generated the contribution presented in the RO3 and RO4 respectively.

- **RO3: creation of a *graph-based representation of a legacy system* for antipatterns detection.** The proposed representation is designed to provide a visual and intuitive way of identifying antipatterns in a legacy system, which can help developers to improve the system's performance, reliability, and maintainability through migration. The process for developing this graph-based representation and the mathematical formulation allowing to detect antipatterns have been carefully crafted has been accepted in the research track of the 17th European Conference on Software Architecture [35]. The conceptual framework and the details of its implementation and validation are presented in Chapter 5.

- **RO4: definition of a *quality-driven refactoring approach* of microservices derived from legacy, monolithic-based, software.** This approach is contained in the migration approach defined and takes into consideration not only the antipatterns detected on the monolith but also the ones detected on the microservice. The idea behind this choice is to analyse the possible relations between the two sets of antipatterns. The approach is presented in Chapter 6 and has been published in the *20th IEEE International Conference on Software Architecture (ICSA-C 2023)* [36].

- **RO5: analysis of a real-world case study to investigate the migration process adopted and monitor non-functional requirements pre and post-migration.** The aim is to acquire and understand: i) their motivations for migration, ii) the approach and technique used, and iii) analyse whether the non-functional requirements were met or not with migration. The case study is presented in Chapter 7 and has been published in the *20th IEEE International Conference on Software Architecture (ICSA-C 2023)* [36].

- **RO6: application of the *quality-driven refactoring approach* on the real-world case study.** The objective is to refactor two already implemented microservices suffering from performance issues. The application results are presented in Chapter 8 and has been published in the *20th IEEE International Conference on Software Architecture Companion (ICSA-C 2023)* [36]. In addition a modification of the *quality-driven refactoring approach* is presented and applied in Chapter 9 for a partial implementation of a new microservices of the same company. The overall industrial experience is under submission as a journal paper.

1.6 Thesis Outline

Table 1.1 provides an overview of the Thesis structure providing a roadmap for the reader. The Thesis is organized into four main parts.

- Part I provides the basic knowledge on microservices and motivations for migration while considering non-functional requirements. In particular, Chapter 2 provides an overview of how companies perceive the migration to microservice. In addittion, the state-of-the-art in *quality-driven migration to microservices* is analysed in Chapter 3.
- Part II presents the research contributions. Chapter 4 aims to describe all the steps performed to produce the proposed *quality-driven migration approach*. One of the tasks of the quality-driven migration approach, namely the graph-based representation of the system for antipatterns detection is presented in Chapter 5. Lastly, Chapter 6 provides an extension of the *quality-driven migration approach* for the refactoring of microservices derived from legacy, monolithic-based, software.
- Part III is dedicated to the industrial experience. More precisely, Chapter 7 presents the BIM Italia case study, analysing the software under the analysis of the migration goal, approach, and issues. Thus, Chapter 8 displays the application of our *quality-driven refactoring approach* on two already deployed microservices of BIM Italia. Based on the knowledge acquired, the *quality-driven migration approach* is modified and applied for the first implementation of another company's microservice in Chapter 9.
- Part IV summarizes the Thesis findings, the lessons learned, and future work on migration to microservices.

Thesis Part	***Chapter***	***Research Objective***	***Description***
Part I	Chapter 2	-	Overview of migration in industry.
	Chapter 3	RO1	Systematic Literature Rewiew on quality-driven migration. [33]
Part II	Chapter 4	RO2	Proposed quality-driven migration approach. [34][36]
	Chapter 5	RO3	Graph-based representation for antipatterns detection. [35]
	Chapter 6	RO4	Proposed quality-driven refactoring approach. [36]

Continued on next page

Table 1.1 continued from previous page

Thesis Part	***Chapter***	***Research Objective***	***Description***
Part III	Chapter 7	RO5	BIM Italia case-study presentation. [36]
	Chapter 8	RO6	Application of the quality-driven refactoring approach in BIM Italia. [36]
	Chapter 9	RO6	Implementation of the DRG functionality in BIM Italia. (Under submission)
Part IV	Chapter 10	-	Findigs, lesson learned and future works.

Table 1.1: Thesis Roadmap

Part I

Motivations and Background

Chapter 2

Migration to Microservices: an Industrial Perspective

This Chapter, provides an overview of how companies perceive the migration to microservices, which is of critical importance given the industrial nature of this Thesis. To this end, we analyze five survey papers found in the scientific literature. By reviewing these papers, we aim to provide a comprehensive overview of the current state of industry perceptions towards microservices migration. Understanding these perceptions is essential for developing effective migration strategies and promoting the adoption of microservices in industry.

2.1 State of the Art Analysis: the Protocol

The migration to microservices architecture has gained significant attention in recent years due to its potential benefits for organizations. The objective of this Chapter is to analyze the current state of the art in microservices migration in companies, with a focus on its impact on the organization, technologies, and other relevant aspects. In order to provide a comprehensive understanding and emphasize the importance of careful migration planning, the chosen approach involves analyzing existing surveys with varying objectives. The inclusion of various surveys allows us to explore different dimensions of microservices migration and draw insights from a range of perspectives. While each survey may have had its own specific research questions, by synthesizing their findings, we can establish a broader context and capture a more comprehensive overview of the topic. By analyzing these surveys, we can identify common challenges, best practices, and lessons learned from real-world experiences. This comprehensive examination enables us to provide valuable insights into the complexities and implications associated with microservices adoption, ultimately emphasizing the need for a well-executed and carefully planned migration process.

The rationale behind the search strategy employed for the analysis of the state of the art presented in this Chapter is to identify survey papers of high quality that are specifically relevant to the industry's migration to microservices. The decision to restrict the research to survey papers is motivated by several factors:

- Comprehensive overview: Survey papers provide a holistic view of the field, incorporating insights from various sources and existing literature reviews.
- Consolidated findings: By analyzing survey papers, common trends, challenges, and best practices in microservices migration can be identified through the synthesis of multiple studies.
- Industry relevance: Focusing on survey papers related to industry practices ensures that the research aligns with real-world experiences and is directly applicable to practitioners and decision-makers.
- Time efficiency: Analyzing survey papers is a time-efficient approach, as they summarize and analyze existing research, enabling the gathering of substantial information within a reasonable timeframe.

Although the approach used in this Chapter is not a systematic review, certain strategies were employed to refine the research and identify papers closely aligned with the research objectives. For instance, the search strategy adopted drew inspiration from systematic reviews, incorporating elements such as the formulation of search strings, library selection, and the application of inclusion/exclusion criteria. To facilitate the organization and synthesis of the results, research questions were formulated. These research questions were derived from the research questions of the individual studies included, serving as a framework to categorize and analyze the findings. By aligning the overarching research questions with the specific research questions of the included studies, the results can be presented in a cohesive and meaningful manner, enabling a comprehensive understanding of the implications and insights derived from the analyzed literature. Figure 2.1 provides the four-step roadmap followed to perform the informal review presented in this Chapter.

2.2 Search Strategy

The first stage of the review process relates to the search strategy. This is a crucial to identify and collect relevant literature that involves selecting appropriate keywords, search operators, and techniques to search databases, journals, and other sources of information. Thus, the search strategy adopted consists of three main tasks namely the creation of the search string, the library selection and inclusion criteria definition. Those three task will be explored in details in the next subsections.

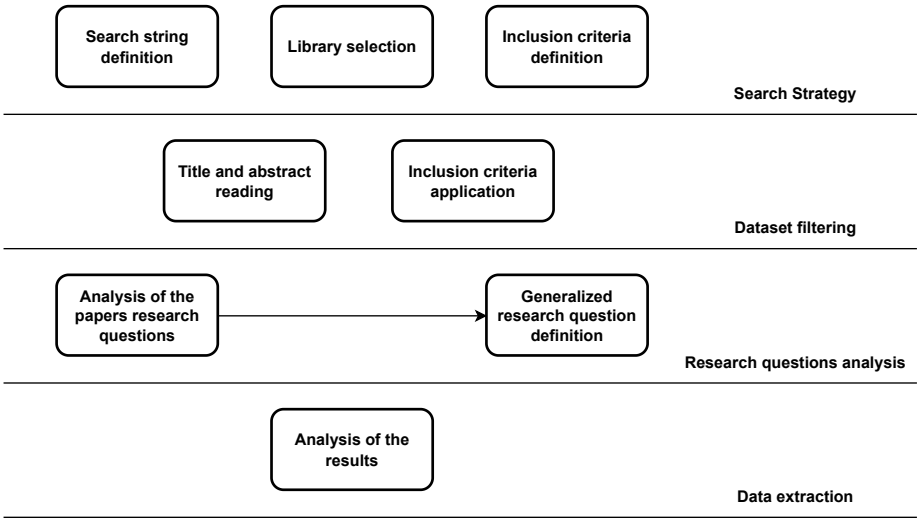

Figure 2.1: Informal Review Procedure

2.2.1 Search String Definition

As already mentioned, the adopted search methodology does not strictly adhere to the conventional systematic review protocol. Indeed, in this Chapter, the objective is not to produce a systematic literature review but rather to gain an overall understanding of the migration to microservices in the industry while analyzing a subset of surveys on the topic. Consequently, the construction of the search string differs from a typical systematic review process as it aims to capture important concepts and terms related to microservices migration in industry captured by surveys studies, rather than being solely focused on identifying works that directly address specific *research questions* (*RQs*). Relevant keywords include *microservices migration*, *microservices adoption*, *survey*, *industry*, *challenges*, *benefits*, and *strategies*. Among the highlighted keywords, we decided to include only the most generic subset to capture as many relevant results as possible. In fact, a more specific set of keywords could potentially exclude important literature that uses different terminology. Therefore, we have included broad terms such as *microservices*, *migration*, *survey*, and *industry*. While more specific terms like *challenges*, *benefits*, and *strategies* may also be relevant, we opted for a more inclusive approach to ensure a comprehensive search. Therefore, the following string has been developed:

*"migration" AND "micro*service*" AND "industry" AND "survey"*

2.2.2 Library Selection

In the context of our search strategy, choosing the appropriate digital libraries to retrieve relevant papers is a crucial step. Although there are several libraries available, such as ACM Digital Library, IEEE Xplore, Scopus, and others, we decided to use only one digital library: Google Scholar. The reason for selecting this digital library is

its capability to access potential grey literature, including non-peer-reviewed articles, which are not typically available in traditional scientific databases. Furthermore, Google Scholar indexes a wide range of sources, including academic publications, institutional repositories, and industry reports, making it a valuable tool to search for survey papers related to microservices migration in industry. We decided to conduct the research using "All Metadata" as the search scope to ensure comprehensive coverage.

2.2.3 Criteria Definition

To broaden our search and capture the relevant papers while maintaining quality standards, we chose to define only inclusion criteria for our search strategy. These criteria include: i) the paper must be written in English, ii) it must investigate the microservices adoption, and iii) it must have been published between 2015 and 2023. We opted to set only one exclusion criteria to exclude any papers not reporting real-world company experiences.

2.3 Research Quesitons

Upon retrieving 6150 results using the search strategy, an analysis of the titles and abstracts of the papers was conducted. In line with the applied inclusion criteria, a careful selection process was undertaken, resulting in the identification of five relevant works. The decision to focus on these five papers was justified by their direct relevance to our research topic and their adherence to the predetermined inclusion criteria. Considering the large number of retrieved results, it was necessary to implement a selection strategy to manage the volume of literature. In order to ensure a reasonable and manageable workload, it was decided to analyze only the first five pages of search results. This approach strikes a balance between conducting a thorough review and the practical limitations of time and resources. By examining the titles and abstracts of the papers within this subset, we were able to identify the most pertinent works that aligned closely with our research topic and met the established criteria for inclusion.

From the selected five papers, we identified *research question* (RQ)s that were either explicitly stated in the papers or could be inferred from the content. These RQs were then used to create three broad RQs that would guide our study. The decision to create broad RQs was made to ensure that we were able to capture a comprehensive view of the state of practice in microservices adoption in industry. By identifying common themes across the five selected papers and using them to create broad RQs, we aim to gain a deeper understanding of the challenges, benefits, and strategies associated with microservices adoption in industry. Additionally, by using broad RQs, we can identify gaps in the literature and areas for further research, which will help us to contribute to the existing body of knowledge on microservices adoption in industry.

2.3.1 Selected Studies

For each paper included in our study, we highlighted the objectives, *RQ*s, and population. By examining these aspects, we can gain a better understanding of the focus and scope of each paper and analyse how it fits into our overall research on microservices migration in industry. The objectives and *RQ*s help us identify the main goals and areas of inquiry that the authors were investigating, while the population helps us understand the context and scope of the study. Ultimately, this information will help us synthesize the findings from each paper and draw meaningful conclusions about the adoption of microservices in industry.

Paper 1: ***Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation.*** **[19]** The authors in this work interviewed 11 practitioners who adopted a MSA style at least two years before the survey to understand their motivations for adopting microservices. The interviewees differed in roles (software architects, project managers, senior developers, agile coaches, and CEOs), organization domains (banks, software service companies, migration consultancies, public administrations, and telecommunications), and types of migration (completed and ongoing). While the paper does not explicitly mention specific *RQ*s for the survey, we have deduced the following *RQ*s based on the study's focus on motivations for and experiences with MSA. The following are three possible *RQ*s related to the study:

- P1.RQ1: Why did companies choose to migrate to MSA?
- P1.RQ2: What are the common issues encountered during migration to MSA?
- P1.RQ3: What are the benefits that companies experienced migrating to MSA?
- P1.RQ4: What are the common steps in the migration process to MSA?

Paper 2: ***Migrating towards Microservices Architectures: an Industrial Survey.*** **[37]** The study investigates the migration practices towards MSA by studying the activities and challenges faced by industrial practitioners. The research methodology involved conducting 5 exploratory interviews and an online survey questionnaire among 18 practitioners across 16 different IT companies. The main contributions of the paper are a survey of 18 practitioners, an analysis of the collected data, a discussion of the obtained results, and the study replication package. The analysis of the collected data discusses practitioners' perspectives on migration activities and challenges in the three phases of architecture recovery, transformation, and implementation, and potentially relevant research directions. The paper aims to answer the following *RQ*s:

- P2.RQ1: What are the activities carried out by practitioners when migrating towards a MSA?

- P2.RQ2: What are the challenges faced by practitioners when migrating towards a MSA?

Paper3: ***Microservices Migration in Industry: Intentions, Strategies, and Challenges.*** **[38]** This work discusses the limited impact of academic research on Microservices adoption and the strong industry interest in migrating legacy systems. To address this gap, the authors conducted 16 in-depth interviews with software professionals based in Germany from 10 different companies. The goal was to reveal intentions and strategies for such migrations as well as challenges that companies were faced with. The interviews covered other topics, such as used technologies and practices to assure the evolvability of Microservices, which are presented in two additional publications. The study provides empirical and industry-focused research and aims to provide rationales and successfully applied practices for Microservices migrations. In this context, researcher generated the following *RQ*s:

- P3.RQ1: What are intentions for migrating existing systems to microservices?
- P3.RQ2: Which Microservices migration strategies and decomposition approaches do companies apply?
- P3.RQ3: What are the major technical and organizational challenges during a microservices migration?

Paper 4: ***Are we speaking the industry language? The practice and literature of modernizing legacy systems with microservices.*** **[39]** The paper aims to investigate the use of microservices in the industry for modernizing legacy systems and whether there is alignment between industry and academic practices. The authors designed a survey based on an 8-step modernization process roadmap and received responses from 56 software companies, of which 35 (63.6%) adopt the MSA in their legacy systems. The survey results provide a comprehensive characterization of industrial practices and reveal that they are aligned with existing research, reinforcing the use of the modernization process roadmap. The paper also identifies research opportunities in database aspects as an input to the modernization process and a factor to reduce microservices coupling at runtime. This survey was conceived to answer the main *RQ*: Are the practice and literature of the modernization of legacy systems with microservices aligned? To this purpose, authors defined the following three sub-*RQ*s:

- P4.RQ1. Why do companies migrate monolithic legacy systems to microservices?
- P4.RQ2. How do companies perform the migration of monolithic legacy systems to microservices?
- P4.RQ3. What are the aspects of data persistence considered in the modernization of legacy systems with microservices?

Paper 5: ***Revisiting the practices and pains of MSA in reality: An industrial inquiry.*** **[40]** This paper discusses the challenges of designing software architecture and how microservices can improve current practices. The author conducted a study through interviews with practitioners from 20 software companies to investigate the gaps between ideal visions and real industrial practices in microservices and the expenses they bring. The study's main contributions are identifying the gaps between the best-known characteristics and real practices in microservices, proposing a unified overview map of general practices and pains, and condensing five decisions that practitioners should make. The author also discusses potential future research directions in this area. To characterize the practices and challenges related to the migration and adoption towards microservices, the authors defined the following *RQ*s:

- P5.RQ1: What are the gaps between visions and the reality of microservices?
- P5.RQ2: What are the pains of implementing microservices in practice?

2.3.2 *RQ*s Generalizzation

To make the results of the five highlighted papers comparable, we clustered their *RQ*s into three more general *RQ*s. This approach allowed us to identify common themes and trends across the papers, and to explore the *RQ*s from a broader perspective. By creating these more general *RQ*s, we were able to conduct a more comprehensive analysis of the literature and gain a deeper understanding of the key issues and challenges related to the topic. Overall, this clustering process helped us to synthesize the findings of the five papers and draw more robust conclusions about the state of research in the field. The three broader categories selected are: reasons and motivations for migration, common challenges faced during migration, and strategies and approaches adopted during migration. For each paper, Table 2.1 maps the *RQ*s reported (or deducted) in the paper with the generalized form. In the following we present the generated *RQ*s:

- RQ1: What are the reasons and motivations for companies to migrate from monolithic legacy systems to microservices? This question aims to understand the main drivers that lead companies to adopt microservices.
- RQ2: What are the common challenges faced during the migration process to a MSA? The goal of this question is to explore the challenges that companies face when migrating to microservices.
- RQ3: What are the strategies and approaches adopted by companies during microservices migration? The aim of this question seeks to identify the best practices for designing and implementing microservices.

RQ	Paper 1	Paper 2	Paper 3	Paper 4	Paper 5
RQ1	P1.RQ1	-	P3.RQ1	P4.RQ1	P5.RQ1
RQ2	P1.RQ2	P2.RQ2	P3.RQ2	-	P5.RQ2
RQ3	P1.RQ4	P2.RQ1	P3.RQ2	P4.RQ2	-

Table 2.1: Mapping between *RQ*s

The provided *RQ*s will be discussed within their result in the followig section.

2.4 Reported Results

In this section, we will delve into the results of each of the five papers to address the three general *RQ*s formulated. By analyzing the results of each paper in the context of these three questions, we hope to gain a comprehensive understanding of the current state of microservices adoption in industry.

2.4.1 RQ1: What are the reasons and motivations for companies to migrate from monolithic legacy systems to microservices?

The aim of RQ1 is to investigate and understand the motivations and reasons that drive companies to migrate from monolithic legacy systems to MSA. This *RQ* seeks to identify the factors that encourage companies to adopt microservices and the specific benefits that they expect to gain from the migration. By answering this question, researchers can provide valuable insights into the practical applications of microservices and the real-world problems that they address. This information can be useful for companies that are considering adopting microservices and for researchers who are interested in studying the adoption and implementation of MSA.

Results from Paper 1 [19] The paper discusses the motivations driving the adoption of MSAs. The primary reasons for adoption are software maintenance, scalability, delegation of team responsibilities, and DevOps support. Other motivations reported were the ease of technology experimentation, fault tolerance, and separation of software responsibilities. Microservices are seen as more maintainable than monolithic systems because they enable developers to make changes and test their service independently, increasing code understandability. In terms of scalability, scaling microservices is easier than scaling monoliths as each microservice can be deployed on different servers with different levels of performance, and a bottleneck in one microservice can be containerized and executed across multiple hosts in parallel. The delegation of clear and independent responsibilities among teams allows splitting large project teams into several small and more efficient teams. Moreover, microservices are cool and popular, and companies are adopting them to avoid being out of the market because of a wrong technology choice.

Finally, microservices are responsible for one single task within well-defined boundaries and are self-contained, greatly simplifying development.

Results from Paper 3 [41] The main reasons for replacing the old system were related to the lack of maintainability, with symptoms such as loss of overview, high cost of changes, and potential side effects. Additionally, there were issues with operability, such as missing traceability, long startup times, downtime during updates, and difficulties in applying updates. Participants also noted problems with performance and the inability to address current functional requirements due to deprecated technologies or existing design limitations. The need to bring new features to market quickly was another driver for change, especially for one system that had a five-month lead time for major changes. The decision to adopt microservices was driven by scalability requirements, as well as the benefits for development teams and the possibility of a multi-vendor strategy. The goal of achieving better manageability and maintainability through smaller, more manageable units was highlighted in several cases.

Results from Paper 4 [39] The transition towards a MSA is driven by various factors that originate from operational, technical, and organizational aspects. It has been observed that the top three driving forces are consistent with the ones highlighted in the modernization process roadmap literature. These include the facilitation of maintenance and evolution, improved scalability, and the ability to deploy independently and automatically. Notably, more than 70% of the organizations cited these three driving forces as the most common ones.

Results from Paper 5 [40] The text discusses the motivations of organizations for adopting microservices. According to a survey, continuous delivery and DevOps are the most common reasons cited by 60% of interviewees. Microservices allow for independent development and deployment, easy scaling and upgrading, and can facilitate the application of DevOps, reducing lead time. Other motivations include reduction of independent organization, technology heterogeneity, and independent testing. Motivations vary by industry, with extensibility and independent organization being important in software/IT, scalability and independent organization in IT service providers, and continuous delivery and DevOps in telecommunications, e-commerce, banking, and finance industries.

Summary Various papers indicate that the primary reasons for adopting microservices include software maintenance, scalability, delegation of team responsibilities, DevOps support, ease of technology experimentation, fault tolerance, separation of software responsibilities, and quick feature deployment. Microservices offer greater maintainability and scalability than monolithic systems by allowing developers to make

changes and test their services independently, leading to increased code understandability. The decision to adopt microservices is often driven by scalability requirements and benefits for development teams, including the possibility of a multi-vendor strategy. Motivations for adopting microservices vary by industry, with extensibility and independent organization being important in software/IT, scalability and independent organization in IT service providers, and continuous delivery and DevOps in telecommunications, e-commerce, banking, and finance industries. Table 2.2 presents the mapping between each motivation for migration and the research papers in which they are discussed.

Motivation	**Papers**
Maintenance and Evolution	Paper 1, Paper 4
Scalability	Paper 1, Paper 3, Paper 4, Paper 5
Independent Development and Deployment	Paper 3, Paper 4, Paper 5
DevOps and Continuous Delivery	Paper 1, Paper 5
Application of Independent Testing	Paper 5
Reduction of Independent Organization	Paper 5
Technology Heterogeneity	Paper 5

Table 2.2: Mapping between Motivation for Migration and Research Papers

2.4.2 RQ2: What are the common challenges faced during the migration process to a MSA?

The aim of *RQ* RQ2 is to identify the common challenges that organizations face while migrating from a monolithic legacy system to a MSA. The focus is on understanding the difficulties encountered during the migration process, such as technical challenges and issues with the adoption of new tools and technologies, as well as non-technical challenges, such as the need for cultural change and organizational restructuring. The goal is to gain insights into the factors that impede the successful adoption of microservices, which can help organizations to prepare better and mitigate risks during the migration process.

Results from Paper 1 [19] The paper discusses the various challenges and issues that can arise when adopting a microservices-based architectural style. While there are certainly benefits to this approach, such as improved maintenance, increased scalability, and better performance, there are also several challenges that practitioners may encounter. One major issue that practitioners face is decoupling from the monolithic system. Database migration and data splitting are other issues that need to be addressed carefully. Some participants recommended splitting the data in existing databases so that each microservice accesses its private database. Communication among services

is also a crucial issue, and every microservice needs to communicate, which can add complexity to implementation, along with possible network-related issues. Estimating the development time for a microservices-based system is considered less accurate than estimating a monolithic system. The interviewees reported an effort overhead of nearly 20% more compared to the effort required for developing a monolithic solution. However, the benefits of increased maintainability and scalability highly compensate for the extra effort. Adopting microservices also requires adopting a DevOps infrastructure, which requires a lot of effort and needs to be taken into account in addition to the development effort. Existing libraries require more effort for conversion and cannot be simply reused. Microservices-based architectural styles also require an orchestration layer, which adds complexity to the system and needs to be developed reliably. People's minds are another issue, as changes in existing architectures are generally an issue for several developers. They may consider the legacy system as their creation and be reluctant to accept such an important change to the software they wrote. Finally, ROI is perceived as an important issue because of the increased effort required for adopting microservices. However, in some cases, practitioners reported that migration was the only choice, independent of the costs, because they were forced to migrate due to the lack of maintainability and the impossibility to scale their legacy systems.

Results from Paper 2 [37] The paper discusses challenges faced during the process of reverse engineering, architecture transformation, and forward engineering. In reverse engineering, participants found releasing new features and maintaining and testing the pre-existing system to be challenging. Technical challenges related to side effects, low developer productivity, and lack of proper documentation were also identified. Convincing business/management about the need for migration was another challenge. Architecture transformation posed challenges related to high coupling among parts of the pre-existing system and identifying service boundaries. Participants also found configuration and setup of automation support for testing to be challenging. Forward engineering presented challenges in setting up the initial infrastructure for microservices, adapting to a different developer mindset, and knowledge sharing. Distributed aspects related to monitoring, logging, and debugging were also reported as challenging. Testing the new system and getting the first team to work together were identified as additional challenges. Some participants reported specific challenges, such as the impact of deploying new microservices on the system, working with the business for testing, and needing infrastructure for container management and service discovery. Overall, the challenges related to technical aspects, communication, business-IT alignment, and management.

Results from Paper 3 [38] The main challenges reported by participants during the transition to a MSA were finding the appropriate decomposition approach and a lack of expertise in the field, which were reported equally often. The issue was to

become familiar with the various technologies and tools in a timely manner. The difficulty in recruiting skilled personnel added to the challenge. The establishment of DevOps practices, such as build and test automation, was necessary to fully benefit from Microservices. However, many organizations faced challenges or postponed such activities. While most participants reported a decent degree of automation, fully automated continuous deployment existed in only three cases. Integrating the services and interoperability with third-party software or the existing monolith was a challenge for three systems. The size, complexity, or outdated technologies of the legacy system were obstacles for some organizations. Assuring the system's security was also reported as a challenge for several systems. Additionally, building a resilient architecture with fault-tolerant services was seen as challenging for some participants. Some organizations needed to transition from traditional process models towards agile methodologies, which involved a mindset change from Waterfall to Agile. Collaboration between autonomous teams was seen as challenging in several cases, causing constant frictions in collaboration during the initial months of restructuring. Companies tried to mitigate this by regular coaching.

Results from Paper 5 [40] The main pain points mentioned by interviewees are outlined in this paper. The most common challenge among those in the software/IT and Telecommunication domains is fault localization complexity. The IT service provider domain identifies fault localization complexity, monitoring complexity, and performance reduction as the main issues. In the e-commerce domain, monitoring complexity is the main concern. Finally, with the exception of monitoring complexity and performance reduction, all other pain points in the figure are associated with the Bank and Finance domain.

Summary The challenges can categorized into technical and non-technical issues. Technical challenges included decoupling from the monolithic system, communication among services, database migration, and finding the appropriate decomposition approach. Non-technical challenges included cultural change and organizational restructuring. Other challenges identified in various papers included resistance to change among developers, lack of expertise, system security, and collaboration between autonomous teams. The challenges varied across different domains, with fault localization complexity being the most common challenge among those in the software/IT and telecommunication domains, monitoring complexity being a primary issue in the e-commerce domain, and other challenges associated with the banking and finance domain. To present the mapping between challenges and related papers in a concise and organized manner, the Table 2.4 was created with unique challenges as rows and related papers as columns. The table lists the challenges and corresponding papers in a clear and readable format, with the number of papers related to each challenge ranging from

one to five. By creating this mapping table, researchers and practitioners can easily identify and address the challenges associated with migrating from monolithic legacy systems to MSAs.

Challenges	**Related Papers**
Technical challenges during migration process	Paper 1, Paper 2, Paper 3, Paper 5
Non-technical challenges such as cultural change	Paper 1, Paper 3
Decoupling from monolithic system	Paper 1, Paper 2
Database migration and data splitting	Paper 1
Communication among services	Paper 1, Paper 2
Resistance to change among developers	Paper 1
Challenges during reverse engineering (coupling)	Paper 2
Challenges in finding appropriate decomposition approach	Paper 3
Lack of expertise in the field	Paper 3
Difficulty in becoming familiar with new technologies	Paper 3
Assuring system security	Paper 3
Building resilient architecture with fault-tolerant services	Paper 3
Transitioning from traditional process models to agile methodologies	Paper 3
Fault localization complexity	Paper 5
Monitoring complexity	Paper 2, Paper 5
Performance reduction	Paper 5

Table 2.3: Mapping between Challenges in Migration and Research Papers

2.4.3 RQ3: What are the strategies and approaches adopted by companies during microservices migration?

The aim of this *RQ* is to identify and analyze the various strategies and approaches adopted by companies during the migration process from monolithic legacy systems to MSA. The focus is on understanding the practical implementation of the migration process and identifying the best practices, tools, and frameworks used by companies to address the challenges and risks associated with the migration. By answering this *RQ*, the study aims to provide insights into the real-world scenarios of microservices

migration and help organizations to plan their migration process effectively.

Results from Paper 1 [19] The article discusses three different processes for migrating from a monolithic system to a microservices-based one. All three processes involve analyzing the system structure, defining the system architecture, prioritizing feature or service development, and carrying out coding and testing. The main difference between the processes is in the prioritization of feature/service development. The first two processes involve reimplementing the system from scratch, while the third approach involves implementing new features as microservices and gradually eliminating the existing system. The main benefit of the first two processes is the complete rearchitecture of the whole system, while the main benefit of the third process is the lower migration cost. The main issue with the third process is the longer time needed to completely abandon the legacy system once all new features have been completely replaced by new ones.

Results from Paper 2 [37] The study describes how practitioners approach the process of reverse engineering and architecture transformation for microservices. The study reveals that participants rely on low-level sources of information such as source code and test suites, as well as higher-level sources such as textual and architectural documents, data models, and diagrams. Knowledge about the system also resides in the people within the organization. When it comes to architecture transformation, the major activities involved include domain decomposition, service identification, application of domain-driven design practices, and system decomposition. Participants also emphasized the need to experiment with microservices through proof-of-concept services or Minimum Viable Products. The main driver of migration was identified as the functionalities, followed by factors such as customer needs and processes, business-IT alignment, and trade-offs between costs and benefits. During forward engineering, participants either started by adding new functionalities as independent microservices or by reimplementing existing functionalities in the pre-existing system as microservices. The initial set of functionalities to migrate from the pre-existing system was identified based on factors such as less dependencies, less use by users, and importance to customers. Phased adoption was the most common method used for adopting the new system in production, with the strangler pattern being a popular approach. Parallel adoption and big bang adoption were less common.

Results from Paper 3 [38] In the study, the most common migration strategy for the systems investigated was rewriting the existing application (nine cases) while also extending its functionality (seven cases). The Strangler pattern [26] was employed in seven cases to gradually replace the existing system with Microservices. However, some practitionairs encountered challenges with splitting up complex systems. The time

frames for the migrations ranged from 1.5 to over 3 years, with some projects starting with a large team which was later reduced in size. For larger projects, multiple contractors were involved or the migration passed through multiple phases until transitioning into a continuous product development mode. Decomposition, which involves dividing a system or problem space into smaller parts, was used by seven participants using a functional decomposition approach, as per Microservices design principles [14]. Domain-Driven Design was used explicitly in only three systems, while other approaches were also used, often described as the architect's task or the result of architecture group meetings. In some cases, the existing system's structure served as a basis for decomposition. However, the Wrong Cuts antipattern [42] was identified in three cases, and the Shared Persistency and Inappropriate Service Intimacy antipatterns were observed as a result of inappropriate cuts. While some participants appreciated the flexibility that comes with fine-grained services, many consciously aimed for more coarse-grained services due to difficulties in finding the right service cut and a desire to avoid complex macro architectures. Microservices-specific refactoring approaches that offer tool support were not considered by any of the participants, with some even believing that it cannot be automated.

Results from Paper 4 [39] According to the survey result, most companies have completed all the activities outlined in the modernization process roadmap that are: analysis ot the driving forces, legacy systems understanding, legacy system decomposition, architecture definition, modernization execution, microservices integration with the legacy system, microservices verification and validation and, microservices monitoring. About 50% of companies opted for an incremental migration approach to decompose their legacy systems based on business capabilities. Companies mainly use scalability (85.7%), requirements (45.7%), and reusability (45.7%) as criteria to identify microservice candidates. Nearly 65% of companies use more than one programming language for their projects. Unit and integration testing are widely used by at least 77% of companies. Only a small percentage (22.9%) of companies address code duplication, and there is no standard policy for customizing source code. The communication interface used to integrate microservices is mostly REST API. Almost all companies (97.1%) use cloud infrastructure services, primarily Amazon Web Services. Spring Boot, Jenkins, Docker, Kubernetes, and ELK are the most popular technologies used for infrastructure configuration, continuous integration, containerization, microservice monitoring, and logging management. The survey results align with the literature on modernizing legacy systems with microservices, as both emphasize the same driving forces, migration strategy, and criteria. The use of at least two programming languages in a project reflects the technology flexibility provided by microservices.

Summary The papers reports on the strategies and approaches companies use when migrating from monolithic legacy systems to MSA. The papers discuss different processes for migrating, including re-implementing the system from scratch or implementing new features gradually. They also highlight the importance of domain decomposition, service identification, and proof-of-concept services. The papers mention several antipatterns that can arise during the migration process. The results of a survey indicate that most companies have completed the modernization process roadmap, and about 50% opted for an incremental migration approach based on business capabilities. Companies use scalability, requirements, and reusability as criteria to identify microservice candidates, and almost all use cloud infrastructure services and REST API communication interfaces. Spring Boot, Jenkins, Docker, Kubernetes, and ELK are popular technologies used for infrastructure configuration, continuous integration, containerization, microservice monitoring, and logging management.

Migration Phase	**Related Papers**
Analyzing the system structure	Paper 1, Paper 2, Paper 4
Defining the system architecture	Paper 1, Paper 4
Prioritizing feature or service development	Paper 1
Domain decomposition	Paper 2, Paper 3
Application of domain-driven design practices	Paper 2, Paper 3
System decomposition (or service identification)	Paper 2, Paper 3, Paper 4
Incremental migration approach	Paper 3, Paper 4
Microservices monitoring	Paper 4

Table 2.4: Mapping between Challenges in Migration and Research Papers

2.5 Discussion

Migrating from monolithic legacy systems to MSA is becoming a popular choice for many companies due to various reasons. One of the main drivers behind this shift is the need for improved quality of the system. Companies strive to achieve better maintainability, scalability, and performance, which are essential for meeting the evolving needs of their customers. Even if these quality objectives are not the primary reasons for the migration, they still represent the ultimate goals that companies aim to achieve. As a result, it is crucial to consider the qualities of the system that one wants to obtain by migrating to a MSA. This consideration ensures that the migration process is aligned with the overall objectives of the organization and meets the needs of the end-users. By focusing on quality attributes such as maintainability, scalability, and performance, companies can ensure that their systems remain relevant and competitive in today's fast-paced business environment. With a well-planned migration strategy that em-

phasizes these objectives, companies can achieve significant benefits such as increased agility, reduced costs, and improved customer satisfaction. Therefore, it is essential to carefully evaluate the system's qualities and ensure that they are adequately addressed during the migration process to MSA.

Although microservices offer many benefits such as increased agility, scalability, and better alignment with business requirements, the migration process is not without its challenges. Companies need to carefully plan and execute the migration, considering factors such as system analysis, defining the system architecture, domain decomposition, and application of domain-driven design practices. Additionally, the migration process requires a high degree of coordination and communication among development teams, infrastructure teams, and stakeholders. Antipatterns can arise during the migration process, and companies need to be aware of these to avoid potential problems. Thus, it is crucial to avoid bad practices that could have negative effects on the resulting system. Therefore, a well-planned migration process can prevent the emergence of antipatterns. Addressing these challenges can be complex and time-consuming, and requires careful planning and execution. However, by carefully designing the migration process and following best practices, it is possible to mitigate these challenges and achieve the benefits of a MSA.

Companies follow different processes when migrating to microservices, depending on factors such as the size and complexity of the existing system, available resources, and business requirements. Two primary patterns for migration have been identified: complete rewriting and strangler pattern. In complete rewriting, the entire system is re-implemented from scratch in a MSA. This approach is more suitable for small or medium-sized systems that are not very complex. On the other hand, the strangler pattern involves gradually replacing parts of the legacy system with microservices. This approach is more suitable for larger, more complex systems that are critical to business operations. Despite the different processes, there are three common phases in the migration process: legacy system comprehension, microservices identification, and microservices assessment and monitoring. During the legacy system comprehension phase, companies analyze the existing system and its components to understand its structure and dependencies. In the microservices identification phase, companies identify potential microservices by applying domain-driven design practices, such as domain decomposition and bounded context identification. Finally, in the microservices assessment and monitoring phase, companies assess the quality of microservices, evaluate their performance, and monitor their behavior in the production environment.

2.6 Conclusion

This Chapter provided an overview of the challenges and benefits involved in adopting microservices, which is of critical importance given the industrial nature of this Thesis.

Through the analysis of five survey papers, a comprehensive view of the current state of industry perceptions towards microservices migration is provided. The investigation process employed to explore the current state of practice in microservices migration involved a search strategy to identify and collect relevant literature. After analyzing the five selected papers, three broad RQs were formulated to guide the study, which were grouped into categories including reasons and motivations for migration, common challenges faced during migration, benefits of migration, and strategies and approaches adopted during migration. Migrating from monolithic legacy systems to MSA is becoming a popular choice for many companies due to various reasons, with improved quality of the system being the main driver behind this shift. The careful evaluation of the system's qualities is essential to ensure that they are adequately addressed during the migration process to MSA. Although microservices offer many benefits such as increased agility, scalability, and better alignment with business requirements, the migration process is not without its challenges, including cultural, organizational, and technological challenges. The results of this study provide valuable insights into the challenges and benefits of microservices adoption, the strategies and best practices employed by companies, and the perceptions and experiences of industry practitioners. These insights can be used to develop effective migration strategies and promote the adoption of microservices in industry.

Chapter 3

State-of-the-art in Quality-Driven Migration to Microservices

In [20], Newman describes different patterns allowing to migrate from monolithic to MSA. Since each monolith is different, choosing the right approach for migrating to microservices is a challenging task. Thus, formalizing a single approach to migration is an impossible chore to achieve. The adopted technique must consider different parameters as the context in which the system lives and the objectives to be achieved with migration. In this context, it is essential not only to focus on the system's functionalities, denying its dependencies and responsibilities but, also to consider all those software qualities on which migration can impact. This Chapter aims to understand how researchers consider software qualities during the migration to microservices. Thus, we investigate possible quality-driven migration approaches by conducting a *Systematic Literature Review* (*SLR*) on the topic. An *SLR* is a research method that involves a comprehensive and structured approach to identifying, evaluating, and synthesizing all relevant literature on a particular *RQ* or topic. It is used to provide an overview of the current state of knowledge on a topic and inform policy or practice. The process involves several key steps, including defining the *RQ*, searching for relevant studies, screening and selecting studies, assessing study quality, and synthesizing the findings. The *SLR* reported in this Chapter follows the rigorous research method presented by Kitchenham et al. [43]. The research method applied to systematically review the scientific literature follows the three-step process proposed by Kitchenham et al. in [43]: *Planning*, *Conducting* and *Reporting*. The main activities carried out to perform this study are reported in Table 3.1.

SLR Phase	Activities
Planning	*RQ*s definition
	Search Strategy Definition
	Data Extraction Plan
Conducting	Study selection
	Data extraction
Reporting	Results discussions

Table 3.1: *SLR*: Steps and Activities.

3.1 Related Work

The subject of migration to microservices has garnered significant research interest because MSA offers various benefits, such as flexibility, scalability, and better maintainability of software systems. This architecture breaks down an application into smaller, more manageable services, which can be independently developed, deployed, and maintained. However, migrating to MSA can be a challenging task due to the need to understand the current monolithic system, identify services, and package them effectively. Therefore, several surveys, systematic reviews, and mapping studies have been conducted to investigate the best practices, challenges, and solutions for migrating to MSA, making this an active area of research.

In this section, we compare the systematic study presented in this Chapter with some of the significant works found in the literature. The authors of [44] aim to address the various challenges associated with MSA, including the migration problem. After explaining the reasons for adopting this architecture, the authors focus on the quality aspects that impact the migration process, which they refer to as factors. These factors include performance, interdependency, costs, scalability, reusability, and continuous development. Additionally, the authors attempted to compile all the techniques used to migrate a system to microservices, but were unsuccessful in their efforts. The authors of [45] seek to examine the significance of variability in the migration process, specifically regarding its mechanisms to support and guide the migration of configurable systems to a MSA. They also aim to explore how variability manifests after migrating to a MSA. The survey conducted by the authors found that 50% of the participants addressed variability with at least one system module from which microservices were extracted. Furthermore, 69% of the participants who migrated systems considered variability a useful or very useful criterion during the migration process. The authors highlight that microservices are typically extracted through an incremental process. The authors in [41] suggest that conventional academic techniques often utilize coupling and cohesion as the primary criteria for microservice extraction. However, as limited knowledge exists on criteria useful to practitioners, they conducted an exploratory online survey to deter-

mine the relative usefulness of seven possible decision-making criteria for microservice extraction. These criteria include coupling, cohesion, communication overhead, reuse potential, requirement impact, and visual models. The survey results show that participants consider criteria related to modularity and requirement impact as relevant, while the other three criteria are considered moderate. Furthermore, the usefulness of the criteria varies depending on the context. The authors of [46] examine the challenges and techniques involved in transitioning from a monolithic to a MSA. They analyze five different migration methods, highlighting their advantages and disadvantages. The main conclusion drawn from this study is that there is no one-size-fits-all approach to migrating from a monolithic to a MSA due to the uniqueness of each application. Various factors such as the application's objectives, complexity, technologies, team size, and skill set must be considered when planning and executing a migration strategy. A rapid review of the migration from monolithic to MSA was presented in [47]. The study analyzed techniques proposed in the literature, the systems to which those techniques were applied, and the type of validation performed. The authors identified three main approaches: model-driven, static analysis, and dynamic analysis. Additionally, they highlighted the challenges associated with migrating from monolithic to MSA and reported evidence of successful migrations from web-based applications to microservices. The validation of the approach was conducted through case studies, which generally involved comparing the performance of the existing monolithic version of the system with the microservices-based version. The study also revealed challenges related to database migration and migration planning, including resource management, developer skills, and implementing the new organization of the company. In [48], a systematic mapping study was conducted to identify the issues and difficulties associated with the migration from monolithic to MSA. The authors identified several challenges, including the requirement for appropriate tools to facilitate migration, the need for team restructuring, the necessity for incremental migration processes, and the challenge of identifying and designing microservices. Additionally, they recognized the importance of ensuring data consistency when transitioning from a single database to multiple databases. Our *SLR* differs from previous studies in several ways:

- Instead of exclusively emphasizing migration techniques, we synchronize each technique with its respective phase within the migration process.
- Our study explores quality factors relevant to all stages of the migration process.
- In contrast to other studies, our primary aim is to determine whether enhancing software quality is a common goal across migration-related literature.

3.2 Planning the Review

The objective of the *Planning* stage in the *SLR* presented in this Thesis is to establish the link between migration strategies to microservices and software qualities. To achieve this, a set of *RQ*s were formulated, which were used to generate the research string. Furthermore, the digital libraries to be considered were defined, along with the inclusion and exclusion criteria. The activities carried out during the planning phase are detailed in the following section. Figure 3.1 shows the activities carried out to perform the planning of this *SLR*.

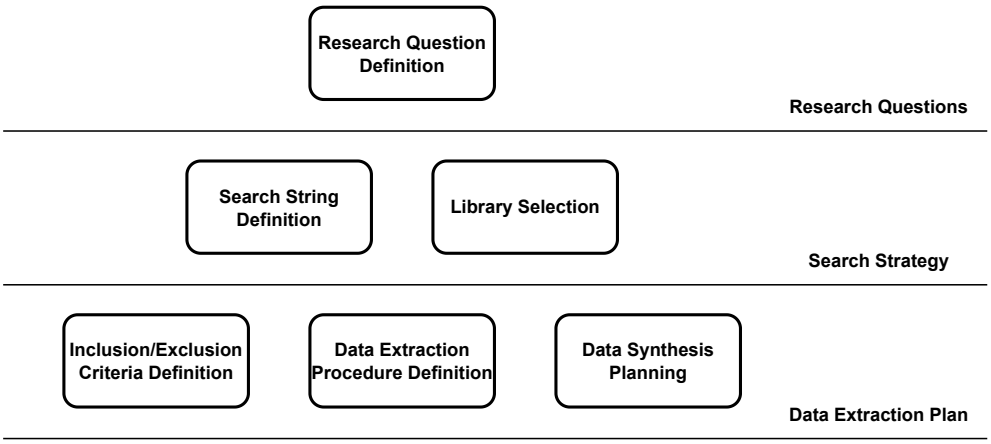

Figure 3.1: Process for Planning the Review

3.2.1 *RQ*s Definition

The aim of establishing *RQ*s in a *SLR* is to provide a clear and structured framework for the review process. It helps the reviewers to define the scope of the review and determine which studies to include or exclude based on their relevance to the *RQ*s. It also helps to identify gaps in the literature and generate insights into the state of the art in a particular research area. The objective of this Thesis is to investigate the following set of *RQ*s:

***RQ1*:** *What is the trend in system migration to microservices from 2015 till now?*

The aim of this *RQ* is to explore the architectures, languages, and domains associated with the migration scenario. Consequently, the subsequent sub-questions were formulated:

> ***RQ1.1*** *What are the software architectures of the systems that have been migrated to microservices?*
>
> ***RQ1.2*** *What are the programming languages of systems migrated to microservices?*

RQ1.3 *Which are the domains of the system migrated to microservices?*

It should be noted that this study takes into account research published from 2015, in accordance with the initial publication date of the book by Newman [15] titled "*Building Microservices*", and July 2020 that corresponds to the last update of the papers to be analysed for the data extraction.

RQ2: *What are the key attributes and characteristics exhibited by the existing studies that investigate the problem of migrating to microservices while considering software qualities?*

We aim to examine how software qualities are taken into account during the migration to MSA and identify the quality attributes considered in each of the three canonical phases. Therefore, the following research sub-questions were formulated:

RQ2.1 *What are the quality attributes of the system considered in the migration scenario?*

RQ2.2 *At what stage of the migration to microservices are quality aspects considered?*

***RQ3*:** *Which of the three steps for the migration to microservices did the researchers focus on?*

The objective of this *RQ* is to examine the techniques employed in different migration stages and link them with quality factors. The subsequent research sub-questions have been identified:

RQ3.1 *What techniques are used for the comprehension phase of the system to be migrated?*

RQ3.2 *What techniques are used for the microservices identification phase of the system to be migrated?*

RQ3.3 *What techniques are used for the packaging phase of the new microservice-based system?*

RQ3.4 *For each quality attributes and each phase of the migration process, which are the most used techniques?*

3.2.2 Search Strategy

Establishing a good research string during the planning phase of a *SLR* is crucial as it helps to identify relevant studies for inclusion in the review. A good research string consists of a combination of keywords, Boolean operators, and search terms that are tailored to the *RQ*s being investigated. By using a well-designed research string, researchers can ensure that they are capturing all relevant studies and avoiding irrelevant

ones, thus increasing the accuracy and completeness of their review. This can ultimately lead to more robust conclusions and recommendations based on the evidence available in the literature. Based on the identified *RQ*s, a set of keywords was selected for conducting the literature search. Consequently, the following research string was generated:

("micro service" OR "microservice" OR "serverless" OR "lambda function" OR "function as service" OR "FaaS" OR "cloud function" OR "utility computing") AND ((("system" OR "software") AND ("evolution" OR "transformation" OR "refactoring" OR "rearchitect" OR "re architect*" OR "reengineer*" OR "re engineer*" OR "migration" OR "modernization" OR "decomposition" OR "modularization")) OR (("service" OR "microservice" OR "micro service") AND "identification"))*

The keywords *quality* and its synonyms were not included in the research as this results in a limited set of papers. Moreover, *serverless* and its synonyms were added to the search string since container technology allows for building microservice-oriented systems with easy scalability and availability, as noted in [49].

The search was conducted on three different digital libraries, including the *ACM Digital Library*, *IEEE Explore*, and *Scopus*. It is worth noting that the *ACM Digital Library* provides two search modes, namely *ACM Full-Text Collection* and *ACM Guide to Computing Literature*. However, not all the *ACM Full-Text Collection* is included in the *ACM Guide to Computing Literature*. Therefore, we performed the search using both modes to ensure comprehensive coverage. After the search was completed, duplicate papers were removed, and the final set of results was included in a single spreadsheet.

3.2.3 Data Extraction Plan

The data extraction plan is an important component of the *SLR* process as it provides a clear and structured approach to data extraction, which helps to ensure the quality and reliability of the findings. It also helps to ensure that the review is transparent, replicable, and provides valuable insights for practice and research.

The data extraction plan typically defines the following information:

- Inclusion and exclusion criteria: This outlines the criteria used to determine whether a study should be included or excluded from the review.
- Data extraction procedures: This outlines the procedures to be followed when extracting data from the studies, such as the tools to be used, the methods for checking the accuracy of the extracted data, and any quality control procedures.
- Data synthesis: This outlines how the extracted data will be synthesized, analyzed, and presented in the final report.

Inclusion and Exclusion Criteria

Establishing inclusion and exclusion criteria is important in a *SLR* because it helps to define the boundaries of the review and ensure that the selected studies are relevant to the *RQ*s. Inclusion criteria define the characteristics that a study must have in order to be considered for inclusion in the review, while exclusion criteria specify the characteristics that would make a study ineligible for inclusion. These criteria are important to ensure that the studies selected are of high quality and relevant to the *RQ*s, and to avoid the inclusion of studies that may bias the results or not be applicable to the research objectives. The inclusion and exclusion criteria, presented in Table 3.2, were established to achieve the *SLR* goal. It should be noted that we chose to consider papers published between 2015 and July 2021.

Inclusion Criteria	Exclusion Criteria
Paper published from 2015 to 2021.	PDF not Available.
The paper refers to migration to MSA or cloud platform.	Paper not in English.
The paper refers to a case study in migration to MSA or cloud platform.	Not Primary Study.
The paper refers to quality aspects in migration to MSA or cloud platform.	
The paper refers to microservices identification from the existing system.	

Table 3.2: *SLR*: Inclusions and Exclusions Criteria.

Data Extraction Procedure

In our *SLR*, we used Zotero[1] and a series of spreadsheets to manage the data extraction process. Zotero is a free and open-source reference management software that can be used to import and organize the papers retrieved during the literature search. We used Zotero to import all the papers identified during the search process, and then eliminate duplicates based on the title, authors, and publication year.

Once we had a final list of papers, we used Excel to create a list of papers and track the data extraction process. Through Excel we recorded the study details, inclusion and exclusion criteria, and specific data items to be extracted from each paper. The specific data items extracted from each paper will be listed in the next section of the review protocol, where we will detail the variables of interest and the outcomes we will be looking for. These variables and outcomes will be based on the *RQ* and review objectives, and will be used to guide the data extraction process. We will also use

[1] https://www.zotero.org/

Excel to monitor the progress of the data extraction process, ensuring that all relevant data items are extracted and recorded accurately. The data extraction procedure is summarized in Figure 3.2.

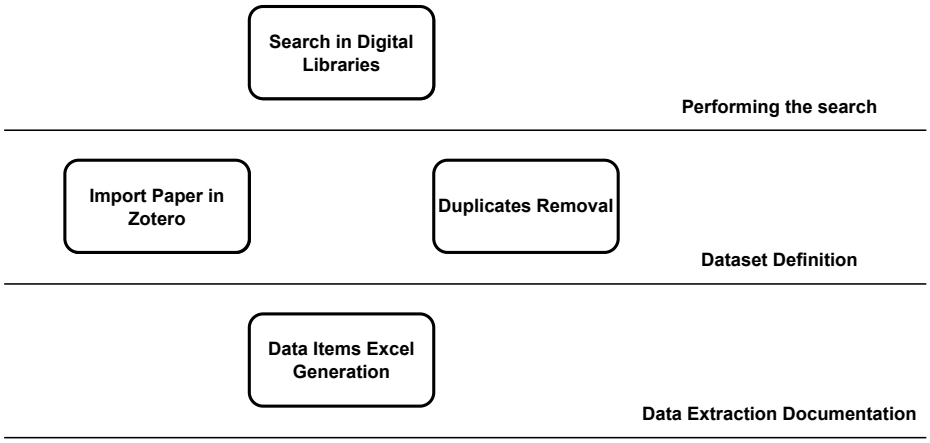

Figure 3.2: Data Extraction Procedure

Data Synthesis Planning

To ensure that our data synthesis is comprehensive and rigorous, we will use appropriate statistical methods and qualitative techniques, depending on the type of data extracted. We will identify patterns, trends, and themes in the data, and synthesize them into meaningful insights that answer the *RQ* and review objectives.

In the final report, we will present all the data necessary to answer the *RQ* and review objectives. We will provide a summary of the findings, including any significant patterns, trends, and themes that emerged from the synthesis. We will also present the limitations of the studies included and the review process, and suggest areas for future research. Our aim is to provide a comprehensive and reliable synthesis of the available evidence to inform practice and policy decisions.

3.3 Conducting the Review

According to the Kitchenham [43] guidelines, the conduction phase of a *SLR* involves carrying out the search, selection, and quality assessment of studies that meet the inclusion and exclusion criteria established in the planning phase. This phase also involves extracting relevant data from the selected studies and synthesizing the findings to address the *RQ*s.

3.3.1 Search Results

In this phase, we applied the search strategy to identify the papers that will be included for further reading. The process conducted after creating the Excel sheet is depicted in Figure 3.3.

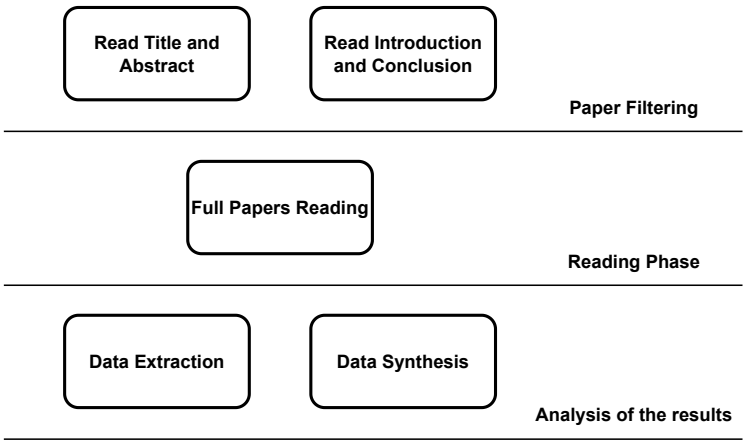

Figure 3.3: *SLR* Steps

Table 3.3 summarizes the search results of the digital libraries, indicating the number of papers retrieved from each library, and the total number of studies after removing duplicates. To determine the final set of papers for the systematic literature review, we followed a two-step approach: first, we screened titles and abstracts, and then we evaluated the introductions and conclusions.

Digital Library	Number of Papers
ACM Full-Text Collection	856
ACM Guide to Computing Literature	1286
IEEE Explore	137
Scopus	375
Total number of papers found	2654
Number of papers after duplicates removal	1615

Table 3.3: *SLR*: Digital Libraries Search Results.

The numerical outcomes of each step are presented in Table 3.4.

Phase	Number of Papers Included
Title and Abstract reading	133
Introductions and Conclusions reading	77
Full reading	58

Table 3.4: *SLR*: Selection of Studies.

3.3.2 Data Extraction

In our *SLR*, we established a range of item types, and defined several parameter categories, including general, RQ1-related, RQ2-related, RQ3-related, and an *other* category. The RQ-related parameters are intended to capture all necessary data for addressing the *RQ*s and their sub-questions. The data extraction parameters for the SLR are presented in detail in Appendix A. This appendix contains two types of tables: one set of tables represents the parameters themselves, and the other represents the possible values associated with some of these parameters. The tables are organized into four columns: ID, attribute name, type, and description. The ID column provides a unique identifier for each parameter, while the attribute name column gives a clear and concise name for the parameter. The type column indicates whether the parameter has an open value, a single value, or multiple values. Finally, the description column provides a brief explanation of the parameter's meaning and purpose.

Table A.1 outlines the general parameters and descriptions. The parameters include information about the study itself, such as its unique identifier (ID), title, authors, institution, venue, page count, keywords, and digital library source. The final parameter, "DOI" (Digital Object Identifier), is a unique identifier assigned to a digital object, such as a research article.

Table A.2 presents the parameters needed to answer RQ1, including publication year, old software architecture, new architecture, old programming language, and application domain. Publication year is a singular type, while old software architecture and old programming language are open types. The new architecture parameter is of multiple types, and the application domain parameter is of an open type.

Table A.3 lists the quality attributes considered in the study and the phase of migration in which they are considered. The second attribute is of type multiple, meaning there can be multiple phases where quality attributes are considered. Table A.4 presents the possible values associated with the second parameter, which are Comprehension, MS Identification, MS Packaging, and MS Assessment. These values indicate the phase of migration in which quality attributes are considered, such as understanding the existing system's structure and behavior or assessing the quality attributes of the newly migrated microservices system.

Table A.5 summarizes the parameters related to *RQ*3, which focuses on which of

the three steps for migration to microservices the researchers focused on. The table lists six parameters, identified by the IDs MS-1 through MS-6, along with their associated attribute name, type, and description. The first two parameters, MS-1 and MS-2, are related to the system comprehension step, which involves understanding the existing system before migrating to microservices. MS-1 and MS-2 describe the techniques and tools used to understand the system. The next two parameters, MS-3 and MS-4, are related to the microservices identification step, which involves identifying the microservices within the existing system. MS-3 and MS-4 describe the techniques and tools used to identify microservices. The final two parameters, MS-5 and MS-6, are related to the system packaging step, which involves packaging the system into microservices. MS-5 and MS-6 describe the techniques and tools used for this process. Two additional tables (Table A.6 and Table A.7) provide more details about the possible values for the MS-1 and MS-3 parameters. Table A.6 describes the different techniques that can be used to comprehend a system before migrating to microservices, including static analysis, dynamic analysis, model-driven analysis, data-driven analysis, and domain-driven analysis. Table A.7 describes the different techniques that can be used to identify microservices in a system, including static analysis, dynamic analysis, model-driven analysis, data-driven analysis, domain-driven analysis, and machine learning/optimization driven.

Table A.8, provides parameters that are not directly related to answering *RQ*s but can be helpful in analyzing trends in the migration to microservices. These parameters include the main topic of the study and how the approach is validated. The *Main Topic* attribute includes values such as migration to microservices, migration to cloud, migration to microservices and quality, and migration to cloud and quality. Those values are reported in Table A.9. The *Evaluation* attribute has two values: *Empirical*, indicating that the approach for migration is presented, and *Case Study*, which can occur in two ways: the authors either show the approach and apply it to a case study or they share the lessons learned from the migration. The described values are reported in Table A.10.

3.4 Reporting the Review

The scope of data synthesis in a *SLR* is to analyze and summarize the findings of the primary studies included in the review. This involves extracting data from the studies, organizing and categorizing the data, and synthesizing the data to answer the *RQ*s. The data synthesis phase is a critical step in a *SLR*, as it allows researchers to draw conclusions and make recommendations based on the findings of the studies reviewed. The synthesis may involve qualitative or quantitative methods, or a combination of both, depending on the nature of the *RQ*s and the available data.

In our *SLR*, we integrated the thematic analysis method [50] for qualitative data

synthesis and utilized meta-analysis [51] for quantitative data synthesis. The following section presents the analysis results to address the given *RQ*s. It should be noted that the small number of primary studies, which were considered until July 2021, is insignificant for the purpose of the annual analysis presented in this section.

Migration Phase	***Case Study***	***Empirical Study***
Comprehension	[52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80] [81]	[82] [83] [84] [85] [86]
Comprehension and Quality	[52] [87] [88] [89] [90] [91] [92] [93] [94]	[95] [96]
Microservices Identification	[52] [53] [54] [56] [97] [58] [87] [60] [61] [49] [88] [89] [63] [67] [68] [69] [90] [71] [98] [72] [74] [99] [100] [75] [77] [94] [79]	[101] [83] [84] [102] [85] [95] [96]
Microservices Identification and Quality	[55] [57] [59] [62] [64] [65] [66] [70] [91] [73] [76] [92] [86] [93] [78] [80] [81] [103]	[82]
Microservices Assessment (Quality)	[104] [56] [59] [105] [87] [49] [64] [69] [99] [75] [93] [79] [106]	[102] [96]
Packaging	[105] [49] [62] [90] [99]	[83] [84]
Packaging and Quality	[77]	

Table 3.5: Relation between Case Study and Empirical Study with Quality in Phases

3.4.1 RQ1: What is the trend in system migration to microservices from 2015 till now?

The objective of this *RQ* is to obtain a comprehensive understanding of the migration to microservices by examining the relevant architectures, languages, and domains associated with the migration process and the resulting systems. Based on the target architecture of the migration, we have identified three primary categories of papers: i) papers focused on the *migration to microservices*; ii) papers on the *migration to the cloud*; and iii) papers on the *migration to cloud microservices*. For each of these categories, there is a corresponding sub-category denoted by the suffix *"and quality"*. Papers that discuss quality aspects in any phase of the migration process are included in these

sub-categories.

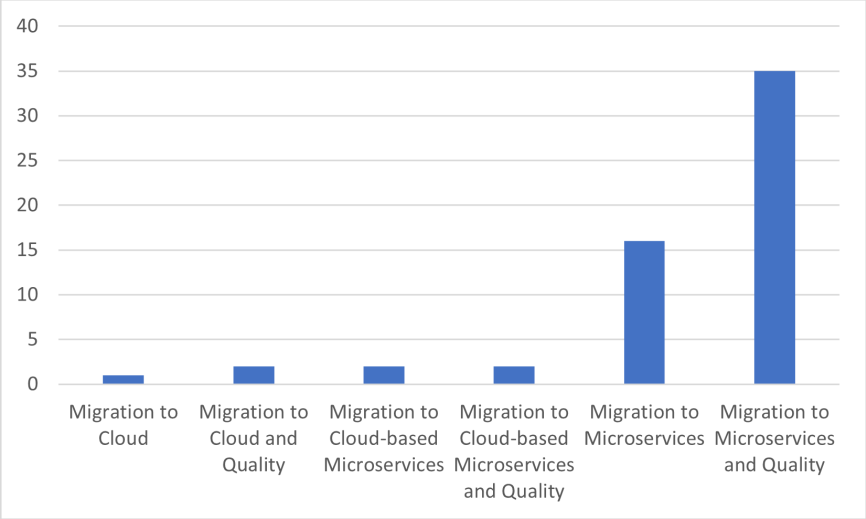

Figure 3.4: Primary Study Main Topic.

The number of papers found within the defined categories is presented in Figure 3.4. Moreover, Figure 3.5 provides an annual view that highlights the increasing trend of migration to microservices, specifically in relation to quality attributes over the years.

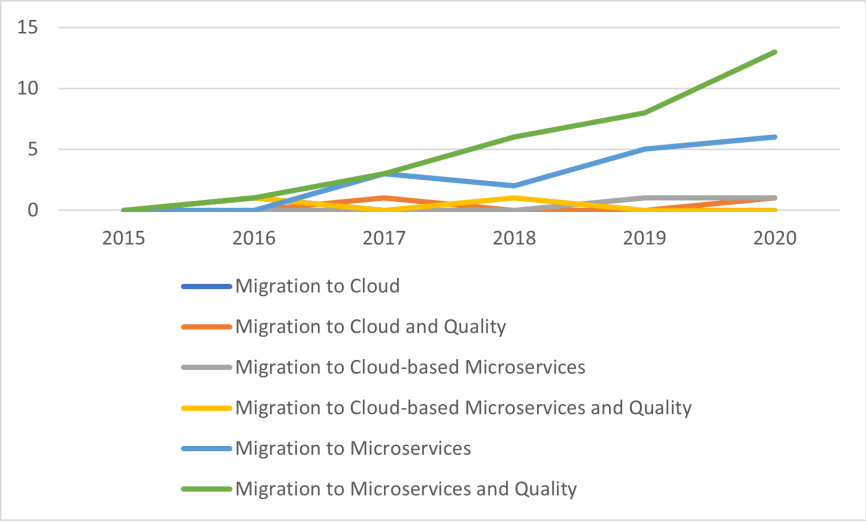

Figure 3.5: Primary Study Main Topic by Year.

We targeted primary studies based on the type of validation approach presented, which falls into two categories: i) *case study* (82.76%); ii) *explanatory study* (17.24%). In the first group, we included papers that presented approaches validated through at least one case study. The second group includes papers with purely theoretical approaches. We will focus on the *case study* labeled papers to answer *RQ1.2* and *RQ1.3*. Table 3.5 summarizes for each validation category, which papers considered the defined phases of

the migration process. Note that this table also distinguishes papers that considered quality aspects during the migration phases from those that did not.

RQ1.1: What are the software architectures of the systems that have been migrated to microservices? Figure 3.6 depicts the software architectures of the systems used as case studies to validate the migration approaches. The majority of these systems are built upon a monolithic architecture. Furthermore, as the migration process is predominantly implemented on legacy systems, researchers do not always provide information on the software architecture, which is often undefined for these types of systems. As a result, numerous generic migration approaches exist that do not rely on the initial system architecture for architectural refactoring.

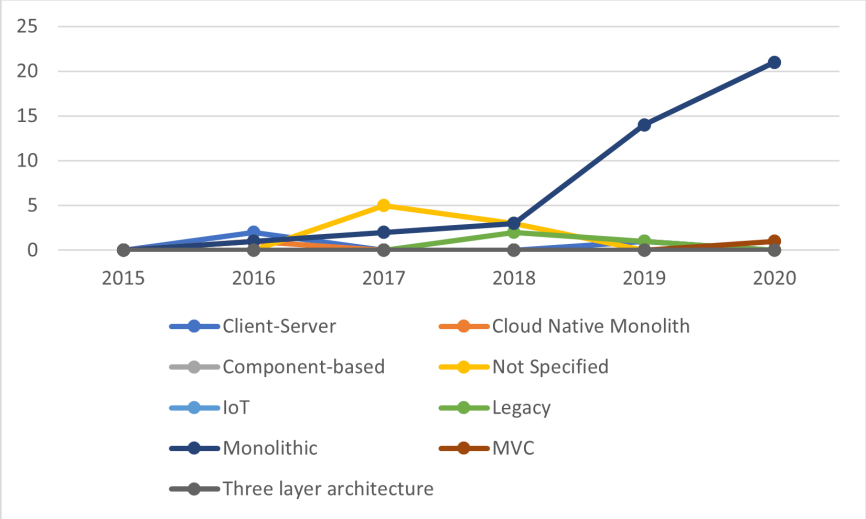

Figure 3.6: Architecture of the Legacy Application.

RQ1.2: What are the programming languages of systems migrated to microservices? The programming languages used in the systems that were considered to validate the approaches presented in the primary studies are illustrated in Figure 3.7. It is important to note that in several cases, the programming language was not explicitly mentioned in the primary studies but inferred from the domain of the application described in the case study. As per the previous findings, it is unsurprising that *Java* is the most commonly used language in the systems targeted for migration. Additionally, the systems that typically use *Java* as their programming language are those that employ monolithic architectures, which are the most commonly considered for migration.

RQ1.3: Which are the domains of the system migrated to microservices? In Figure 3.8, the application domains of the case studies used for validating the presented

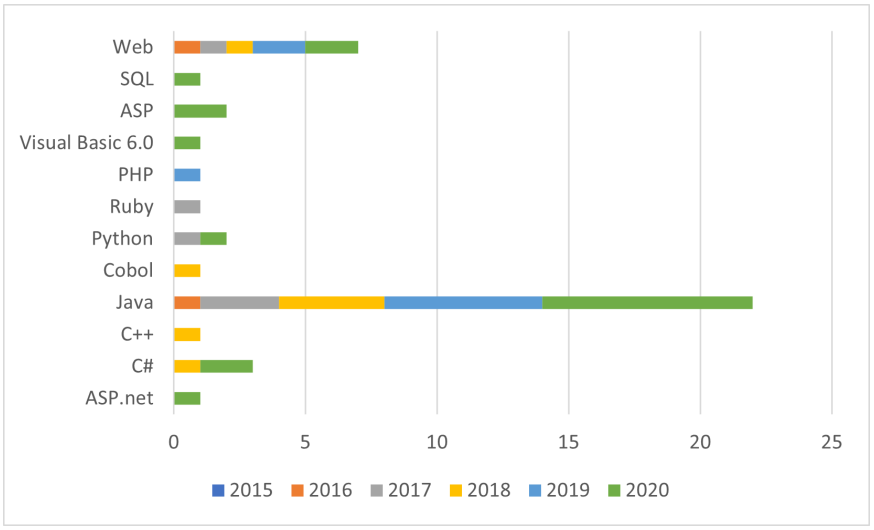

Figure 3.7: Languages of the Case Studies reported in the Accepted Papers.

approaches are displayed. As expected, a large proportion of these domains are related to web-based applications. Many of the systems being considered are open-source projects. Another commonly studied domain is Industrial Applications, highlighting the strong connection between research and business. As mentioned earlier, the field of migration is highly valued in industrial settings as well.

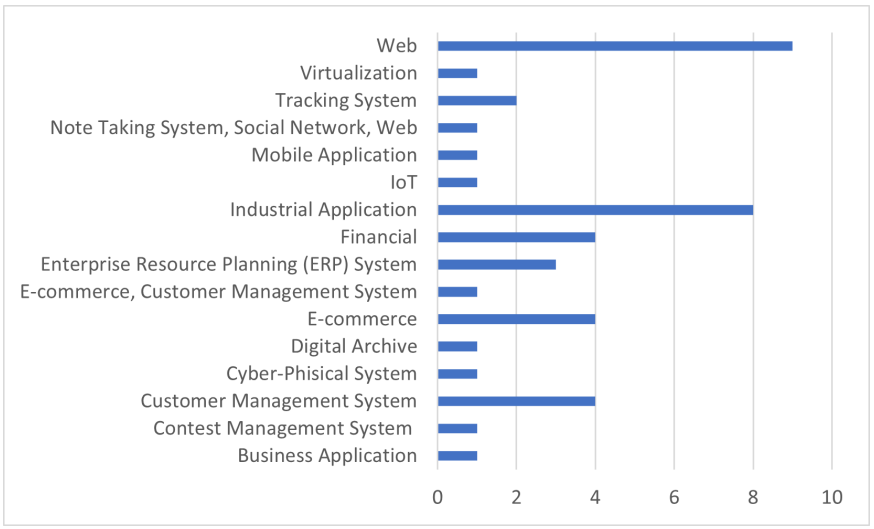

Figure 3.8: Domains of the Case Studies reported in the Accepted Papers.

3.4.2 RQ2: Are there any studies that address the problem of migration to microservices considering the quality aspects?

The data analysis conducted to answer *RQ* RQ1 has revealed a growing interest in considering software quality aspects when migrating a software system to MSA. This subsection presents an overview of the quality attributes identified and how they are

related to the different stages of the migration process.

RQ2.1: What are the quality attributes of the system considered in the migration scenario? The number of occurrences and all the quality attributes identified during the reading phase of the accepted papers are displayed in Figure 3.9.

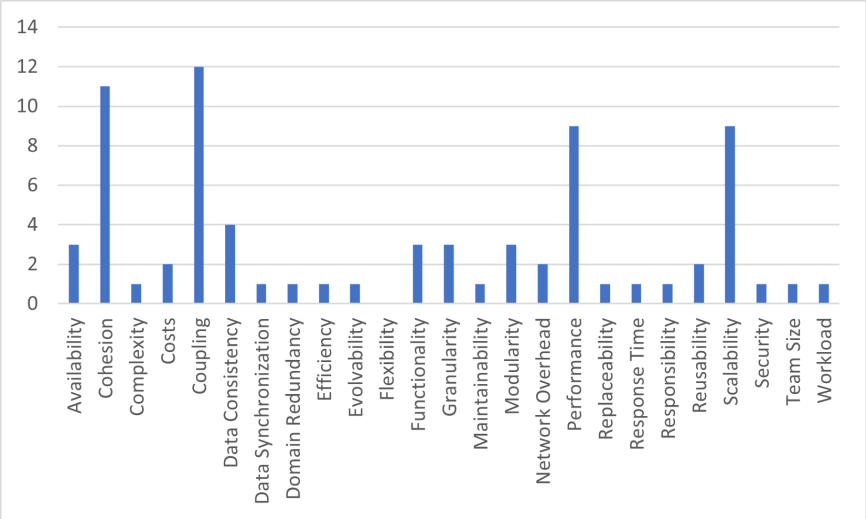

Figure 3.9: Quality Attributes Considered During Migration.

The findings indicate that researchers tend to take into account quality attributes like coupling, cohesion, scalability, and performance while making decisions about migrating to microservices. It is also worth noting that a considerable number of papers mention data consistency. These results are not particularly unexpected as properties such as coupling, cohesion, scalability, and modularization are inherent aspects of a MSA. However, the information on performance and data consistency is particularly noteworthy.

RQ2.2: At what stage of the migration to microservices are quality aspects considered? The relationship between the number of quality attributes found and each phase of the migration process is depicted in Figure 3.10. It is worth noting that an additional step, referred to as *Microservices Assessment*, has been introduced. This step involves evaluating the quality aspects of the identified microservices before proceeding with their packaging. Table 3.5 maps each primary study with the respective phase in which quality aspects are considered.

The findings reveal a tendency among researchers to focus on quality aspects during microservices identification and assessment. The limited attention given to quality attributes during the system comprehension phase suggests that the primary goal of most microservices migrations is not to enhance or meet specific quality requirements. The small number of works considering quality attributes during the microservices packaging

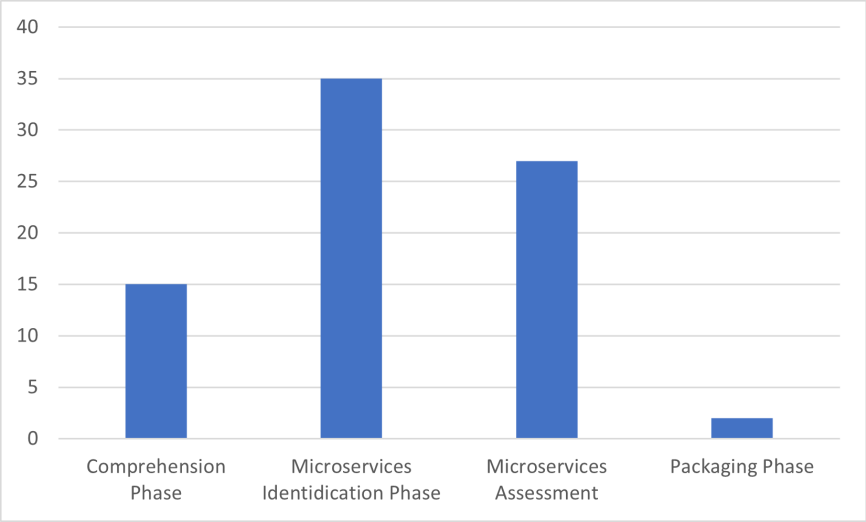

Figure 3.10: Quality Attributes in Migration Phases.

phase is attributed to its infrequent use in modernization processes. Figure 3.11 depicts the quality attributes considered during the comprehension phase of the system and their association with the reference period. The results indicate a growing interest in performance and scalability, while the trend for other quality attributes remains steady.

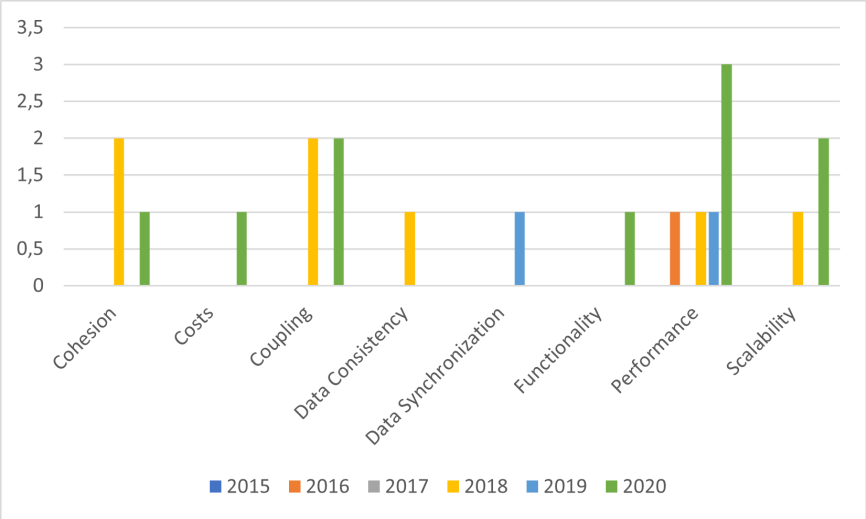

Figure 3.11: Quality Attributes in Comprehension Phase.

On the other hand, Figure 3.12 illustrates an increasing trend in the consideration of attributes such as availability, cohesion, coupling, modularity, and scalability during the identification phase of microservices.

The trend observed during the microservices assessment process shows that researchers shift their focus from studying properties such as availability, modularity, and scalability to placing greater attention on cohesion, coupling, and performance.

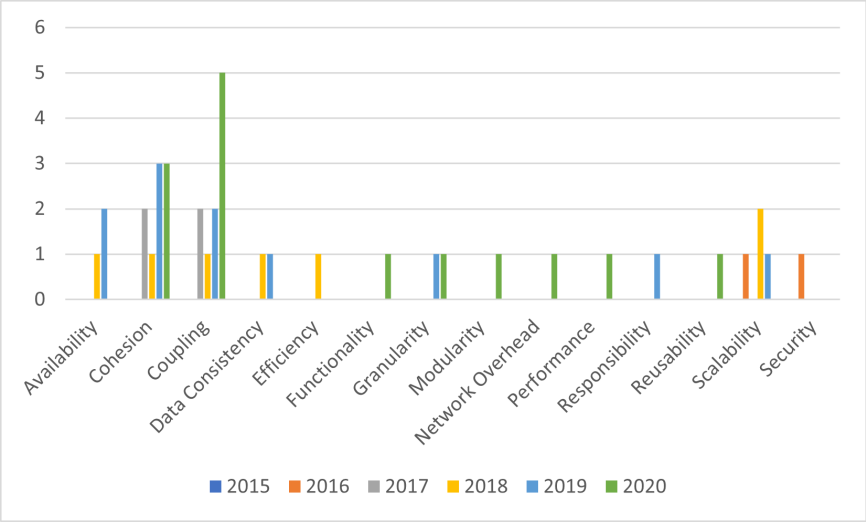

Figure 3.12: Quality Attributes in Microservices Identification Phase.

This trend is highlighted in Figure 3.13.

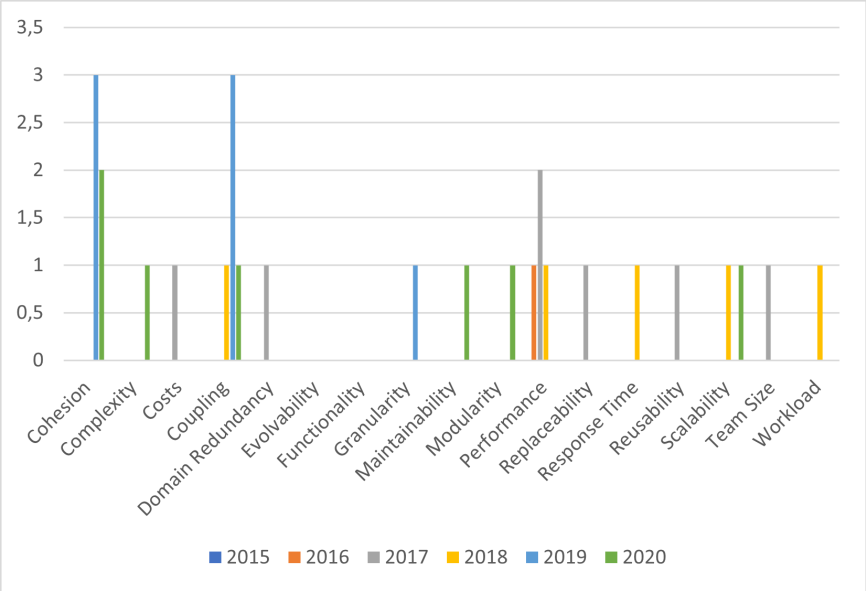

Figure 3.13: Quality Attributes in Microservices Assessment.

The quality attributes analyzed in the packaging phase of microservices deserve a separate discussion, as only a limited number of works exist on this topic. In fact, we found that only one work has considered data consistency in the system packaging phase [77].

Another noteworthy point to examine is the extent to which works address quality aspects in more than one phase of the migration. Table 3.6 presents the different combinations of migration phases and the number of papers that cover quality attributes.

Migration Phases	*Number of Papers*
Comprehension Phase and Microservices Identification Phase	2
Comprehension Phase and Microservices Assessment	2
Comprehension Phase and Packaging Phase	0
Microservices Identification Phase and Microservices Assessement	2
Microservices Identification Phase and Packaging Phase	0
Microservices Assessment and Packaging Phase	0
Comprehension Phase, Microservices Identification Phase and Microservice Assessment	1
Comprehension Phase, Microservices Identification Phase and Packaging Phase	0
Comprehension Phase, Microservices Assessment, Packaging Phase	0
Microservices Identification Phase, Microservices Assessment and Packaging Phase	0
Comprehension Phase, Microservices Identification Phase, Microservices Assessment, Packaging Phase	0

Table 3.6: Number of Studies Considering Quality Attributes in More than One Phase

Based on the analysis of the data, it appears that not all migration phases reflect the quality aspects considered in the different approaches. Only in one study [93], quality aspects were taken into account in three out of the four migration phases.

An additional interesting finding pertains to the quality attributes that are examined in more than one phase of the migration within the same study. Table 3.7 indicates that only a few quality attributes are considered in more than one phase within the same paper: i) Coupling [93], ii) Functionality [91], and iii) Performance [84] [96].

The analysis carried out reveals a significant correlation between coupling and cohesion. Specifically, papers that consider multiple quality attributes in the same phase tend to study coupling and cohesion together in: i) 27.27% of cases in the comprehension phase, ii) 47.37% of cases in the microservices identification phase, and iii) 36.36% of cases in the microservices assessment phase.

Quality Attributes	*Migration Phases*	*Number of Papers*
Coupling	Comprehension and Microservices Identification	1
Functionality	Comprehension and Microservices Identification	1
Performance	Comprehension and Microservices Identification, Comprehension and Microservices Assessment	2

Table 3.7: Number of Studies Considering the same Quality Attributes in more than one Phase

3.4.3 RQ3: Which of the three steps for the migration to microservices did the researchers focus on?

The aim of this *RQ* is to examine which of the three migration phases researchers focus on more and the techniques used. The number of papers working on each migration phase is summarized in Table 3.8.

Migration Phases	*Number of Papers*
System Comprehension	44
Microservices Identification	53
Packaging	7

Table 3.8: Number of Studies Working on the Related Phase

In general, there is a higher inclination towards identifying microservices by conducting a thorough examination of the system being migrated. Another interesting aspect to consider is the correlation between the different migration phases in the same paper. In particular, Table 3.9 presents the count of primary studies that concentrate on the mentioned migration phases.

Migration Phases	*Number of Papers*
Only on System Comprehension	0
Only on Microservices Identification	6
Only on Packaging	1
Comprehension and Microservices Identification	41
Comprehension and Packaging	0
Microservices Identification and Packaging	3
Comprehension, Microservices Identification and Packaging	3

Table 3.9: Number of Studies Working on the Related Phases

The findings indicate a close correlation between the understanding of the current system and the identification of microservices. This is because the techniques for identifying microservices are often based on the output obtained in the system comprehension phase. The following paragraphs will describe the most commonly used approaches for each phase. It is worth noting that for both the *system comprehension phase (CP)* and *microservices identification phase (MIP)*, the six categories listed in Table 3.10 were established.

Approach	*Phase*	*Description*
Static Analysis	CP, MIP	The approach is based on: code, dependencies, classes, methods, packages, files, directories, text analysis.
Dynamic Analysis	CP, MIP	The approach is based on: logs, traces, use case testing, user experience testing, user stories testing.
Model-Driven Analysis	CP, MIP	The approach is based on: models and meta-models of the system.
Data-Driven Analysis	CP, MIP	The approach is based on data.
Domain-Driven Analysis	CP, MIP	The approach is based on: bounded context, business process, requirements analysis.
Machine Learning or Optimization Driven	MIP	The approach is based on clustering techniques.

Table 3.10: System Comprehension and Microservices Identification Approaches Definition

Finally, we examined whether the same approach was used in more than one phase

within the same paper. Table 3.11 displays the results, showing a tendency to use static analysis and domain-driven approaches together in both the system comprehension and microservice identification phases. However, regarding the use of the same approaches in both phases, the only significant finding is the simultaneous use of static analysis and domain-driven approaches.

Approach	*Number of Papers*
Static Analysis	9
Domain-Driven	6
Dynamic Analysis	3
Static Analysis and Domain Driven	2
Data Driven	1
ML/OPT	1
Static Analysis and Dynamic Analysis	1
Domain-Driven, Data-Driven	1
Model-Driven, Domain-Driven	1

Table 3.11: Relation Between Approaches Both in Comprehension and Microservices Identification Phase

RQ3.1: What techniques are used for the comprehension phase of the system to be migrated? The distribution of system comprehension approaches among the primary studies included in this *SLR* is depicted in Figure 3.14.

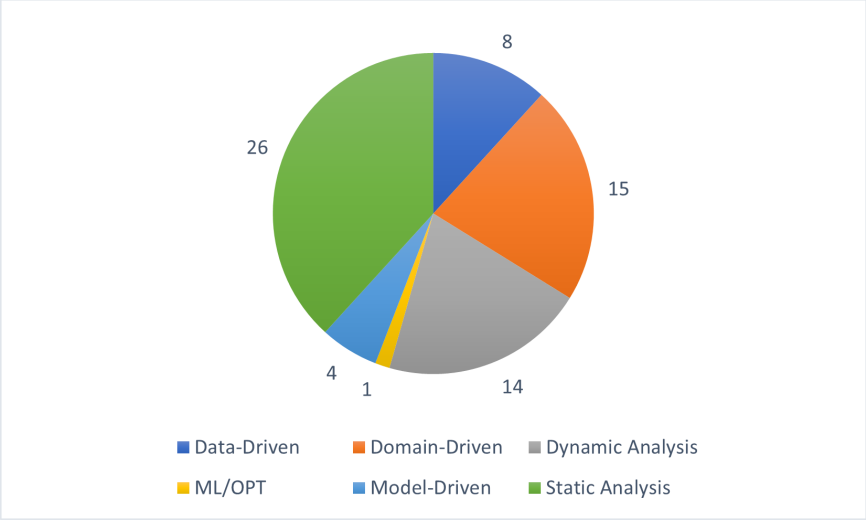

Figure 3.14: Comprehension Approaches Distribution

The majority of the works included in this *SLR* utilize techniques that fall into three

categories: static analysis, dynamic analysis, and domain-driven analysis. Several papers also focus on data-driven approaches. Figure 3.15 presents a diagram that displays the distribution of these approaches over the reference period. The graph indicates a growing interest in techniques such as static and dynamic analysis, as well as the study of the domain, which is consistent with the previous graph in Figure 3.14.

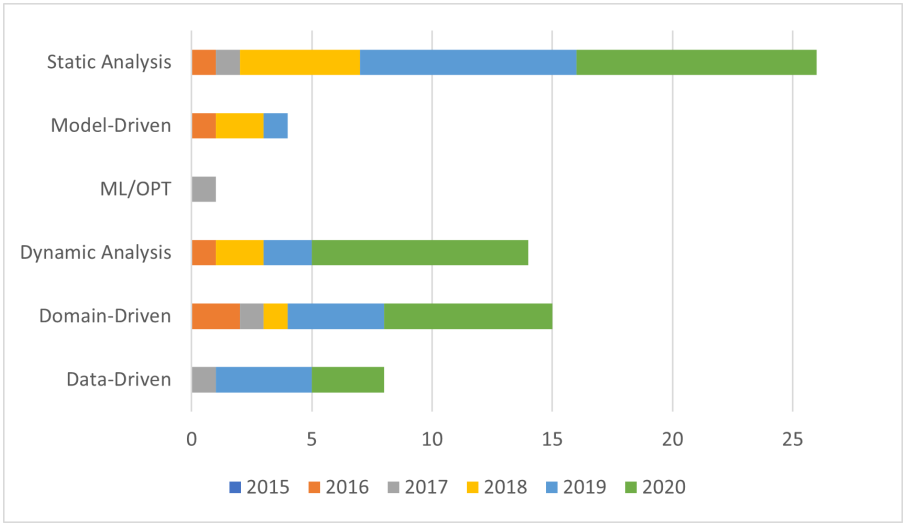

Figure 3.15: Comprehension Approaches Distribution by Year

Upon analyzing the papers that perform the comprehension phase, it is evident that 27.27% of them use both static and dynamic analysis techniques simultaneously. Additionally, static analysis has been combined with domain-driven approaches in 20.45% of the papers. Finally, domain-driven approaches are used in conjunction with data-driven techniques in 9.09% of the papers.

RQ3.2: What techniques are used for the microservices identification phase of the system to be migrated? As we move on to the phase of microservices identification, there is a noticeable change in the approach. As illustrated in Figure 3.16, the three most commonly used techniques remain those of static and dynamic analysis of the system, with a considerable rise in the use of machine learning/optimization. Clustering algorithms are frequently employed on graphs created based on the aforementioned static and dynamic analyses.

The distribution of techniques over time is illustrated in Figure 3.17. It is evident that the use of static analysis, machine learning/optimization, and domain-driven approaches has been increasing over time.

The analysis results reveal significant relationships between various approaches. One notable finding is that the ML/OPT category of approaches is often paired with i) static analysis in 17% of the papers, ii) domain-driven approaches in 5.66% of the papers, and

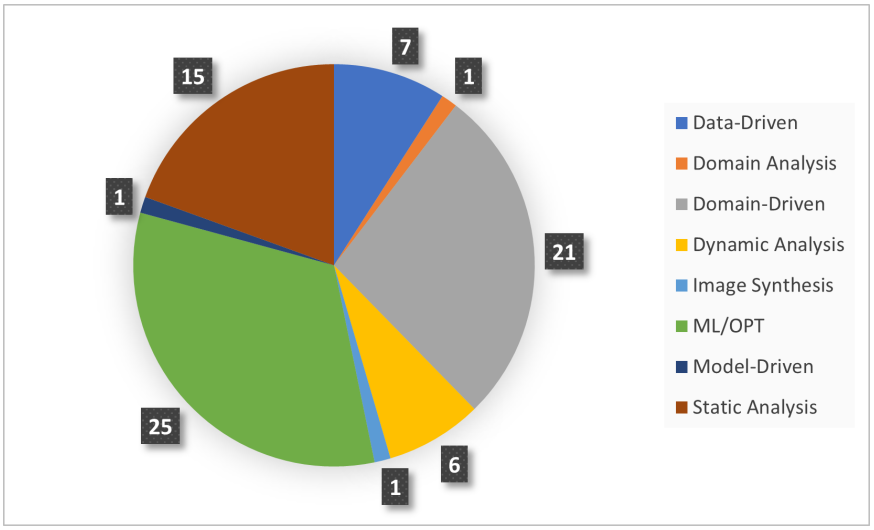

Figure 3.16: Microservices Identification Approaches Distribution

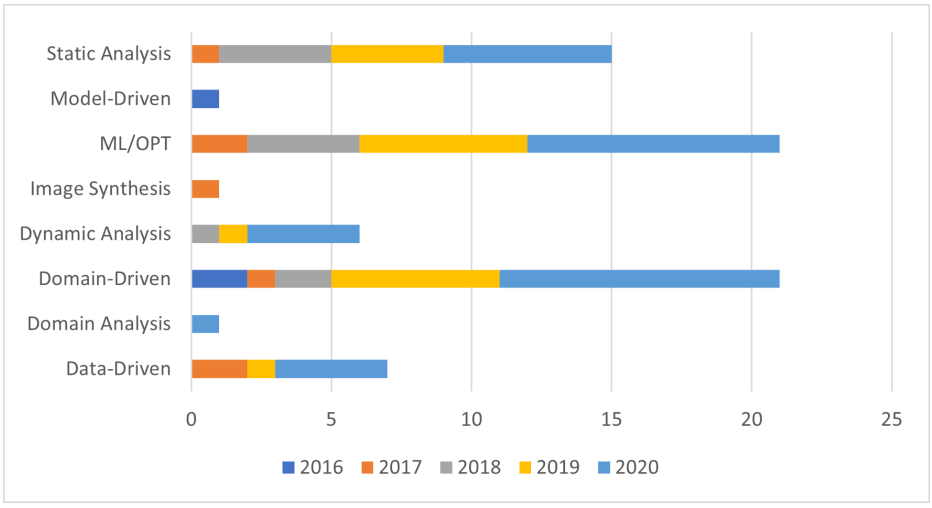

Figure 3.17: Microservices Identification Approaches Distribution by Year

iii) data-driven approaches in 5.66% of the papers. Furthermore, there is a strong correlation between data-driven and domain-driven techniques, which are used together in 7.55% of the cases. Lastly, there is an interesting trend towards the concurrent use of static analysis and domain-driven approaches, which is observed in 13.21% of the papers.

RQ3.3: What techniques are used for the packaging phase of the new microservice-based system? There are only seven papers that address the microservices packaging phase, which are listed in Table 3.5. All of these studies utilize containerization techniques, employing tools such as Google Cloud and Docker.

RQ3.4: For each quality attributes and each phase of the migration process, which are the most used techniques? The research sub-question aims to

explore the relationship between quality attributes and the approaches used in the system comprehension and microservices identification phases. The matching of the top five approaches used in each phase is summarized in Tables 3.12 and 3.13.

Quality	**Data Driven**	**Domain Driven**	**Dynamic Analysis**	**Model Driven**	**Static Analysis**
Cohesion			X	X	X
Coupling			X	X	X
Data Synch.	X	X			
Functionality			X		
Performance	X	X	X	X	X
Scalability		X			

Table 3.12: Relation Between Quality Attributes and Comprehension Phase Approaches

Upon analyzing the relationship between quality attributes and approaches in the comprehension phase, it appears that the coupling and cohesion quality attributes are assessed using the same techniques: static and dynamic analysis, and model-driven approaches. The same trend is observed in the microservices identification phase, with the addition of data-driven, domain-driven, and ML/OPT approaches to the static and dynamic analysis techniques used in the comprehension phase.

However, when it comes to the performance attribute, there is no overlap between the approaches used in the two phases. Regarding the scalability attribute, there is a discrepancy between the approaches used in the two phases. While in the comprehension phase, scalability is associated solely with domain-driven approaches, in the microservices identification phase, dynamic analysis and ML/OPT techniques are also used.

Quality	Data Driven	Domain Driven	Dynamic Analysis	ML or OPT	Static Analysis
Availability		X	X	X	X
Cohesion	X	X	X	X	X
Coupling	X	X	X	X	X
Data Consistency	X	X		X	
Efficiency		X	X		
Functionality			X		
Granularity		X		X	
Modularity	X			X	X
Network Overhead	X			X	X
Performance				X	
Responsibility				X	X
Reusability	X			X	X
Scalability		X	X	X	
Security		X			

Table 3.13: Relation Between Quality Attributes and Microservices Identification Phase Approaches

3.5 Discussion

The trend of considering quality attributes during microservices migration is increasing in the research community. Out of all the papers included in this *SLR*, 67.24% of them consider software qualities during migration. However, quality improvement does not seem to be a primary goal of migration, as only a limited number of works (5.12%) consider the same set of quality attributes in at least two phases of the migration. The most commonly considered quality aspects during the comprehension phase are coupling (36.36%), cohesion (27.27%), performance (54.54%), and scalability (27.27%). Similarly, during the microservices identification phase, the most studied qualities are coupling (57.89%), cohesion (52.63%), scalability (21.05%), availability (15.79%), and modularity (15.79%). When assessing the identified microservices, the most studied quality attributes are cohesion (35.71%), coupling (35.71%), and performance (28.57%).

Although many approaches are based on the static and dynamic structure of the system, the relationship between quality aspects and the techniques used in each phase is not consistent. In the comprehension and microservices identification phases, Data-Driven, Domain-Driven, Dynamic Analysis, and Static Analysis approaches are used primarily to focus on system functionality. Clustering algorithms are often applied to identify microservices based on the system's static and dynamic structure. However, there is no evidence of qualities related to the packaging phase since researchers con-

sider this phase in very few cases. The results indicate a great interest in attributes such as coupling and cohesion, while other quality aspects are not given much attention. Nonetheless, performance-related aspects are gaining increasing attention from researchers.

3.6 Conclusion

The trend of migrating to microservices is increasingly popular in the research community. When planning this process, software engineers must take various parameters into account, from functionality to the quality of the system. Additionally, it is crucial to have a clear goal for the migration. To examine how researchers have considered quality aspects in the migration process, we conducted a *SLR* on quality-driven approaches to migration. Our selection procedure resulted in 58 papers published between 2015 and July 2021, out of over 2000 results. The study revealed that while many researchers consider quality attributes during the migration, none seem to prioritize quality improvement as a goal. Most of the selected works propose migration approaches from monolithic systems, predominantly web-based languages. We also analyzed the quality attributes focused on during each phase of migration, highlighting strong attention to attributes such as coupling, cohesion, and performance. Finally, we related a set of related approaches to each quality attribute for each migration phase.

Part II

Research Contributions

Chapter 4

Quality-Driven Migration Approaches

This Chapter introduce the concept of a quality-driven migration approach and how it differs from other migration approaches. The significant stages involved in this approach and how each stage impacts the overall success of the migration project will be discussed. The Chapter will present two different conceptual approaches. The first, has been presented in the Early Career Research forum at the European Conference on Software Architecture 2021 but not implemented. The latter, is a refinement of the process that allows to overcome the limitations of the previous one. Part of the second approach has been validated as reported in next Chapters. By the end of this Chapter, readers will have a deeper understanding of the benefits of a quality-driven approach to software migration and will be equipped with the knowledge needed to plan, execute, and validate a successful migration project that meets the desired quality standards.

4.1 Related Work

This Section reports the major work presented in the literature to highlight the novelty of the proposed approach. As reported in Chapter 3, different researches addressed the problem of migration to microservices from monolithic systems. In [87], the authors present a program analysis-based method to migrate monolithic applications to a microservices architecture using static and dynamic analysis of the system. Their method has high accuracy and low performance cost. The authors in [88], propose an approach for automatic identification of microservices from Object Oriented source code, measuring both the structural and behavioural validity of the identified microservices and their data autonomy. In [107], a tool to continuously streaming down performance data, analyzing them and feeding back to the migration process the results of the analysis is presented. This approach guarantee that the new system does not fall short in terms of performance. The authors in [108] consider five quality criteria observed as rele-

vant by practitioners for system migration. Compared with a baseline approach that considers just coupling and cohesion, their approach reinforced the need for adopting more criteria than traditional ones. In [91], the authors decompose a monolith application into independent microservices. The functionality distribution is optimized to guarantee a system with high cohesion, low coupling and good-sized services focused on core functionality. The authors in [79] propose the Functionality-oriented Service Candidate Identification (FoSCI) framework for the identification of service candidates from a monolithic system. Both the dynamic analysis and search-based functional atom grouping algorithm serves to service candidates extraction. The service candidate evaluation suite uses 8 different metrics, measuring functionality, modularity, and evolvability of the identified service candidates. In [106] the authors provide a validation framework for microservices resulted from (semi-)automatic decomposition of monoliths validated through an open-source framework.

The analysis of the state of the art, as reported also in Chapter 3, revealed that the main driver for the decomposition of legacy systems are the functional requirements. In addition, non of the approaches found in the literature considers antipatterns as a technique to comprehend the existing systems for performing the migration. In the approach presented in this Chapter, the uncovered aspects are examined with the aim of performing a quality-driven migration to microservices.

4.2 Process Objective

The current state of migration to microservices aims to move from a monolithic architecture to microservices. In the identification of microservices, two quality aspects, namely coupling and cohesion, are given due consideration. Further, an evaluation of the extracted microservices is carried out to guarantee their quality. The migration process follows a functionality-driven approach, where the functionality of the monolithic application serves as a basis for extracting microservices.

To enhance the quality attributes of a system, we propose a quality-driven migration to a MSA. In our context, the migration to microservices will serve as an instrument used to enhance software qualities. To ensure the success of this migration, our quality-driven process prioritizes software qualities such as performance, security, and maintainability throughout the migration process. Our focus will be on developing high-quality microservices that are well-designed, structured, and documented. By following a quality-driven process, we aim to ensure that the migrated system meets our quality standards and delivers the expected benefits.

As mentioned in the introduction, the main goal of this Thesis is to define a process that can enhance system quality attributes by migrating to microservices. Thus, we have formulated a RQ and sub-RQs that guide our investigation.

RQ1 : *How to create a quality-driven migration process?*

RQ1.1 : *How to perform a quality-driven microservices identification?*

RQ1.2 : *How to deal with multiple quality attributes?*

The idea behind our process is to start by identifying a subset of quality attributes that we want to improve and analyze the existing system's problems in relation to those quality aspects. Once we have identified the weaknesses, we will proceed by resolving them by migrating to microservices. By following this approach, we aim to develop a process that can effectively enhance the system's qualities. Figure 4.1 shows the overall idea of the process.

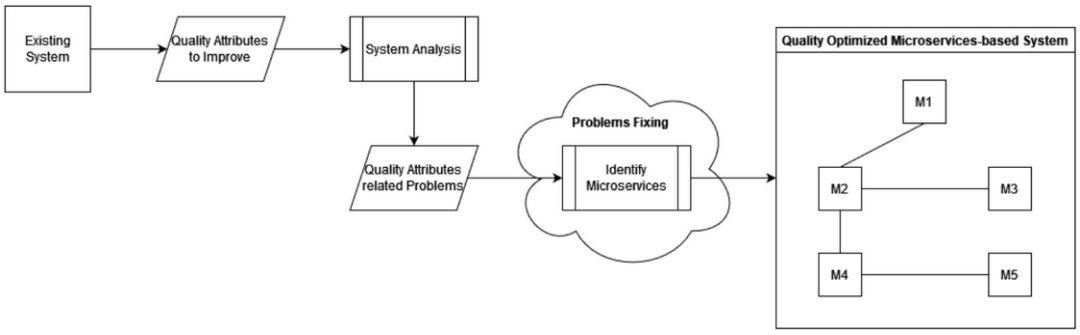

Figure 4.1: Overall Quality-Driven Migration to Microservices

4.3 Quality-Driven Process

One of the goal of this Thesis is to define a process allowing to improve software qualities by migrating to microservices. The aim of the process is to guide software architects migrating their systems to microservices while preserving or achieving predefined non-functional requirements. The proposed approach matches the common system migration to microservices process focusing on a set of system quality attributes. The three considered steps for migration are the following: the *existing system comprehension*, *microservices identification and assessment*, and *microservices packaging*.

4.3.1 Existing System Comprehension

A crucial step in establishing a quality-driven migration process is to gain a comprehensive understanding of the current system and identify areas where quality is lacking. This involves pinpointing specific quality attributes that require improvement. Next, microservices are identified for their potential to address the corresponding quality issues. Thus, the microservices are then packaged.

One obstacle in the creation of a quality-driven migration process is to understand the part of the system degrading the defined quality of the legacy system. To address this

challenge, an antipattern analysis can be conducted. Antipatterns are commonly used to identify recurring design problems and poor programming practices that negatively affect software qualities. By analyzing the existing system for antipatterns associated with the targeted subset of quality attributes, areas that need attention can be identified and prioritized for migration to microservices.

The process of understanding the existing system, referred to as the "existing system comprehension phase" in Figure 4.2, involves examining the system for antipatterns that impact the subset of quality attributes selected for enhancement. This stage employs a range of artifacts, including models, code, traces, and logs, to obtain a comprehensive understanding of the system. The process is driven by quality concerns and involves both static and dynamic analysis of the system. To determine the quality aspects that require improvement, architects may need to conduct a preliminary analysis of the system. To recognize the elements that are diminishing the system's quality for each quality attribute, a set of architectural antipatterns is provided. This process is known as antipattern detection and produces the number of instances of the identified antipatterns and a collection of associated patterns.

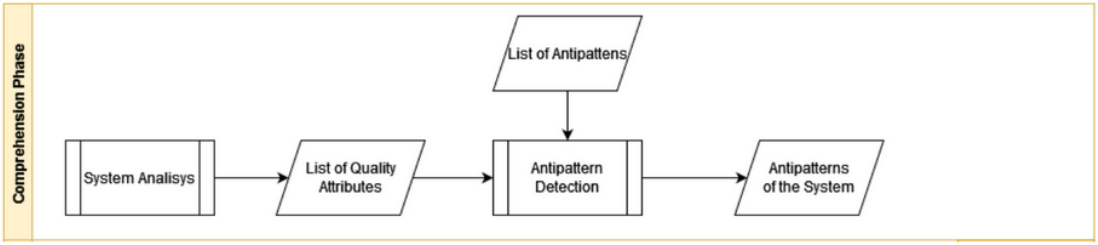

Figure 4.2: Quality-Driven System Comprehension

4.3.2 Microservices Identification and Assessment

Various approaches are available to perform a *quality-driven microservices identification*, such as static analysis, dynamic analysis, domain-driven analysis, data-driven analysis, and others, to address RQ1.1. However, the challenge is to select the most suitable approach that aligns with the project's goals and requirements. Therefore, the proposed solution is to apply well-defined patterns that can serve as a microservices identification approach. These patterns provide a set of guidelines for identifying, extracting, and defining microservices, ensuring a quality-driven migration process and achieving the desired outcomes. The *microservices identification phase* inherits the quality aspects directly from the system's comprehension phase. The list of patterns is restricted to the set of antipatterns detected in the first phase of migration. Each pattern consists of several tactics, and the goal is to augment each pattern with a well-defined microservices identification strategy based on a static or dynamic view of the system. The microservices identification is then performed according to the augmented pattern, and

the new system is simulated and analyzed to check if the quality attribute is achieved. These steps are repeated until the most suitable tactic is added to the pattern. The *microservices identification phase* outputs one possible decomposition for each quality attribute to be improved. The process of identifying microservices is detailed in Figure 4.3.

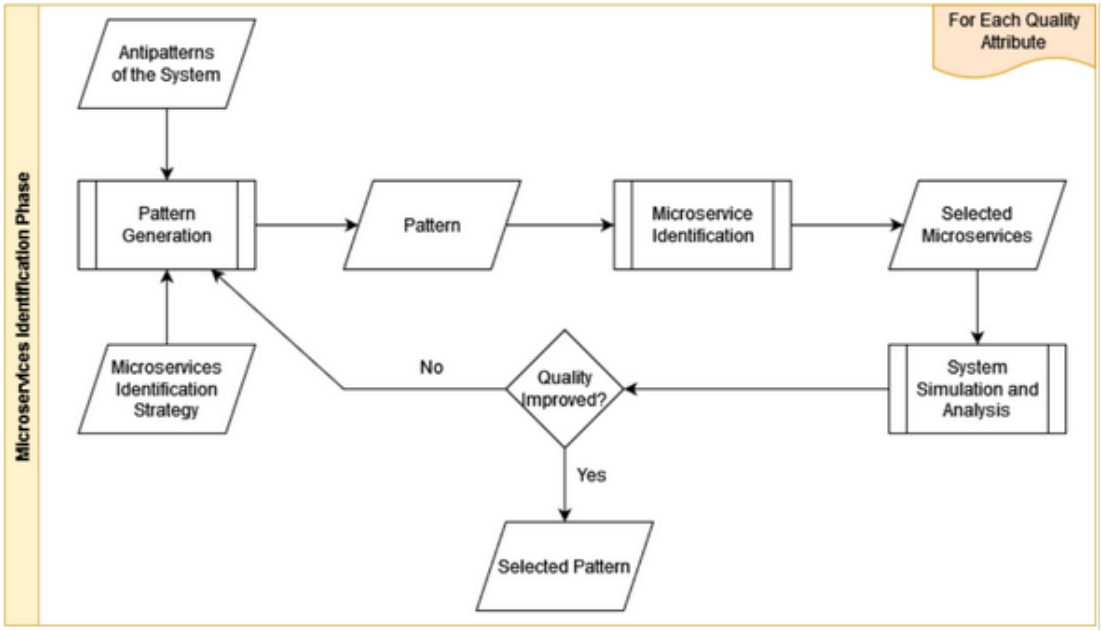

Figure 4.3: Details of Quality-Drien Microservices Identification

In the *microservices identification and assessment phase*, the input is the antipatterns identified during the Existing System Comprehension Phase and the available microservices identification strategies, and the output is the selected pattern serving as the microservices identification strategy for the migration process. The method used involves generating patterns for microservices identification and then applying them to identify the microservices needed. Finally, the system is simulated to assess their effectiveness in improving the identified quality attributes.

4.3.3 Microservices Packaging

In order to address multiple quality attributes during microservices identification, it is necessary to prioritize and select the most important ones for improvement. However, choosing a microservices identification approach that maximizes the results for all quality attributes can be a difficult task. To tackle this issue, an optimization algorithm can be used to determine the most suitable microservices identification strategy based on the prioritization of quality attributes. The algorithm can take into account various factors, such as the effectiveness of the strategy, the implementation cost, and the overall impact on the system. With the application of an optimization algorithm, an informed

decision can be made about the microservices identification strategy that will provide the best overall result for the system. The *microservices identification and assessment phase* yields a collection of possible decompositions, with each of them relating to a single quality attribute to be improved. As our objective is to propose the optimal system decomposition strategy based on a set of quality attributes, a pre-processing step is required in this phase. To achieve this, the idea is to provide a multi-objective optimization algorithm to discover the optimal decomposition. The output of this algorithm will be the set of microservices to be packaged and deployed. Figure 4.4 outlines the necessary steps involved in performing the *packaging phase*.

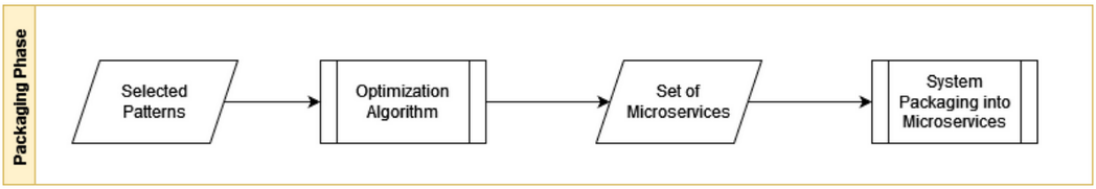

Figure 4.4: Quality-Driven Microservices Packaging

4.3.4 Summary and Evaluation of the Proposed Process

Figure 4.5 depicts the entire Quality-Driven Migration process. The figure represents the complete flow of operations required to achieve the desired objective.

The process consists of three phases. The first phase, *existing system comprehension*, encompasses all activities necessary to understand the existing system. In our case, this involves analyzing the system's problems, which entails identifying and analyzing antipatterns. The second phase, *microservices identification and ssessment*, focuses on identifying microservices and assessing them based on their quality characteristics. This phase also involves increasing the number of solution patterns using tactics, which are strategies for decomposition. The third phase, *microservices packaging*, evaluates all the proposed decompositions and selects the best one that satisfies most (if not all) of the non-functional requirements. This phase ensures that the final microservices-based system meets the quality constraints specified in the first phase. Overall, the figure presents a comprehensive overview of the Quality-Driven Migration process, highlighting the key steps involved in achieving the desired outcomes.

Table 4.1 provides a clear and concise summary of the pros and cons of the quality-driven process.

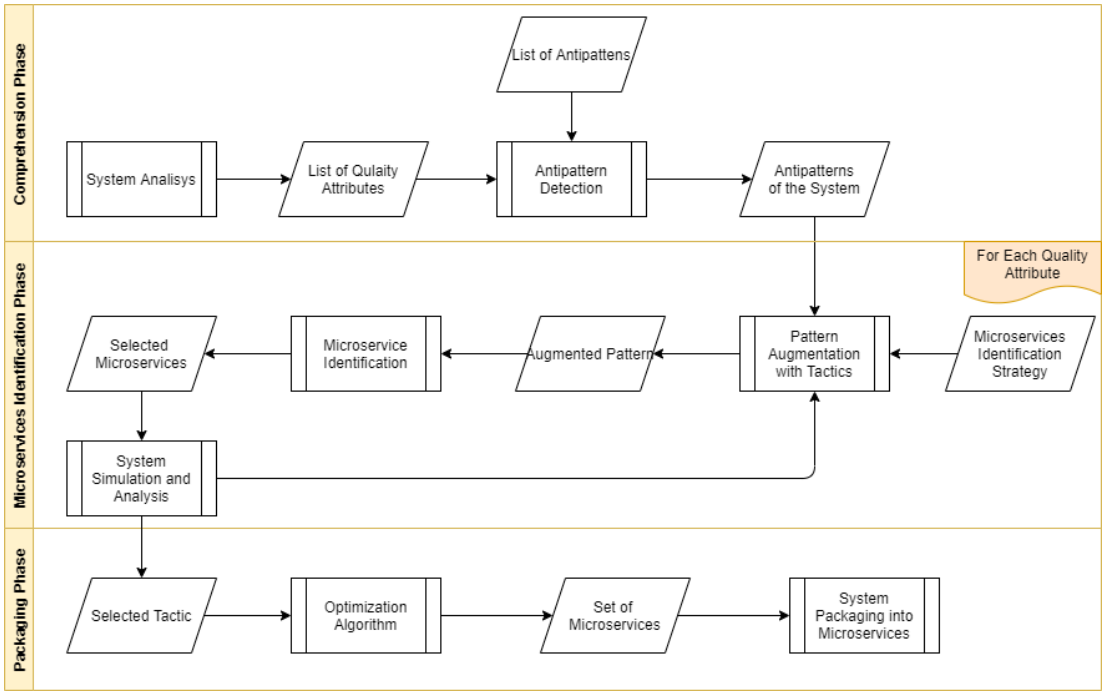

Figure 4.5: Quality-Driven Migration to Microservices Process

Pros	***Cons***
Ensures high quality software by focusing on quality attributes in each phase	Lengthy and complex process
High probability of meeting established quality constraints	Difficulties in categorizing anti-patterns and associating them with quality attributes
Promotes effective decomposition	Determining decomposition strategies for resolution patterns can be challenging
Enhances software qualities required	Selected patterns must not negatively impact other software qualities
-	May discourage or intimidate companies from using due to its complexity and length

Table 4.1: Evaluation of the Proposed Quality-Driven Migration Process

Pros Focusing on quality attributes in each phase ensures that the resulting software is of high quality, which is a major advantage. However, there are several challenges associated with this process. The process is lengthy and complex, which can make it difficult to implement.

Cons One of the challenging aspects of the quality-driven process is how to categorize antipatterns and associate them with one or more quality attributes. Additionally, determining which decomposition strategies to associate with resolution patterns can be difficult. It is also crucial to ensure that the selected patterns do not negatively impact other software qualities required. Moreover, it is important to consider that this process's industrial nature may discourage or even intimidate companies from using it due to its complexity and length (evaluated with the partner company). This is especially relevant in the context of this doctoral Thesis, which aims to produce a final product that meets industrial standards.

Solutions Therefore, a simplified version of the process has been created to mitigate these concerns, which will be presented in the next section. Overall, while the quality-driven process has many benefits, its length and complexity present several challenges that must be overcome to ensure successful implementation.

4.3.5 Semplification of the Quality-Driven Process

The process shown in the previous section provided an opportunity to develop a more streamlined quality-driven migration process in which microservice migration and architectural refactoring are integrated to create a single approach. The final migration process is depicted in Figure 4.6. This quality-driven migration process consists of three phases: migration planning, system decomposition and migration execution. Each of these phases will be explained in detail in the following subsections.

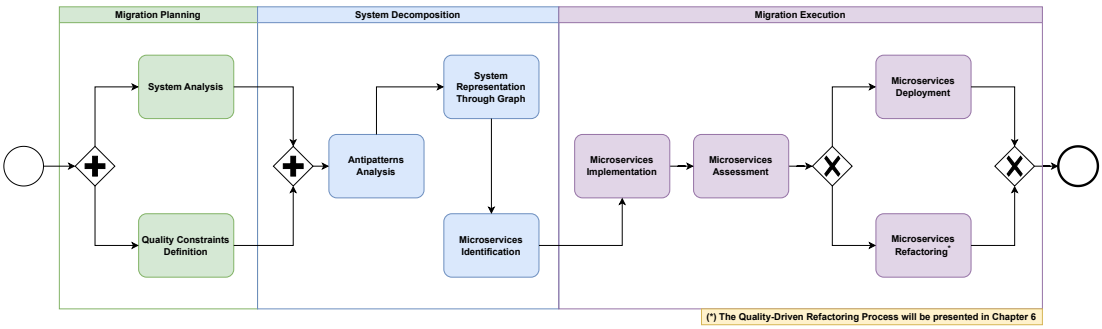

Figure 4.6: Quality-Driven Migration to Microservices Process

It is important to note that the process presented is merely a guideline for practitioners. As such, precise details on how to carry out all subtask associated with the various phases are not provided. Every system is unique, and the specific requirements and challenges of each migration project will vary. Therefore, it is up to the practitioners to tailor the process to fit the specific needs of their project. However, the general framework presented here provides a solid foundation for approaching quality-driven migration to microservices.

Migration Planning The migration planning phase of our quality-driven migration approach involves analyzing the current system's functionality and dependencies, context, and eventually market demands with the aim of planning the migration process. In this phase, the quality objectives are then defined, which may involve improving or maintaining existing quality levels in the legacy system, or achieving new quality goals.

System Decomposition The system decomposition phase is a crucial step in our quality-driven migration process, which involves several activities. Firstly, we conduct an in-depth analysis of the legacy system to identify any antipatterns that are relevant to the quality objectives we have defined. Next, a graphical representation of the system in terms of classes and methods is created. Then, a graph augmentation techniques is applied to visualize the antipattern data on the graph. A possible system representation of a legacy system will be provided in Chapter 5. Finally, based on the informations reported on the graph, the system is decomposed using machine learning or optimization algorithms that are tailored to meet the system's specific requirements. By breaking down the system into its individual components, we can ensure that it meets the quality standards we set.

Migration Execution In the migration execution phase of our quality-driven migration process, the microservices are implemented, followed by an assessment to determine if they meet the defined quality standards. The assessment can have one of two outcomes: either the microservice(s) meet the quality criteria, and can be deployed, or the microservice(s) require refactoring to improve their quality. If the microservice(s) meet the defined quality standards, they can proceed to deployment. However, if the assessment indicates that the microservice(s) do not meet the quality standards, a quality-driven refactoring process is required to improve them. We will discuss the refactoring process in more detail in the Chapter 6. Overall, the migration execution phase is critical to ensure that the microservices are implemented correctly and meet the required quality standards.

Process Improvements and Weaknesses The modified version of the quality-driven migration approach eliminates many of the drawbacks of its predecessor. Firstly, the identification of microservices is simplified, removing the difficulty of associating each pattern with a decomposition strategy. Secondly, the process is made more straightforward, leaving companies with the sole task of defining quality constraints and executing the migration. By streamlining the microservice identification process, the updated approach removes a significant obstacle in the migration journey. In the original method, determining which microservices to create and how to break down monolithic applications into these services was a complex and time-consuming process. With the new approach, this process is simplified, allowing companies to more easily

identify and define the necessary microservices. Additionally, the updated approach allows for greater flexibility in defining quality constraints. This flexibility allows businesses to tailor the migration process to their specific needs and goals.

The proposed approach still remains conceptual since not all the component has been implemented and validated. In addition, only the performance has been considered as a quality aspect. Chapters 5 will present the semi-automatic approach for the detection of performance antipatterns on a graph based representation of the system. Chapter 6 presents the qualtiy-driven (antipatterns-based) refactoring component validated within the real-world case study presented in Part III.

4.4 Conclusion

This Chapter presented two different quality-driven migration approaches. The first process consists of three phases: system comprehension, microservice identification and assessment, and microservices packaging. Quality attributes are considered in each phase, and the output of the quality attributes is inherited from the previous phase. During the system comprehension phase, an analysis of antipatterns is performed. The aim is to eliminate identified antipatterns not only by applying related patterns during microservice implementation but also by increasing these patterns with tactics that represent precise decomposition strategies. Among all possible decompositions, the one that comes closest to optimal is selected. The Chapter highlights how this process is lengthy and complex and emphasizes the challenges involved. Therefore, a second quality-driven approach based on antipattern analysis is also presented. In this case, the process consists of three phases: migration planning, system decomposition, and migration execution. In migration planning, the system is analyzed, and quality constraints are defined. During the system decomposition phase, antipattern detection is performed. The system is then represented through an augmented graph that allows not only the visualization of classes, methods, and dependencies but also the antipatterns present in the system. The graph-based representation of the system will be analysed in Chapter 5 within its implementation. The output of the microservices identification phase is a unique system decomposition. In the migration execution phase, microservices are implemented, and an assessment is made that can have two different results: the microservice(s) comply with the quality constraints and are implemented, or the constraints are not met, and the system needs to undergo a refactoring process explained in Chapter 6. Both proposed approaches provide guidelines for practitioners.

Chapter 5

Graph-based Software Representation for Antipatterns Detection

This Chapter presents a graph-based approach for modeling software, along with a mathematical formulation for detecting common antipatterns. We provide implementation details and examples of antipattern detection. Our approach accurately identifies potential antipatterns in the system and offers insights for software design improvement.

5.1 System Representation through Graph

The presented graph-based modeling approach visually represents the software system, with nodes representing classes or methods and edges representing their relationships. We created a directed graph $G = (V, E)$ to capture all relations between classes and methods, where V is the set of nodes representing classes and methods, and E is the set of edges representing different types of relationships between them.

5.1.1 Type of nodes

We define the set of nodes V as the union of two disjoint sets: $V = M \cup C$. The set C consists of nodes representing classes, while the set M consists of nodes representing methods. We define these sets as follows:

$C = c_1, c_2, ..., c_n$, where each c_i is a unique identifier for a class in the graph.

$M = m_1, m_2, ..., m_k$, where each m_j is a unique identifier for a method in the graph.

5.1.2 Type of edges

The set of edges E depicts various types of relationships between nodes. Relationships are determined based on the types of nodes involved. Below are the types of edges we have identified along with the relationships they represent:

Relations between classes The investigation primarily focuses on analyzing class relationships to uncover the intricate structure and dependencies inherent in the system, thereby providing valuable insights into its underlying architecture. In this study, we consider two general classes, referred to as c_1 and c_2, both belonging to the set of classes C. We have identified four types of relationships between these classes: `exists` `implements`, `imports`, and `composed_by`.

- `exists`: The `exists` relationship establishes class hierarchies and reveals interdependencies between classes. This relationship occurs when class c_1 is a subclass of class c_2. Figure 5.1 provides an example of the `exists` relationship between c_1 and c_2, represented by a directed arrow, indicating that c_1 exists within the class set C and is a subclass of c_2.

Figure 5.1: `extends` relation between classes

- `implements`: The `implements` relationship provides insights into the interdependencies and interactions within a software system. When class c_1 implements class i_1, the `implements` relationship is established as $(c_1, i_1) \in E$. Please note that interfaces and abstract classes are nodes in the class relationship graph. By establishing the `implements` relationship, class c_1 realizes the specified methods and behaviors encapsulated by class i_1. Figure 5.2 depicts a scenario where class c_1 effectively implements the methods and behaviors specified by interface i_1. The relation is represented by the directed arrow from class c_1 to interface i_1.

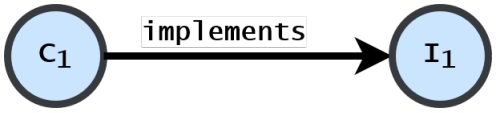

Figure 5.2: `implements` relation between classes

- `imports`: The `imports` relationship represents a dependency between classes, where one class relies on another for specific functionality. If class c_1 imports class c_2, it establishes the `imports` relationship $(c_1, c_2) \in E$ and is represented in

Figure 5.3 by the directed arrow from c_1 to c_2. This relationship highlights the dependency of class c_1 on class c_2 for the necessary functionality.

Figure 5.3: `imports` relation between classes

- `composed_by`: The `composed_by` relationship signifies the composition and aggregation of classes in a system. Note that, if the class c_1 is composed_by the class c_2, than c_1 has an attribute of type c_2. Thus, if class c_1 is composed or aggregated by class c_2, then the edge $(c_1, c_2) \in E$ is created. Figure 5.4 provides an example of the `composed_by` relationship where class c_1 is composed or aggregated by class c_2. The relationship is represented by the directed arrow from c_1 to c_2. Understanding the `composed_by` relationship provides insights into the structural organization of classes in the system, showcasing how they are composed or aggregated to create more complex entities.

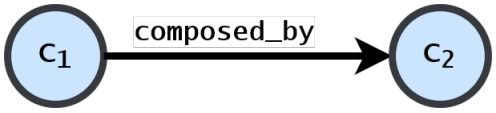

Figure 5.4: `composed_by` relation between classes

Relations between classes and methods Another type of relationships that we analyze are those that exist between classes and methods. These relationships play a crucial role in understanding the interactions and dependencies within a software system. Specifically, we consider four types of relationships: `owns`, `uses_as_var`, `uses_as_arg`, and `returns`. For the sake of exemplification, in the following we consider two classes, $c_1, c_2 \in C \subset V$ and a method $m_1 \in M \subset V$.

- `owns`: The `owns` relationship is essential for organizing and defining software systems. It allows classes to manage and manipulate methods efficiently, leading to coherent designs. When a class owns a method, denoted as $(c_1, m_1) \in E$, it takes responsibility for its implementation and accessibility within its scope. This relationship enables encapsulation, with the class gaining authority over the owned method. Class c_1 owning method m_1 is visually represented in Figure 5.5 as a directed edge outgoing from c_1. and incoming in m_1.

- `uses_as_var`: Understanding the `uses_as_var` relationship between methods and classes is essential for comprehending data flow in a software system. It enables the

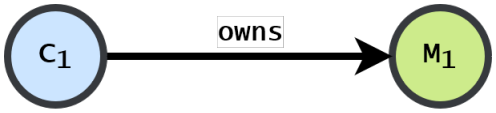

Figure 5.5: Owns relation between classes and methods

effective utilization of method outputs as variables, supporting class functionalities and complex computations. The `uses_as_var` relationship signifies the usage of a method's return value as a variable within a class. If method m_1 defines a local variable of type c_2, this relationship can be represented as $(m_1, c_2) \in E$. Figure 5.6 illustrates an example of the `uses_as_var` relationship, representing a dependency where a class relies on a method's return value for specific operations. Modeling these dependencies provides insights into the interdependencies between classes and methods, facilitating better analysis and design.

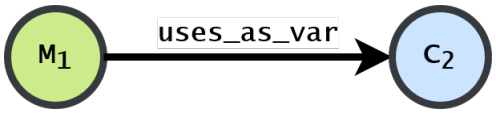

Figure 5.6: `uses_as_var` relation between methods and classes

- `uses_as_arg`: Analyzing the `uses_as_arg` relationship, along with other class-method relationships, enhances our understanding of how classes depend on method parameters to achieve desired functionalities. It reveals interactions and dependencies within a system, contributing to software architecture comprehension and design. The `uses_as_arg` relationship involves the utilization of method parameters within a class. If method m_1 has a parameter of type c_2, we represent this relationship as $(m_1, c_2) \in E$. It signifies the dependency between a method and a class, where the class relies on a specific parameter of the method to fulfill functionalities. By capturing and analyzing these dependencies, we gain a deeper understanding of the interactions and interdependencies between classes and methods in a system. Figure 5.7 visually represents an instance of the `uses_as_arg` relationship, showing a directed edge from method m_1 to class c_2 to symbolize their dependency. This edge visualizes the flow of information or data from the method to the class, highlighting the significance of the method's argument in the class's functionality.

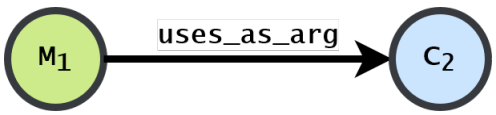

Figure 5.7: `uses_as_arg` relation between methods and classes

- `returns`: Understanding and analyzing the `returns` relationship aids in comprehending how methods generate specific object or class outputs. It reveals dependencies and interactions within a system, contributing to software architecture comprehension and design. The `returns` relationship denotes the return type of a method, which can be an object or a class. If method m_1 returns an object of type c_2, we represent this relationship as $(m_1, c_2) \in E$. It signifies the dependency between a method and an object or class, where the method produces an output of type c_2 upon execution. By capturing and analyzing these dependencies, we gain a deeper understanding of the relationships between methods and their return types. Figure 5.8 provides an example of the `returns` relationship, illustrating the connection between method m_1 and object c_2, where m_1 returns an object of type c_2. This visual representation emphasizes the dependency between the method and its return type, illustrating their connection.

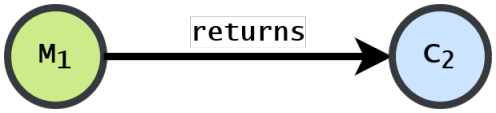

Figure 5.8: `uses_as _var` relation between methods and classes

Relations between methods In the scope of graph representation for Java systems outlined in this Chapter, we have specifically focused on the `calls` relationship as the sole relationship between methods that is crucial for our objectives. The `calls` relationship denotes the invocation of one method by another method within the system. The `calls` relationship as the primary connection that adequately represents the method-level dependencies essential to achieving our objectives. By leveraging this relationship, we can establish a comprehensive understanding of how methods interact and rely on one another within the Java system. If we have two methods, $m_1, m_2 \in M$, where m_1 calls m_2, we represent this relationship as $(m_1, m_2) \in E$. This relationship captures the fundamental dependencies and interactions between methods, allowing us to analyze the control flow and data flow within the system. Figure 5.9 depicts an example of the `calls` relationship. Thus, if the method m_1 calls method m_2, then a direct edge is created.

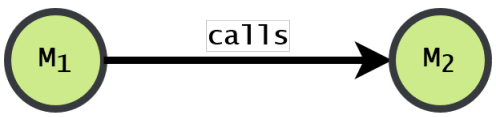

Figure 5.9: `calls` relation between methods

5.1.3 Implementation

The creation of the graph representation of the system has been implemented to capture class and methods relations in Java project. The Java source code files have been parsed by a custom Java source code parser based on *ANTLR*[1] and *Java 9 grammar definition*[2]. In particular, ANTLR has been instructed to generate a *Java lexer* and a *Java parser skeleton* using Python3 as destination language with the following commands:

```
antlr4 -o parser -Dlanguage=Python3 java_grammars/JavaLexer.g4
antlr4 -o parser -Dlanguage=Python3
                              -lib parser/lexer java_grammars/JavaParser.g4
```

The parser skeleton has been then adopted in a custom Python3 script. The script is used to intercept the class definitions, method definitions and all the relations defined in the previous Section to derive a structured representation. For example, starting from the following Java source code file:

```
public class Example {
    public static void main(String[] args)
    {
            method();
    }

    public void method()
    {
            System.out.println("Hello world");
    }
}
```

the script produces the following Python3 data structure:

```
{'Helloworld': {
    'imports': set(),
    'extends': None,
    'interfaces': set(),
    'methods': {
            'main': {
                    'vars': set(),
                    'args': {'String'},
                    'return': None,
                    'use': set(),
                    'calls': {"method"},
                    'varinfo': set()
            },
            'method': {
                    'vars': set(),
```

[1] https://www.antlr.org/
[2] https://github.com/antlr/grammarsv4

```
                        'args': set(),
                        'return': None,
                        'use': set(),
                        'calls': {
                                'System.out::println'
                        },
                        'varinfo': set()
                }
        },
        'composed': set()
}}
```

The script is capable of combining data from multiple Java source files and complete codebases into a single resulting data structure. The final graph is created by using *Neo4J* graph-based DBMS [3] and its official Python3 library.

5.2 Antipatterns Mathematical Formulation

In the forthcoming subsections, we will introduce the mathematical formulations of three specific antipatterns: *God Class*, *Circuitous Treasure Hunt*, and *Empty Semi-Truck*. We decided to analyse those three antipatterns since they are largely discussed in the litterature, giving us the opportunity to evaluate the results of the application of our approach. Each mathematical formulation will be applied on the PetClinic project to exemplify their application. Using the mathematical models, we aim to identify potential instances of these antipatterns to be considered in the refactoring process.

5.2.1 *God Class* Antipattern

The *God Class* antipattern occurs when a class takes on excessive responsibilities and violates the *Single Responsibility Principle* [109][110][111]. The problem arises when a class performs all of the application's work or holds all of its data, leading to increased message traffic and degraded performance. The recommended solution involves redistributing intelligence across top-level classes, achieving a balanced distribution of data and behavior throughout the system. Table 5.1 shows the problem and possible solution as they has been represented in the literature [109].

[3] https://neo4j.com/

God Class	
Problem	**Solution**
Occurs when a single class or component either 1) performs all of the work of an application or 2) holds all of the application's data. Either manifestation results in excessive message traffic that can degrade performance.	Refactor the design to distribute intelligence uniformly over the application's top-level classes, and to keep related data and behavior together.

Table 5.1: God Class - Problem and Solutions

To identify instances of the *God Class* antipattern within the graph representation, a metric C has been devised based on the class nodes' incoming and outgoing edges. The metric, denoted as $C(c)$, quantifies the complexity of a class node $c \in C \subset V$ relative to the total number of nodes $N = |V|$ in the graph. A high number of connections indicates a *God Class* burdened with numerous relationships and functionality. The metric is calculated using the formula:

$$C(c) = \frac{I(c) + O(c)}{N} \tag{5.1}$$

Where:

- $I(c)$ is the number of incoming edges to class node $c \in C \subset V$.
- $O(c)$ is the number of outgoing edges from class node $c \in C \subset V$.
- $N(c)$ is the number of class and method nodes in the graph.

In the context of the modeled graph, the calculation of the metric $C(c)$ considers the following edges:

- `owns`: given a class $c \in C \subset V$, we analyse all its outgoing edges labeled as `owns`. As already described, the `owns` relation connect the class with each of its methods. Thus, we can recognize the number of method in the class. Indeed, a class that contains to many methods have an high probability to expose to many functionalities having high responisibility.
- `import`: given a class $c \in C \subset V$, we count both the incoming and outgoing edges of this type. On one hand, if the class c imports a lot of classes, there is an high probability that it will be used for many funtionalities violating the principle of *single responsibility*. On the other hand, if c is imported into many classes, this means that it is required to performs different operations. This situation may represent a high degree of coupling.

- `implements`: given a class $c \in C \subset V$, that implements interfaces, we consider all the outgoing edges of this type. If a class implements a lot of interfaces, there is an high probability that the class has to many responsibility.

By comparing the calculated values across class nodes, excessively connected classes, known as *God Classes*, can be identified. This metric helps detect and assess *God Classes*, allowing developers to address design issues and improve software quality and maintainability.

To identify instances of *God Classes*, a threshold value T is established. If the calculated metric $C(c)$ for a class node $c \in C \subset V$ exceeds the threshold value ($C(c) > T$), the corresponding class node is classified as a *God Class*. Thus, the presence of the *God Class* antipattern within the graph can be evaluated using the mathematical expression:

$$\forall c \in C,\ \textit{if}\ C(c) > T,\ \textit{then}\ c\ \textit{is a God Class}. \tag{5.2}$$

The threshold value T is specific to the domain and may vary across different projects. By applying this mathematical expression software developers and analysts can effectively detect potential *God Classes* within the graph representation of the system.

Example. In the following we provide an example of the detection of this antipatterns into the PetClinic application. o this purpose, we generated the following Cypher code running on the Neo4j Graph to retreive all the information needed to evaluate the metric $C(c)$. Table 5.2 reports the first six results of the Cypher query.

```
MATCH (c:Class)
RETURN c.name,
    size([o=(c)-[:OWNS]-()|o]) AS ownsOutgoing,
    size([i1=(c)-[:IMPORTS]-()|i1]) AS imports,
    size([i2=(c)-[:IMPLEMENTS]-()|i2]) AS implements,
    (size([o=(c)-[:OWNS]-()|o])
        +size([i1=(c)-[:IMPORTS]-()|i1])
        +size([i2=(c)-[:IMPLEMENTS]-()|i2])) AS totalEdges,
    (size([o=(c)-[:OWNS]-()|o])
        +size([i1=(c)-[:IMPORTS]-()|i1])
        +size([i2=(c)-[:IMPLEMENTS]-()|i2]))/187.0 AS complexity,
ORDER BY (complexity) DESC
```

If we consider the threashold $T > 0.9$, the classes *Owner* and *Pet* reveal the possible *God Class* antipattern. Figure 5.10 highlights all the relationship of type `owns`, `implements`, and `import` in which the class *Owner* is involved.

c.name	ownsOut.	imports	implements	totalEdges	complexity
"Owner"	11	10	0	21	0.11229946
"Pet"	7	13	0	20	0.10695187
"Vet"	6	10	0	16	0.08556149
"OwnerController"	12	2	0	14	0.07486631
"PetController"	10	2	0	12	0.06417112
"Visit"	4	5	0	9	0.04812834

Table 5.2: Results of the Cypher Query for the God Class Antipattern.

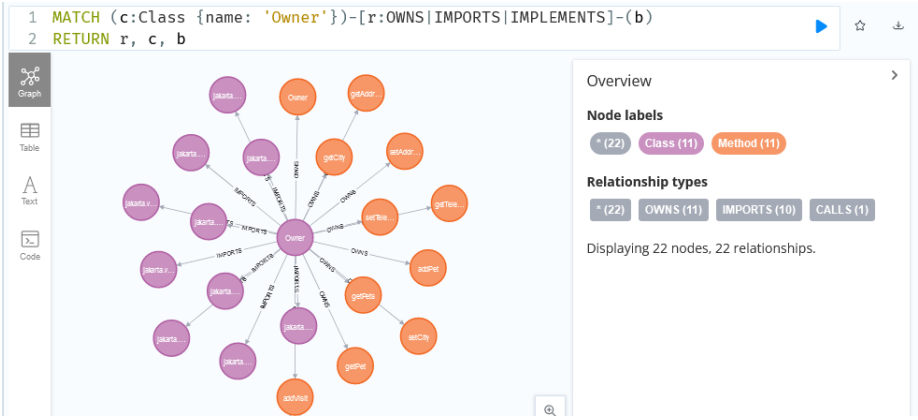

Figure 5.10: *God Class* - Example: Class *Owner*

5.2.2 *Circuitous Threasure Hunt* Antipattern

The *Circuitous Treasure Hunt* antipattern is distinguished by an extended sequence of method invocations or attribute accesses spanning across multiple classes degrading the software performance [109][112][111]. Table 5.3 shows the problem and possible solution for this antipattern [109].

Circuitous Treasure Hunt	
Problem	**Solution**
Occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each look, performance will suffer.	Refactor the design to provide alternative access paths that do not require a Circuitous Treasure Hunt (or to reduce the cost of each look).

Table 5.3: Circuitous Treasure Hunt - Problem and Solutions

To detect the *Circuitous Treasure Hunt* antipattern in the graph representation, we use a metric that measures the length and complexity of method call chains. The metric is based on the idea that a long and convoluted call chain indicates participation in the antipattern. To this end, we introduce the metric $C(m)$ for a method node $m \in M \subset V$,

defined as follows:

$$C(m) = L(m) \times W(m) \tag{5.3}$$

In the above formula:

- $L(m)$ denotes the length of the method call chain originating from the method node m. The recursive definition of $L(m)$ is as follows:

$$L(m) = \begin{cases} 0 & \forall\ m \in M \text{ s.t. } \nexists\ m' \in M \text{ s.t. } (m, m') \in E \\ 1 + \max(L(m')) & \forall\ m' \in M \text{ s.t. } (m, m') \in E \end{cases} \tag{5.4}$$

  Thus, the value $L(m)$ is equal to 0 if m lacks outgoing edges representing method call. Otherwise, the value is set to $1 + \max(L(m'))$ for all m' such that m calls m'.

- $W(m)$ signifies the weight of the method node m, reflecting its complexity. We define $W(m)$ as follows:

$$W(m) = N(m) + 1 \tag{5.5}$$

  Note that $N(m)$ represents the count of outgoing edges from m that pertain to method calls, incremented by one to avoid multiplication by zero in formula (5.3) when m lacks outgoing edges representing method calls. Note that $W(m)$ is necessary to avoid that wrapper and helper function are wrongly detected as *Circuitous Threasure Hunt*. Thus, the parameter is used to improve the trustability of the formula.

Considering the graph-based representation of the system, this antipattern can be recognized on a method $m \in M \subset V$ by analysing its outgoing edges of type `calls`.

For the purpose of identifying the *Circuitous Treasure Hunt* antipattern, a threshold value T is established. If the calculated metric $C(m)$ for a method node m exceeds the predefined threshold value ($C(m) > T$), the corresponding method node is considered to be part of the antipattern. Consequently, we formulate the mathematical expression to assess the presence of the *Circuitous Treasure Hunt* antipattern within the graph as follows:

$$\forall m \in M,\ \textit{if}\ C(m) > T,\ \textit{then}\ m\ \textit{is part of the Circuitous Treasure Hunt}. \tag{5.6}$$

Note that the threshold value T is context-dependent and may vary based on the specific characteristics of the project. The utilization of this mathematical expression empowers software developers and analysts to effectively detect instances of the *Circuitous Treasure Hunt* antipattern within the system's graph representation.

Example. In the following we provide an example of the detection of this antipatterns into the PetClinic application. To this purpose we created the Cypher query shown in

Figure 5.11: *Circuitous Treasure Hunt* - Example

Figure 5.11. The query is performed to evaluate the parameter $L(m) = 4$. Considering, the number of outgoing edges of type `calls` of the method m *processNewVisitForm*, its weight results to be $W(m) = 3$. Thus, in this case the metric is equal to $C(m) = 4 \times 3 = 12$.

On the other hand, the same metric for the lenght of the method m' *loadPetWithVisit* is $W(m') = 2$. Thus, in this case the metric results to be $C(m') = 4 \times 2 = 8$. If we consider the threashold $T = 8$, then we can consider the method m named *processNewVisitForm* as a part of a potential *Circuitous Threasure Hunt*.

5.2.3 *Empty Semi-Truck* Antipattern

The *Empty Semi-Truck* antipattern occurs when a class lacks attributes or methods and serves solely as a namespace or grouping mechanism [109][110]. It represents a situation where the class lacks functionality and is used only for organizing or categorizing elements. The problems and solutions related to the Empty Semi-Truck are presented in Table 5.4 [109].

Empty Semi-Truck	
Problem	**Solution**
Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient in terface, or both.	The Batching performance pattern combines items into messages to make better use of available bandwidth. The Coupling performance pattern, Session Facade design pattern, and Aggregate Entity design pattern provide more efficient interfaces.

Table 5.4: Empty Semi-Truck - Problem and Solutions

To identify this antipattern within the graph-based representation of the system, we introduce a metric that gauges the emptiness of a class node. The emptiness metric

$E(c)$ for a class node c is defined as follows:

$$E(c) = F(c) + M(c) \tag{5.7}$$

In the provided formula, $F(c)$ denotes the number of attributes associated with the class node, whereas $M(c)$ signifies the number of methods associated with the class node c.

To evaluate if a class c represent this antipattern, we evaluate two types of edges:

- `composed_by`: the sum of the outgoing edges of this type, represents the number of field of the class $c \in C \subset V$.
- `owns`: the number of outgoing edges of this type, represents the number of methods associated to the node class $c \in C \subset V$.

To detect the *Empty Semi-Truck* antipattern, a threshold value T is established. If the calculated emptiness metric $E(c)$ for a class node c falls below the predefined threshold ($E(c) < T$), the class node c is identified as part of the potential antipattern. Thus, the related mathematical expression is formulated as follows:

$$\forall c \in C, \textit{ if } E(c) < T, \textit{ then } c \textit{ is part of the Empty Semi-Truk.} \tag{5.8}$$

Example. In the following we provide an example of the detection of this antipatterns into the PetClinic application. To this purpose we created the following Cypher query to extract all the `composed_by` and `owns` relationship in wich the class c is involved. Thus, the metric $E(c)$ has been evaluated by following the proposed formula. Table 5.5 shows the first six results of the created query.

```
MATCH (c:Class)
RETURN c.name,
    size([o=(c)-[:COMPOSED_BY]-()|o]) AS compositionsNumber,
    size([i1=(c)-[:OWNS]-()|i1]) AS methodsNumber,
    size([o=(c)-[:COMPOSED_BY]-()|o])
        +size([i1=(c)-[:OWNS]-()|i1]) AS complexity
ORDER BY complexity
```

5.3 Related Work

In recent years, there has been a growing interest in the field of software engineering to develop effective techniques for refactoring and improving the quality of software systems. In this context two types of studies can be identified. On one hand there are scientific works addressing the challenge of software comprehension through graph-based representation of the software. On the other hand, researchers have explored various

c.name	compositionsNumber	methodsNumber	complexity
"Visit"	1	0	1
"OwnerController"	1	0	1
"PetType"	1	0	1
"SetVisit"	1	0	1
"PetController"	1	0	1
"PetTypeFormatter"	1	0	1

Table 5.5: Results of the Cypher Query for the Empty-Semy Truck Antipattern.

approaches to address common code smells and antipatterns in software projects. In this section we will present the main contributions in the field, comparing them to our work.

5.3.1 Graph-Based Representation of Object-Oriented Projects

In the realm of representing dependency relationships, various language-specific graph variations have been proposed for C++ [113][114][115][116] and Java programs. These variations aim to capture the interdependencies among elements within the program. One such representation is the call-based object-oriented system dependence graph (COSDG), which incorporates additional annotations to account for calling context and method visibility details. Several studies have employed and advocated for the use of COSDG in their research [117][118]. Another notable representation is the Java software dependence graph (JSDG), which comprises multiple dependence graphs that depict control, data, and call dependencies among different program elements. JSDG has gained extensive usage within the Java context [119]. In order to enhance the capabilities of JSDG, Zhao proposed an augmented version known as the Java system dependence graph (JSDG+), which includes a specialized mechanism to handle polymorphism and interfaces, thereby improving the representation of dependencies in Java programs [120]. Building upon JSDG, JavaPDG provides a static analyzer for Java bytecode as well as a browser for visualizing various graphical representations, such as the procedure dependence graph, system dependence graph, control flow graph, and call graph [121][119]. Additionally, an improved version of JavaPDG called jpdg focuses specifically on enhancing the representation of program dependence graphs (PDGs) for code mining purposes [122][121].

Our proposed tool for graph-based representation carefully analyzes the relationships between classes and methods. In contrast to the existing works, our approach utilizes a simpler notation and focuses solely on capturing the essential elements required for detecting antipatterns. As a result, our approach achieves easier graph generation and, unlike other tools, it has been specifically designed for the purpose of antipattern detection.

5.3.2 Antipatterns Detection

Authors in [123] introduce MicroART, an Architecture Recovery Tool systems based on microservices. Utilizing Model-Driven Engineering principles, this tool generates software architecture models for microservice-based systems. These models can be managed by software architects to support system maintenance and evolvability. In [124] the PADRE tool is presented, which detects performance antipatterns in UML models. The tool also applies refactoring techniques to eliminate identified antipatterns from the UML models. Similarly, in [125], the authors propose a technique for enhancing the quality of use case models, demonstrated using a real-world system. This method detects antipattern defects in use case models and automatically refactors them through appropriate model transformations. Authors in [126] propose an algorithm that analyzes multiple architectural antipatterns representing the modularization problems to identify refactoring opportunities. The recommendations aim to minimize changes that have a significant impact on the overall system quality. In the study presented in [127], authors provide a systematic process, to identify and resolve performance issues with runtime data by using load testing and and data profiling. Similarly, authors in [128], presents a tool for antipatterns detection that uses Java profilers allowing to perform the dinamic analysis of the system. It is noticeable to present two industrial project: *Arcan*[4] and *Designate*[5]. The first, helps discovering the architectural debt to prevent its accumulation. The second, identifies architecture smells and visualize them. Each detected smell is presented with its definition and its cause.
Our work differs from the presented because of the following advantages:

- Our approach does not require the use of UML models, which can be limiting. Nowadays, with the increasing adoption of agile development processes, there is a lack of documentation. This often results in the absence of pre-existing UML models, which need to be regenerated. Even though automatic generation of such models is possible, it still requires the intervention of a software architect to ensure their correctness. In contrast, our system relies solely on the code, reducing the need for software architect intervention.

- The dynamic analysis of the system facilitates the detection of antipatterns as the system is simulated, producing more accurate results. However, with the advent of cloud architectures, dynamic system analysis becomes more complicated. Simulating the behavior of the system becomes challenging because a single application can be executed simultaneously on multiple cloud nodes. This makes the analysis of results more difficult, as they are influenced by uncontrollable external factors. Our approach, on the other hand, focuses solely on the internal structure of the system, bringing to light the most common antipatterns that have

[4]https://www.arcan.tech/
[5]https://www.designite- tools.com/

been deemed significant and impactful on performance within the scientific and industrial community.

5.3.3 Open Challenges of the Approach

This section highlights potential limitations that affect our study.

- Limited coverage of antipatterns: although the graph representation presented in this study is sufficiently general to detect various antipatterns, its coverage is still limited. Additionally, not all antipatterns can be identified solely through static analysis; dynamic analysis is required. To overcome these limitations, we plan to expand the range of antipatterns that our tool can detect by i) carefully analyzing new antipatterns to create a valuable mathematical formulation, ii) incorporate dynamic analysis capabilities to annotate the graph, thereby potentially increasing the number of detectable antipatterns.

- Manual intervention for antipattern detection: while we employed static analysis techniques to automatically construct the graph representation, our approach does not include an automated antipattern detection mechanism. As a result, the architect's assistance is necessary to identify the antipatterns by manually creating Cypher queries. By highlighting this limitation, we emphasize the need for developing an automated tool based on our methodology. This would not only validate its effectiveness but also enhance its practical usage, as it would eliminate the reliance on manual intervention and make the detection process more efficient.

- Lack of comparative tools for validation: as discussed various tools have been proposed in both academic and industrial contexts. However, we were unable to directly compare our work with these tools for a significant reason: unlike other existing approaches, our methodology relies solely on static analysis. Consequently, we cannot evaluate the accuracy of our results through a comparative analysis.

- Specificity to Java projects: Although the methods for representing software using graphs and detecting antipatterns mathematically are universally applicable to Object Oriented languages, the tool designed for automatic graph generation is limited to Java projects. This restriction arises from the tool's reliance on a parser developed specifically for the Java 9 grammar. Therefore, while the general approaches and mathematical formulations are transferable, the tool's functionality is confined to Java-based systems.

5.4 Conclusion

In this chapter, the approach to modeling legacy systems was presented. This approach represents one of the task related to the microservices identification phase of the quality-driven migration approach shown in Chapter 4. The graph representation considers nodes of class and method types, and edges that represent the various possible relationships between them. An implementation of the representation was shown using *ANTLR* based on *Java parser* and *Neo4j*.

The chapter presented the mathematical formulation that allows the detection of three antipatterns: *God Class*, *Circuitous Threasure Hunt*, and *Empty Semi-Truck*. For each antipattern, an example of detection on a case study was reported. The graph representation and ANTLR-based parser can facilitate the visualization and manipulation of complex legacy systems, making it easier to identify antipatterns and dependencies that might otherwise be difficult to discern.

Chapter 6

Quality-Driven Refactoring Approach

The current Chapter presents a quality-driven refactoring approach, which corresponds to the refactoring activity described in Chapter 4 of the quality-driven migration approach. The approach is novel compared to existing refactoring processes in the field, as the antipattern detection is not limited to the microservice under refactoring but also covers the monolith it originates from. Although the relationships between antipatterns and quality metrics have been extensively explored in the literature, to the best of our knowledge, the relationships between antipatterns themselves have not been investigated yet. The validation of the proposed process will be presented in Part III.

6.1 Related Work

Refactoring is defined as *the process of improving the design of existing code by changing its internal structure without affecting its external behaviour* [4]. Despite there is explicit assumption that software refactoring improves the quality of the software, a lot of effort has been made by researchers to investigate their correlation. In [129], the authors present a hierarchical quality model to study the effect of software refactoring on the quality model concerning reusability, flexibility, extendibility and effectiveness. The results show that despite the majority of the refactoring techniques adopted do improve quality, some approaches have a negative impact. The goal of the authors in [130] is to quantitatively assess the effect of refactoring on different external quality attributes, which are: adaptability, maintainability, understandability, reusability, and testability to decide if the cost and time put into refactoring are worthwhile. Even in this case, the results show that refactoring does not necessarily improve these quality attributes. In [131] the authors analyse source code version control system logs of open-source software systems. The aim was to detect changes to examine their impact on software metrics. In this case, the results suggest that: i) the refactoring process does not always improve the

software quality in a measurable way and ii) developers do not use refactoring effectively as a means to improve the quality of the system. A similar approach has been adopted in [132]. In this work, the authors mine the history of three Java open-source projects. The goal is to investigate whether refactoring activities occur on code components for which quality metrics might be needed for refactoring operations. Their results pointed out that there is no clear relationship between refactoring and quality metrics.

In the literature, architectural smells and antipatterns have been largely considered in the refactoring scenario. In fact, software development is a complex process that involves solving various problems during the design and development phases. To simplify this process, designers and developers use design patterns and antipatterns as guides to identify common solutions and pitfalls. Design patterns are *solutions to common problems* that have emerged through experience in software development. They provide a general solution that can be adapted to a specific context. Best practices for software system design are derived from the knowledge gained through design patterns. This enables the reuse of such knowledge and its application in the design of different types of software systems. Design patterns can be applied to different categories of problems and solutions related to software development, including architecture, design, and development. On the other hand, antipatterns are the opposite of design patterns. They document *recurring solutions to common design problems* that are often suboptimal and lead to negative consequences. Antipatterns are conceptually similar to patterns in that they document the "bad practices" applied during software development, as well as solutions to them. Antipatterns provide indications on what to avoid and how to correct design problems if they are encountered. Like patterns, antipatterns address both architectural and design problems and can be applied to the development process. In [133], authors present a methodology to systematically identify the architectural smells that possibly violate the main design principles of microservices and to select suitable architectural refactorings to resolve them. An Architecture Recovery Tool for microservice-based systems called MicroART is presented in [123]. This tool using Model-Driven Engineering principles generates models of the software architecture of a microservice-based system, that can be managed by software architects for multiple purposes supporting the maintenance and evolvability of the system. The authors in [124] present the PADRE tool. It permits the detection of performance antipatterns in UML models. In addition, the tool refactors the UML models removing the detected antipatterns. A similar approach is presented in [125]. In this paper, the authors propose a technique for improving the quality of use case models and demonstrate it on a real-world system. The method detects defects, in terms of antipatterns, in a use case model and automatically performs improvements by refactoring the use case models through proper model transformations. The authors in [126] propose an algorithm that, based on the simultaneous analysis of multiple architectural antipatterns, provides high-impact refactoring opportunities. The suggestions aim to reduce the number of

changes having a major impact on the overall quality of the system. The algorithm has been then validated on 95 open-source Java programs for instances of four architectural patterns representing modularisation problems.

Our work propose a quality-driven refactoring approach supporting the refactoring of microservice derived from legacy systems. Thus, it is interesting to investigate whether researchers and practitioners consider quality aspect to drive the migration to microservice. As stated before, a non-quality-driven process would entail the need to refactor the developed microservices to respect software quality constraints, which would justify the need for our proposed approach. The results of our investigation shown that few works consider quality constraints during the migration to microservices. We briefly report some of them. In [88] authors propose an approach to identify microservices from OO source code in an automatic way. A quality function is used to measure both the structural and behavioural validity of microservices and their data autonomy. Authors in [57], incorporate security and scalability requirements in their conceptual methodology of decomposing systems into microservices. In [92], authors proposed an automated microservice identification (AMI) approach that extracts microservices considering both functional and non-functional metrics.

In summary, the analysis of the state of the art revealed that:

- Refactoring does not necessarily allow software quality improvements.
- Antipatterns analysis revealed to be a suitable approach for software refactoring aiming to improve software quality aspects.
- Quality improvement is not the main goal when considering migration to microservices. Thus, a quality-driven refactoring approach may be needed.

Antipatterns analysis demonstrated to be a good choice when deciding to perform software refactoring considering software quality. In addition, despite the number of related works found, none of them refers to the refactoring of microservices derived from legacy systems. The novelty of our quality-driven refactoring approach consists of the application of antipatterns analysis in both monolith and microservices in the scenario of refactoring microservices decomposed from a legacy system with the aim to achieve predefined performance requirements.

6.2 Proposed Quality-Driven Refactoring Process

Microservices revealed to be a powerful architectural style for monolithic system modernization through migration. A microservice based system has a dynamic nature due to its continuous integration and delivery. Unfortunately, often this can lead to design and implementation decision that introduce poorly design solutions: the antipatterns. Those antipatterns may affect the maintainability and other quality aspects of the

system [134]. Therefore, a quality-driven refactoring process should consider the antipatterns affecting the system. Those shall be restrict to the quality constraints the architect wants to improve or achieve. In the following we present a quality-driven refactoring approach based on antipatterns analysis that can be applied for the refactoring of microservices migrated from legacy systems regardless of whether these have been deployed or not. Please note that the refactoring process is an essential component of the migration process from monolithic systems to microservices, as discussed in Chapter 4, and it considers both of these software architectures. Nevertheless, the process has been designed in a generalized manner to facilitate its adoption and adaptability for architectural refactoring based on various architectures. An overview of the process is depicted in Figure 6.1 where a BPMN model is presented. According to the BPMN notation, the *'+'* symbol represents parallel activities. On the other hands, the *'x'* notation denotes two (or more) alternative activities.

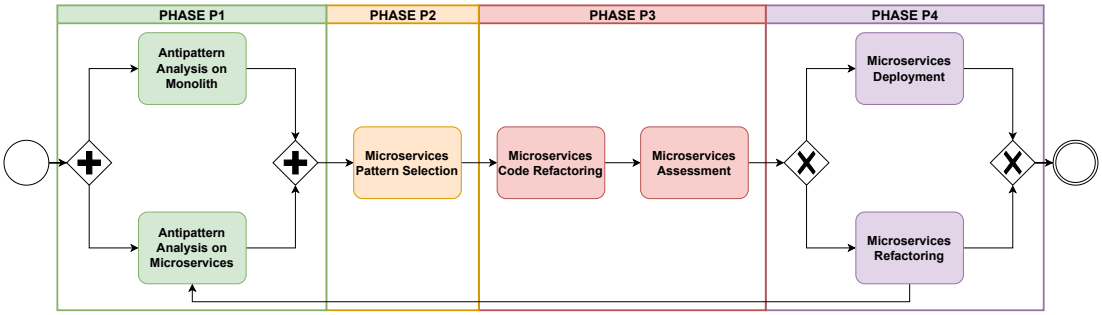

Figure 6.1: The Proposed Quality-Driven Refactoring Approach.

The presented quality-driven refactoring approach takes in input both the monolith and the derived microservices and consists of four phases. The first phase focuses on analyzing antipatterns in the monolith and its corresponding microservices. In this phase, various relationships between the antipatterns are analyzed. In fact, considering those relations in the refactoring process enables us to gain comprehensive insights, facilitate a seamless transition, transfer knowledge, and adapt best practices. The second phase aims to select patterns that help resolve the antipatterns that degrade the quality of the system. Once again, the selection of these patterns is not only based on the microservice antipatterns, but also on those present in the monolith. The third phase corresponds to code refactoring and assessment of the microservice. In the fourth and final phase, based on the assessment results, a decision is made whether to proceed with deployment or iterate the refactoring process. The four constituent phases are described in the following subsections.

6.2.1 Phase 1: Antipatterns Analysis on Monolith and Microservices

The first phase of the proposed quality-driven refactoring approach consists of two macroactivities. Firstly, antipattern detection is performed on both monolith and mi-

croservices. Secondly, we analyze whether there is a relation between the antipatterns detected on the monolith and the microservices.

In particular, we analyze the relationships between two sets of antipatterns: one identified in the monolithic system and the second set specific to the microservices-based application. In the following we discuss the main reason for consider both the subset and not only the antipatterns affecting the microservices:

- Comprehensive understanding: By examining the antipatterns in both the monolithic system and the microservices-based application, we develop a comprehensive understanding of the potential design flaws or architectural limitations present in the system. This analysis helps us identify areas that require improvement and optimization.

- Targeted refactoring: Analyzing the antipatterns in the monolithic system allows us to identify the areas where the original architecture might have hindered scalability, maintainability, or flexibility. By understanding these issues, we can focus our refactoring efforts on those specific aspects to enhance the overall quality of the microservices architecture.

- Proactive issue resolution: Studying the antipatterns specific to the microservices-based application helps us proactively address any potential challenges or pitfalls that might arise due to the decomposition process or the complexities of the microservices architecture. This analysis enables us to identify patterns that could lead to suboptimal design choices or performance issues, allowing us to rectify them during the refactoring process.

- Optimal adaptation: By examining the relationships between the antipatterns in the monolithic system and the microservices-based application, we can adapt best practices and strategies to ensure optimal refactoring. This analysis helps us leverage past learnings and experiences to identify the most effective approaches for improving the microservices architecture. Additionally, by studying the antipatterns, we can also identify worst practices to avoid during the migration process. This comprehensive analysis allows us to design a set of guidelines encompassing both the best practices to adopt and the worst practices to steer clear of during the migration to microservices.

In this context, we distinguished three different cases:

1. **Intersection between antipatterns**: The migration strategy did not resolve the issues with the monolith, as at least one antipattern that was present in the monolith was also found in the microservice. Figure 6.2 provides an example, illustrating two sets of antipatterns. The first set includes the antipatterns detected

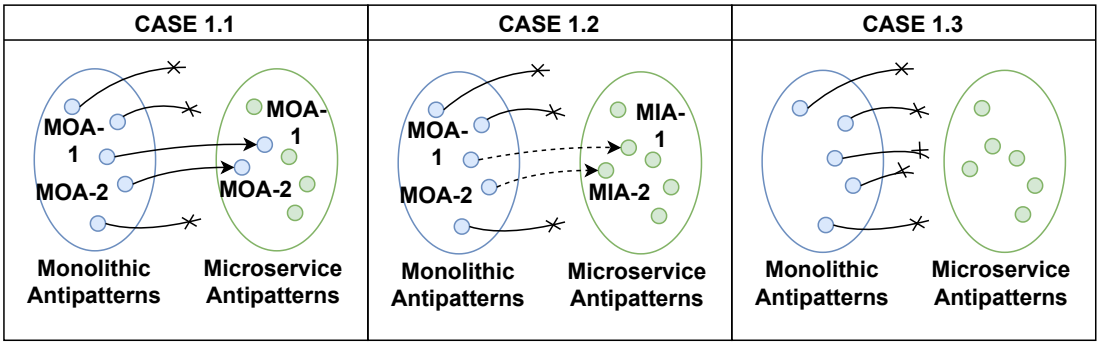

Figure 6.2: Antipatterns in Migration to Microservices: An Example.

in the monolith, while the second set corresponds to the antipatterns identified in the microservices-based system. Case ***1.1*** is represented by the two monolithic antipatterns MOA-1 and MOA-2, which persist in the microservices-based system after migration. In Figure 6.2, the lined arrows indicate that the antipatterns are still present in the microservices-based system, while the crossed arrows indicate that the antipatterns are eliminated with migration.

2. **Disjoint sets of antipatterns - cause-effect relation**: There are no antipatterns found in the microservice that were detected in the monolith. In this scenario, it is advisable to investigate if any of the antipatterns identified in the monolith are responsible for the emergence of one or more new antipatterns in the microservice. Migration strategies and decisions can potentially give rise to fresh antipatterns in the migrated microservices. Figure 6.2 illustrates this situation. In Case ***1.2***, MOA-1 and MOA-2 related to the monolith antipattern set, evolved into MIA-1 and MIA-2 of the microservices respectively. The dotted arrows indicate the evolution of one antipattern into another. An example of a cause-effect relation between antipatterns will be provided in III while the process will be applied on a real-world case study.

3. **Disjoint sets of antipatterns - none relations**: There are no antipatterns that are present in both sets of antipatterns. Additionally, none of the antipatterns that were identified in the monolithic system resulted in any antipatterns in the MSA. This scenario can be seen in Case ***1.3*** of Figure 6.2, where all the arrows from the monolithic to microservices antipatterns set are shown as crossed.

6.2.2 Phase 2: Resolutive Patterns selection

In the second phase of our quality-driven refactoring approach, we select patterns that are designed to address the issues identified through the detection of antipatterns in both the monolithic and microservices systems. The patterns selection depends on the relation between antipatterns resulting from the first phase of the process. Therefore, the possible selection strategies fall into the following cases:

1. **Intersection between antipatterns**: Referring to Case ***1.1*** depicted in Figure 6.2, the primary objective is to address the set of antipatterns that are shared between the monolithic and microservices systems. In such cases, the selection of patterns is limited to those that assist in resolving the common antipatterns.

2. **Disjoint sets of antipatterns - cause-effect relation**: This scenario pertains to Case ***1.2*** of the first phase of the refactoring process, wherein there is no shared antipattern between the monolithic and microservices systems. However, if the antipattern analysis has identified that an antipattern in the monolithic system caused a distinct antipattern in the MSA, the pattern selection should be focused on those that can resolve both antipatterns.

3. **Disjoint sets of antipatterns - none relations**: This case refers to case ***1.3*** where, no common antipatterns are found and no monolithic antipatterns caused a different microservice antipattern. Thus, the idea is to restrict the patterns selection to the ones allowing to solve just the microservices antipatterns.

Table 6.1 summarize the pattern selection strategies based on the relationship found between antipatterns detected in the monolith and the microservices.

Antipatterns Detected	***Pattern Selection Strategy***
The antipattern MOA-1 is detected on both monolith and micorservices.	Select a pattern that resolve MOA-1.
The antipattern MOA-1 detected on the monolith caused the presence of MIA-1 in the microservices.	Select the patterns that resolve MOA-1 and MIA-1.
There are no relationship between the antipatterns of the monolith and the antipatterns of the microservices.	Select the patterns that allow to resolve the microservices antipatterns.

Table 6.1: Antipatterns Relationship and Patterns Selection Strategy

6.2.3 Phase 3: Code refactoring and assessment

During the third phase of the quality-driven migration process, the focus is on refactoring the microservice using the patterns that were selected in the previous phase. Before applying the selected patterns, a preliminary analysis is conducted to determine their suitability with respect to the microservice's code structure and estimated applicability cost. This analysis helps to identify potential issues that may arise during the refactoring process. By taking these factors into consideration, the team can ensure that the selected patterns are the best fit for the microservice and that their application will result in an improved overall system. Since the primary goal of the refactoring process

is to satisfy software quality constraints, an assessment is performed to evaluate the impact of the changes on the overall system. This assessment process ensures that the refactored microservice meets the required quality standards without negative effects.

6.2.4 Phase 4: Microservice deployment or refactoring

The fourth phase of the quality-driven refactoring process leads to one of two possible outcomes upon assessment. The first outcome is that the microservices meet the quality constraints, and no further action is required. In such a case, the microservices can be deployed as is.

However, if the quality constraints are not met, another round of refactoring is necessary. In this scenario, the antipattern detection process is performed solely on the microservices since the antipatterns present in the monolithic architecture are already known. During the microservices refactoring process, new antipatterns may arise, and thus, it is crucial to conduct another round of antipattern detection. On the other hand, the relationship between the antipatterns present on the monolith and the ones detected in the microservices must be analysed according to the process. The remaining phases are performed as already shown.

6.3 Conclusion

This Chapter introduced the quality-driven refactoring process as part of the quality-driven migration approach presented Chapter 4. The novelty of this process was highlighted in comparison to the state of the art, and the process was explained in each of its four phases. The first phase considers the analysis of antipatterns in both the monolith and microservices. Various relationships that may occur between the antipatterns found in the monolith and those related to microservices were explained. Three possible relationships were analyzed: i) one or more antipatterns in the monolith also appear in microservices, ii) one or more antipatterns in the monolith cause the emergence of one or more antipatterns in microservices, iii) no relationship is identified.

In the second phase, resolution patterns are selected. The resolution strategy depends on the relationships found in the previous phase. Specifically, if one or more antipatterns in the monolith are also found in microservices, the strategy involves resolving only the common antipatterns. Alternatively, if one or more antipatterns in the monolith cause one or more antipatterns in microservices, the idea is to resolve both antipatterns. In fact, one of the triggering causes of this situation could be an incorrect design choice made to solve an antipattern in the monolith, which led to the introduction of another antipattern in microservices. A concrete example of this case will be shown in Part III of the Thesis. Finally, if the sets of antipatterns are disjoint, the classic solution is to resolve only the antipatterns present in microservices.

The third phase of the quality-driven refactoring process involves the actual refactoring of microservices using the selected patterns. The microservices are then subjected to an assessment process that can have two outputs: i) the microservices meet the required system quality, ii) the microservice does not comply with quality constraints. Based on this result, it is decided whether the system will be deployed in the fourth phase or whether the process needs to be repeated.

Part III

Industrial Application

Chapter 7

Case Study: BIM Italia

Planning the migration of a system to microservices is a very delicate activity. On one hand, an accurate migration planning supports the creation of a microservices-based system faithful to the original while satisfying all the functional and non-functional requirements. On the other hand, poor migration planning can slow down the modernization process as it requires more iterations to meet requirements. A real-world example comes from BIM Italia. Their migration planning was conducted by dividing the system functionalities according to dependencies and risks. Despite the adoption of the best practices, the company encountered performance issues on the migrated microservices. This Chapter presents the BIM Italia case study, focusing on its product, the migration approach and the related issues on the first two microservices deployed.

7.1 Migration to Microservices: Motivations and Planning

BIM Italia is a company operating in the Healthcare sector offering solutions that integrate software products and professional services. The company is experienced in evaluating and measuring production in the healthcare sector. BIM Italia counts among its clients Hospital Companies, Local Healthcare Companies, and private hospitals in the national territory. The most relevant software of BIM Italia is called QuaniSDO. This system helps the clients in the monitoring of hospital discharge flows before they are reported to the regional health authority.

7.1.1 Motivations

In recent years, the company has decided to make an architectural refactoring of the QuaniSDO software moving to microservices-based system. The QuaniSDO product is a legacy system based on monolithic architecture.

The software has more than ten years and has gone through several changes. The reasons are that i) the application domain is continuously evolving as the local authority's rules are frequently changing and ii) new features are deployed according to

the customer's needs. Thus, the monolithic system encountered maintenance problems quantified by analyzing related time, costs, and effort. This analysis considered the revisions that might need to be applied to the monolith to implement the annual changes requested by the regional authorities. Furthermore, customers often requested to purchase only part of the product limited to certain features. Unfortunately, due to the monolithic nature of the system, this was not possible and caused economic losses. To tackle these problems, the company decided to modernize the system. They decided to design the new version of the QuaniSDO using a MSA. Thus, the company conceived a system whose microservices could be sold separately reducing its maintenance time, cost, and effort over time. The new version of the system would also be released to old customers. Thus, the migration approach was constrained to keep the response time of the new system comparable to the one of the legacy system. Otherwise, the old customer could have complained about a slowdown in operations.

7.1.2 Planning

The transition to a MSA is a multifaceted process that necessitates meticulous planning and execution. This process affects not just the product, but the organization as a whole. The move from a monolithic architecture to microservices entails breaking down the system into smaller, independently deployable components, which necessitates changes to the development process and could even require adjustments to the organizational structure and development team [135]. In a monolithic architecture, the development team might be organized to match the system's structure, with different teams managing different parts of the application. In contrast, microservices teams could be organized around particular microservices, with each team accountable for building, deploying, and maintaining their own microservice [135]. While this can lead to a more decentralized and flexible development process, it also necessitates changes to team structure and management. In summary, migrating to a MSA necessitates careful consideration of various technical and organizational factors, and it is critical for companies to approach this process with a clear understanding of the potential impacts and resource requirements needed to make the transition to a MSA successfully.

In order to successfully migrate to a MSA, BIM Italia conducted a thorough analysis of the necessary steps and technologies to be used, taking into account not only the technical aspects but also the organizational structure of the team. The planning of the migration process was carefully divided into several stages, each of which was essential to ensure a smooth transition. Firstly, the team received training on MSAs, including the principles and best practices associated with this approach. This training was designed to ensure that the team had a thorough understanding of the benefits and challenges of microservices, as well as the technical skills required to implement this architecture. Secondly, the team was divided according to their technological and domain skills. This

ensured that each member of the team was working on tasks that aligned with their areas of expertise, and that the team as a whole had the necessary skillset to effectively manage the migration process. Thirdly, the system was functionally decomposed into smaller, independently deployable components. The team analyzed the system's dependencies and made any necessary adjustments to ensure that each microservice could be developed, deployed, and maintained independently. This involved making changes to the codebase, as well as modifying the team's development processes to accommodate the new architecture.

Table 7.1 provides a comprehensive list of technologies employed for different aspects of the MSA. The table includes the backend technologies used for the microservices, the tools utilized for automating the front-end development process, the frameworks for executing JavaScript code, the packet managers used for managing JavaScript packages, the database management systems used for data storage and retrieval, the containerization tools used for packaging the microservices, the orchestrators for automating the deployment and scaling of containers, and the cloud platforms for hosting and managing the MSA.

Context	**Technology**
Backend	Java (OpenJDK 11)
Build automation tool	Grandle 6.2.1
Web development framework	Angular 8.0
JavaScript runtime	NodeJS 10.16.0
JavaScript Packet Manager	npm 6.9.0
DBMS	PostgreSQL 11 / OracleDatabase
Containerization	Docker
Orchestrator	Kubernetess
Cloud platform	AWS: Elastic Kubernetess Services

Table 7.1: Implementation Technologies

Figure 7.1 displays a diagram showing six microservices identified by the company. Each of these microservices is centered around the Hospital Discharge Record (HDR), which serves as the core of the computation. The three microservices considered in this Thesis are highlighted in yellow in the diagram, providing a clear visual representation of the components that are the focus of our study.

Henceforth, the microservice formerly known as "Formal Logic Controls" will be referred to as "Control" for the sake of simplicity.

After defining a set of microservices to be implemented, the company started a prioritization process that took into account several factors. Firstly, they analyzed the complexity of implementing each microservice, taking into account the required resources,

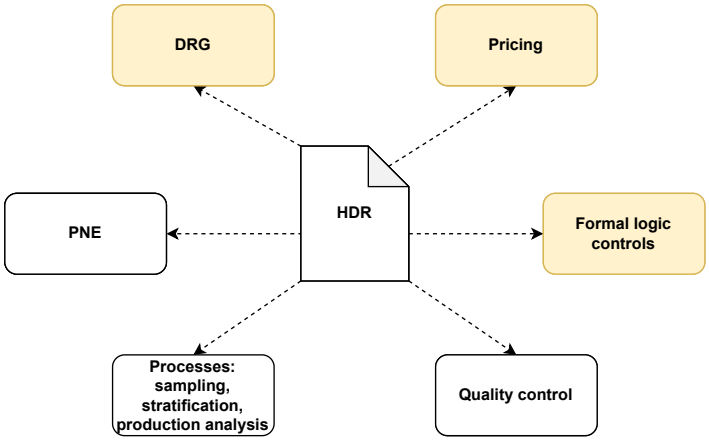

Figure 7.1: The *QuaniSDO* Defined Microservices.

time, and skillset. Secondly, they evaluated the ease of isolating each microservice from the monolithic architecture. Lastly, they considered the market demands and requirements to ensure that the implementation of microservices aligns with the company's goals and meets customer needs. By following this process, the company was able to prioritize the implementation of microservices effectively and efficiently. The outcome of this prioritization process resulted in selecting the pricing and control functionalities as the first ones to be implemented as microservices. These two functionalities will be explained in detail in the following section. In addition, their implementation, equivalence relationship, and performance issues will be addressed in the remaining part of the Chapter.

7.2 The QuaniSDO Software

BIM Italia's most notable software, known as QuaniSDO, assists clients in monitoring the flow of hospital discharges before they are reported to the regional health authority. It is crucial to validate the data for refund estimation, and it is essential that the data submitted to the Region complies with the established formalism; otherwise, the refunds cannot be processed.

The hospital is a human service enterprise whose products are the specific sets of services provided to individual patients [136]. In this context, the QuaniSDO system, takes as input one or more *Hospital Discharge Records* (*HDR*). An *HDR* is the main instrument for collecting information on patients discharged from public and private hospitals in the country [137]. Each hospital product has a different price. To identify each product a *HDR* must be processed and categorized using a system called *Diagnosis Related Group* (*DRG*) [138]. Based on this information, each *HDR* is priced to compute the refund to be requested from the regional authority. Once an *HDR* has been priced, it is processed following rigorous checks. The goal is to verify that each *HDR* respects the defined regional rules. Figure 7.2 shows the event flow, to clarify how the system

works.

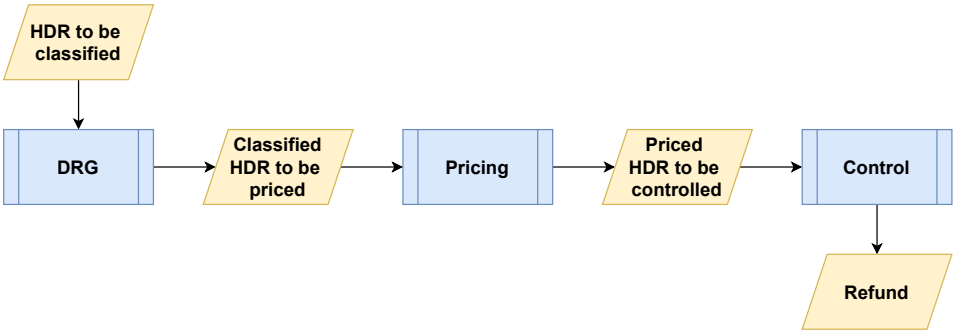

Figure 7.2: The General *QuaniSDO* Workflow.

Hardware and software requirements of the monolith are shown in Table 7.2.

	Client	**Server**
Hardware Requirements		
Platform	Windows 7/10	Windows Server 2008 R2
RAM	4 GB	4 GB
NIC	1000 Mbps	1000 Mbps
Processor	2 CPUs real/virtual	2 CPUs real/virtual
Hard Disk	50 GB	100 GB
Software Requirements		
Frameworks	.NET 3.5/.NET 4.5.1	SQL Server 2008
Programming Languages	Visual Basic 6	Visual Basic 6

Table 7.2: Monolith's Requirements

This Chapter will focus on the *Control* and *Pricing* features. They correspond to the two microservices first deployed by the company. In the following we provide a brief description of the two features reporting how the company performed the migration. The DRG Calculation functionality will be described in the next Chapter 9.

- **The *Pricing* feature** is responsible for attributing an economic reimbursement value to an *HDR*. A tariff is assigned to each element of the *HDR* which takes into account all the variables expressed at the regulatory level such as acute and post-acute, ordinary regime or day hospital, threshold days and over-threshold rates, etc.. The software maintains a complete history of regional tariffs that are applied to the *HDR* based on the discharge date. The data processed by this feature is fundamental for the calculation of the reimbursement obtained for that specific admission. This highlights the importance of this feature for monitoring the activity of a Hospital Company, or for regions that use it as a reference product

and certification of the data.

- **The *Control* feature** verifies the data reported on the *HDR* based on the defined regional rules. Each rule represents a particular check. It is important to highlight that if there is an error in the *HDR*, it can not be processed and accepted by the local authority and thus not refund. This highlights the importance of this feature for the customer, who can detect the presence of any errors in a preliminary stage of the flow. QuaniSDO is constantly updated to the regional specifications for formal validation of the *HDR* flow. The software maintains a complete history of the formalisms that are automatically applied to the cards based on the discharge date. The formalisms check both the obligatoriness and the validity of the data reported on the SDO. When an error is detected, the incorrect SDOs are moved to a specific area, each with an indication of its type of error. Any type of possible modification is automatically verified by the tool, which certifies its correctness or not.

7.3 The Migration Approach and Performance Issues

The system migration started with the development of the *Pricing* microservice. Before its release, the microservice was tested to ensure that performance constraints were met. The successful testing allowed to release the *Pricing* microservice. Although the company had not yet received customer feedback for the *Pricing* microservice, the development of the *Control* microservice started. This microservice was developed based on the knowledge acquired from the experiece of migrating the *Pricing* functionality. This was possible due to their equivalence:

1. They both need a type conversion to proceed with the computation. In fact, in both functionalities, the data to be processed is allocated to a regionalized SQL Database, which is invoked to retrieve all the data needed to perform the calculation. After loading the data from the database, the *Data Transformation Services* (*DTS*) create a package to make the data available to the calling function. Thus, the *HDR* write the retrieved data into a .txt file. This operation implies, in both functionalities, the need to perform data conversion.

2. Each functionality foresees the application of a set of regional rules. For the *Pricing* the rules defines the set of prices to be applied. Concerning the *Control*, the rules defines the set of check to be performed to the *HDR*. The regional rules are constantly subject to annual regulatory adjustments and change based on the region. The only difference in the application of the rules lies in the fact that, to price the *HDR*, the application sequence is well-defined, while for its *Control*, the application can occur randomly.

3. The two identified functionalities are the most relevant for the system. In fact, their application is essential for the *QuansiSDO* purposes.

Even for the *Control* microservice, the necessary performance tests were succesfully carried out. The implemented microservice has been deployed to *AWS*. Technical details about the *AWS*, and *PostgreSQL* configurations are reported in Table 7.3.

	AWS	**PostgreSQL**
Type	T2.large	db.t3.small
Operating System	Linux Centos 7.7.1908	-
CPU	2, virtual	2, virtual
RAM	8 GB	2 GB
Hard Disk	10 GB	-

Table 7.3: *AWS* and *PostgreSQL* configurations

As already mentioned, the testing results reveald that the microservice met the required constraints. Unfortunately, the performance testing was executed using old and incomplete data. Thus, once the customers operate on the system using the real data, some criticism arised for the *Control* microservice. In fact, the response time worsened considerably. In particular, considering a number of requests equal to 10,000, the response time went from 177.0 seconds for the monolithic system to 560.0 seconds for microservices-based one. Therefore, using the same input dataset, the company carried out new tests on the *Pricing* microservice. In this case, a minor deterioration in performance was noticed. In particular, for 10,000 requests, the response time went from 12.2 seconds to 14.2 seconds. The migration of the *Control* microservice was conducted, as mentioned, using the lesson learned from the *Pricing* migration. While this strategy helped to quickly develop the *Control* microservice, it did not consider its peculuarities. As a consequence, the decrease in performance observed on the *Control* microservice was more prominent than the *Pricing* microservice. Figure 7.3 shows the *Control* performance degradation. The orange pipe, represents the monolothic response time over the number of requests. It is considered as the response time baseline.

The *Control* microservice response time, represented as yellow pipe, has an exponential trend. Figure 7.4 shows the same trend for the *Pricing* microservice.

Based on this performance issues, the company decided to proceed with a refactoring of the two microservices.

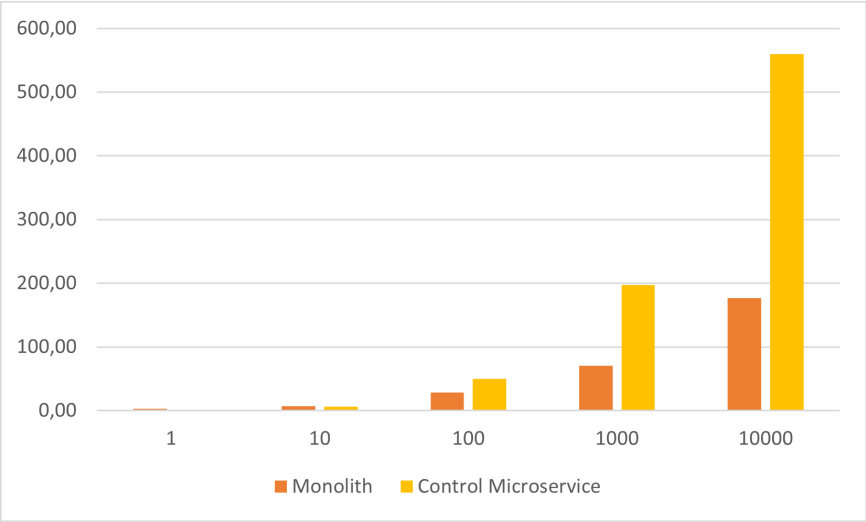

Figure 7.3: The Control Microservice Performance Degradation.

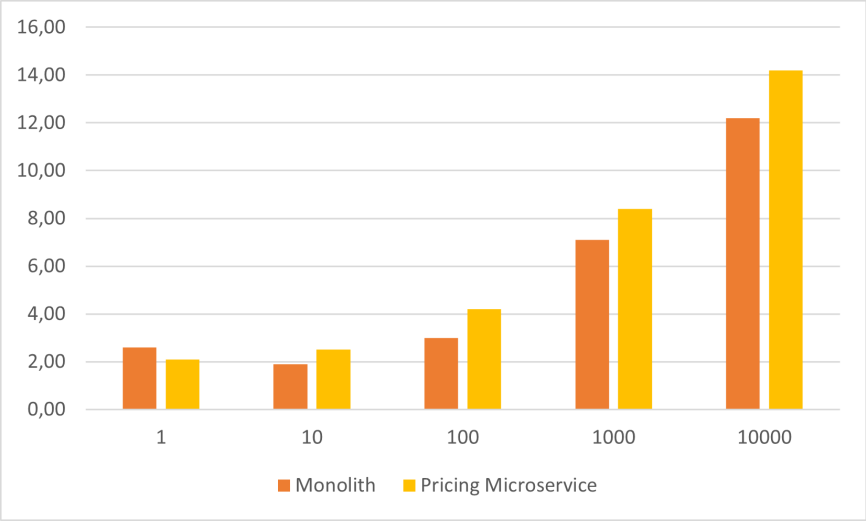

Figure 7.4: The Pricing Microservice Performance Degradation.

7.4 Conclusion

The company then identified the functionalities and dependencies of the monolithic system to determine the microservices required. The prioritization of microservices led to the selection of controls and pricing as the first two functionalities to implement. The equivalence relationship between these functionalities led to the same implementation choices for both microservices.

Despite thorough testing before the product release, the customer reported a significant degradation in system performance, particularly in terms of response time. This highlights the importance of continuous monitoring and optimization of microservices to ensure their performance meets customer expectations.

Overall, BIM Italia's experience with QuaniSDO underscores the importance of careful planning, prioritization, and testing when migrating to MSA. It also emphasizes the ongoing need for monitoring and optimization to ensure optimal system performance.

In the next Chapter, the process of quality-driven refactoring presented in Chapter 6 will be applied to the two microservices, controls, and pricing. By applying this process to the two microservices, BIM Italia aims to improve the performance of the system and address any issues that may have arisen after its release.

Chapter 8

Quality-Driven Refactoring in BIM Italia

This Chapter presents the proposed the application of the quality-driven refactoring approach presented in Chapter 6 for the refactoring of the *Pricing* and *Control* microservices migrated from BIM Italia.

8.1 *Control* microservice refactoring

This section presents the application of the quality-driven refactoring method presented in Chapter 6 for the reengineering of the *Control* microservice. Thus, the process follows the four steps: i) antipattern analysis on monolith and microservices, ii) resolutive patterns selection, iii) code refactoring and assessment, and iv) microservices deployment or refactoring. Since the performance problems arises form the *Control* microservice, this was the first refactored by the company. The expertise acquired during the refactoring of this microservice, has been employed as a basis for the refactoring of the *Pricing* microservice. The entire process reflected the steps represented in Figure 8.1.

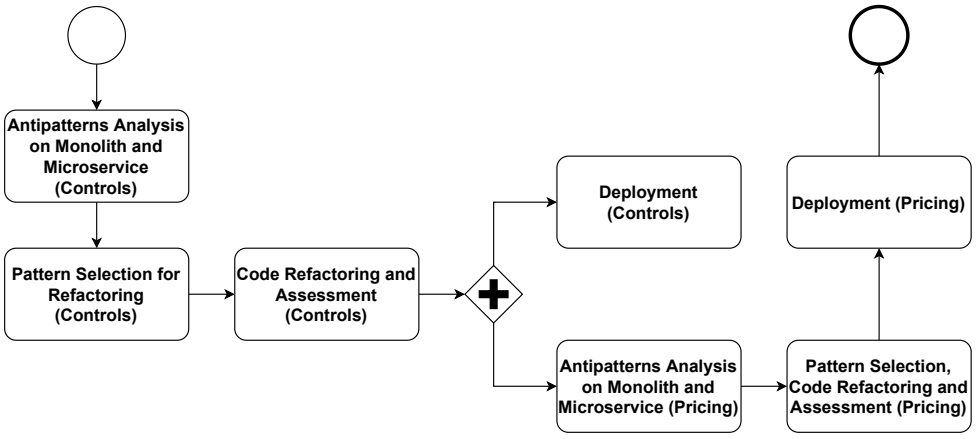

Figure 8.1: Refactoring Timeline.

8.1.1 Antipatterns Analysis for the *Control* Microservice

As shown in Figure 8.1, the refactoring process starts with the antipattern detection on both monolith and *Control* microservice. The objective is to investigate whether the performance antipattern present in the monolith affected the migrated microservice.

Monolith Antipatterns. The company adopted static and dynamic analysis to detect antipatterns on the monolith. The first, has been performed throught code analysis. The latter one, considering the data flow necessary to carry out the two functionalities. Due to the old nature of the technology employed by the monolith, both the analysis has been executed manually, based on the team's knowledge. Table 8.1 describe the most relevant performance antipattern over the ten found referring to the *Control* functionality in the monolith: the *Tower of Babel* (*ToB*). It highlights the descriptions, the causes and their identification in the monolith. Based on the developers experience and measurements, we discovered that the predominant antipattern affecting the performance of the monolith has been the *ToB*. The other antipatterns detected on the monolith are shown in Appendix B.

Tower of Babel	
Description	Occurs when processes excessively convert, parse, and translate internal data into a common exchange format [139].
Causes	The same information is often translated into an exchange format (by a sending process) and then parsed and translated into an internal format (by the receiving process).
Identification	The data has been converted at least two times: from data to *DTS* and viceversa.

Table 8.1: Tower of Babel - Control Functionality - Monolith

Micorservices Antipatterns The antipatterns detection on the microservice was conducted using *Embold*[1], an *IntelliJ IDE* plugin. *Embold* is a software analytics tool based on Artificial Intelligence. This plugin is conceived to improve *software quality* by analyzing source code. Using this platform, the development team recognized four antipatterns on the *Control* microservice. Table 8.2 describes the most relevant performance antipatterns detected: the *Data Taffy* (*DT*). Indeed, the *DT* has the highest impact on performance. In particular, the access to the database revealed to represent a bottleneck due to the DBMS has to respond to the requests coming from both the monolith and the microservice. The remaining antipatterns detected on the microservices are shown in Appendix B.

[1] https://plugins.jetbrains.com/plugin/14711-embold

Data Taffy	
Description	All services have full access to all objects in the database. This is also referred as Entangled Data [140, 141].
Causes	Lots of stored procedures, embedded complex queries, and object relationship managers all accessing the database [142].
Identification	The access to data is always complete regardless the invocation.

Table 8.2: Data Taffy - Control Functionality - Microservice

8.1.2 Patterns Selection for the *Control* Microservice

Once the antipatterns has been detected, following our quality-driven refactoring approach, the company evaluated the three cases *intersection between antipatterns*, *disjoint sets of antipatterns - cause-effect relation*, and *disjoint sets of antipatterns - none relations* mentioned in Chapter 4. The analysis displayed that none of the antipatterns detected on the monolith is still present in the microservice (no *intersection between antipatterns*). In addition, the analysis revealed a cause-and-effect relationship between the two antipatterns *ToB* and *DT* (case *disjoint sets of antipatterns - cause-effect relation*). This relationship emerge from the excessive use of *DTS* and further parsing and transformation of data in the monolith. Investigating the *BIM Italia* migration approach, we realized that the company gave direct and complete access to all database objects to avoid the excess in parsing. This strategy caused the presence of the *DT* antipattern on the microservice.

As a consequence, the pattern selection was performed to solve this two antipatterns. The patterns selected are presented in Tables 8.3, 8.4, and 8.5. For each pattern are reported the description, application strategy and the motivations behind their choice. Note that *iterator* and *Template-Method* (*DAL*) pattern resolve the *ToB* antipattern present in the monolith whereas the *Cache-Aside* (*CA*) is adopted as a solution for the *DT* antipattern.

Iterator	
Description	Allows to process every element of a container while isolating the user from the internal structure of the container [143].
Application Strategy	Provides a way to access the elements of an aggregate object without exposing its underlying representation. This permits to step through the elements of an aggregate without knowing how things are represented [143].
Motivation	The *Control* functionality uses complex data structures. This pattern helps to simplify access to data.

Table 8.3: Iterator - Control Functionality - Microservice

Template-Method (DAL)	
Description	Also called *DAL*, it defines the skeleton of an algorithm deferring some steps to sub-classes. Sub-classes redefine certain steps of an algorithm without changing the algorithm's structure [144].
Application Strategy	Minimize the number of primitive operations that a subclass must override to flesh out the algorithm. Identify the operations that should be overridden by adding a prefix to their names [144].
Motivation	The *Control* functionality uses a complex algorithm which proceeds in steps using similar structure. Thus, the application of the patterns allows optimize the code.

Table 8.4: Template-Method (DAL) - Control Functionality - Microservice

Cache-Aside	
Description	Applications use a cache to optimize repeated access to information held in a data store [145].
Application Strategy	Determine whether the item is currently held in the cache. If the item is not currently in the cache, read the item from the database. Store the copy of the item in the cache [145].
Motivation	For each request, the data are retrieved from the database even if similar requests have been already performed. The result is a performance setback. This pattern solves the problem.

Table 8.5: Cache-Aside - Control Functionality - Microservice

8.1.3 Control Microservice Refactoring

In the following we describe how the selected patterns has been used for the refactoring of the Control microservice. Note that each pattern has been applied once.

DAL Since the *validation check algorithm* used by the Control functionality consists in different checks, the *DAL* pattern has been applied for refactoring the Control microservice. Each check uses the same structures to access or validate the data for a given predefined date and time. Thus, the algorithm has been refactored: an abstract class has been introduced to take care of the common parts while a set of subclasses is responsible for each specific check to be implemented.

Iterator The data stored in the database are complex. To make the access and the retreival process more efficient, the *Iterator* pattern has been applied. Thus, an OdrIterator class has been introduced. This represent the iterator for the *HDR*s class. The iterator class is accountable for retrieving only the data needed. Therefore, it is no more necessary to get back all the classes from the database.

CA The analysis of the system behaviour concerning the Control functionality revealed that the catalogues stored in the databases are retrieved different time to satisfy similar requests. Thus, the *CA* pattern has been applied by converting the catalogues in a JSON file. Once the data queried from the database is saved in the cache, they remain available to satisfy new similar request.

Figure 8.2 shows the performance improvement that will be discussed in the next Section.

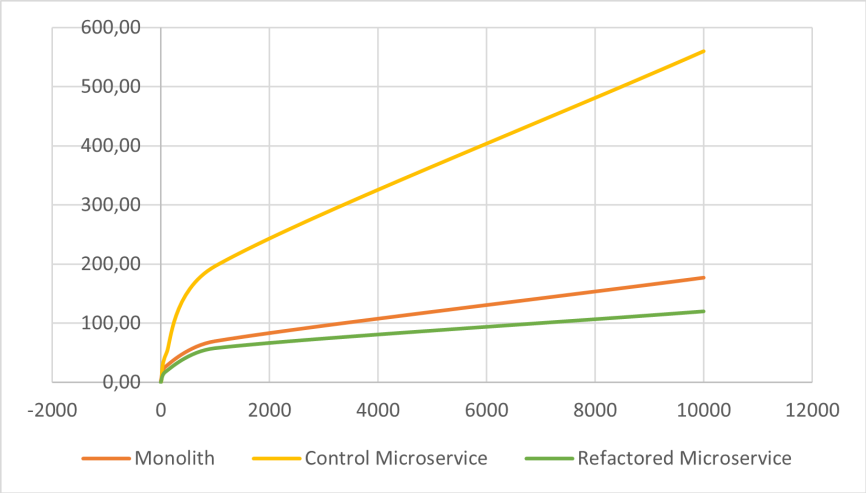

Figure 8.2: The Control Microservice Performance Improvement.

8.1.4 Control Microservice: refactoring results

As stated before, the performance testing on the first version on the microservices has been performed using old and incomplete data. Thus, the refactored *Control* microservice has been tested using the JSON files directly provided by the customers. This allowed to have a more reliable results making the testing outcome comparable with both the monolith and the first version on the *Control* microservice. Since each *HDR* different information can be requested to the database, the test has been replicated 30 times. Table 8.6 compares the average response time of the monolith and the two versions of the *Control* microservice. The new version of the *Control* microservice has a mean response time improvement of the 47% with respect to the monolith. The enhancement with respect to the first version of the microservise is of the 69%. Concerning 10,000 numbers of requests, whose worsening caused the need of refactor the *Control* microservice, the perfomance improvement is considerable. In fact, the refactored version of the *Control* microservice performs five times faster than the first version of the microservice.

Number of Requests	Monolith	Control Microservice	Refactored Microservice
1	3.0 seconds	1.1 seconds	0.15 seconds
10	7.2 seconds	5.9 seconds	3.0 seconds
100	28.0 seconds	50.0 seconds	19.0 seconds
1000	70.0 seconds	197.0 seconds	58.0 seconds
10000	177.0 seconds	560.0 seconds	120.0 seconds

Table 8.6: Performance Analysis after the Refactoring of the *Control* Functionality

8.2 *Pricing* microservice refactoring

In this section we briefly describe the application of the presente *quality-driven refactoring approach* on the *Pricing* microservice.

Antipatterns Analysis As already mentioned, the refactoring of the *Pricing* microservice started simultaneously with the Control assessment and deployment. Based on the performance results, we decided to tackle the refactoring of the *Pricing* microservice using the same approach. Due to the limited time required for the refactoring of the *Pricing* microservice, the company decided to restrict the antipatterns detection to *ToB*. This was possible because of the already presented equivalence relationship between the two functionalities. Similarly, the set of antipatterns considered to be detected on *Pricing* microservice was then restricted to the DT. The analysis revealed that this two antipatterns also affected the monolith and the *Pricing* microservice.

Patterns Selection and *Pricing* Microservice Refactoring The *Pricing* microservices was refactored based on the lesson learned of the Control microservice refactoring. Thus, the selected patterns and the refactoring strategy were directly inherited. Therefore, the adopted patterns are *DAL*, *Iterator*, and *CA*. As for the Control microservice, for the refactoring of the *Pricing* microservice, each pattern has been applied once. The *Pricing* microservice uses a *pricing algorithm* that associate a value to each part of the *HDR*. Thus, the *DAL* was applied using the same strategy adopted for the Control microservice. Even for the *Pricing* microservice, for an efficient access to the complex data, the *Iterator* pattern was applied. Thus, an iterator class has been introduced. Using this strategy the microservice retreives only the needed data. The pricing functionality does not uses catalogues. Thus, the *CA* pattern has been applied by using the memory cache to store partial data retreived for each request. Figure 8.3 shows the performance improvement.

Pricing Microservice: refactoring results We report below the results of the testing carried out on the second version of the *Pricing* microservice. Table 8.6 summarizes the obtained results. The refactored *Pricing* microservice has a mean response time improvement of the 39.8% compared with the monolith. The improvement with respect to the original microservice is 51% on average. Concerning 10,000 numbers of requests, the refactored microservice performs till 2 times faster than the monolith and the first version of the microservice.

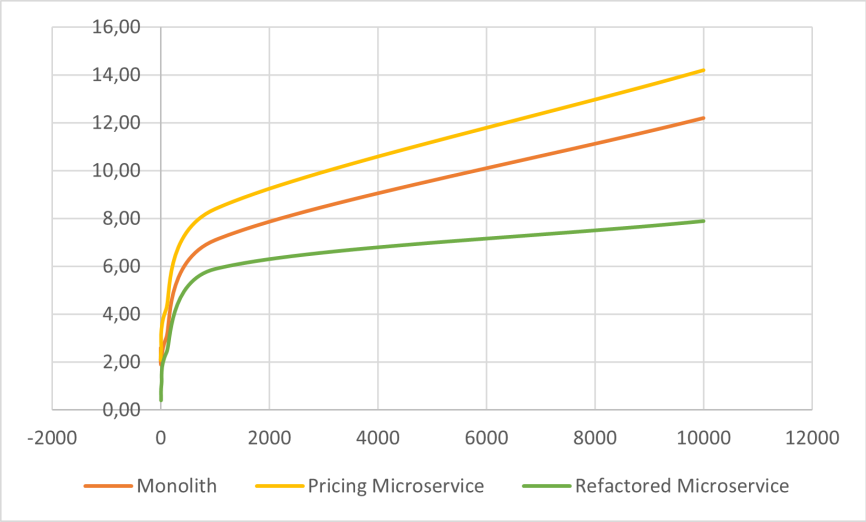

Figure 8.3: The Pricing Microservice Performance Improvement.

Number of Requests	Monolith	Pricing Microservice	Refactored Microservice
1	2.6 seconds	2.1 seconds	0.401 seconds
10	1.9 seconds	2.5 seconds	1.1 seconds
100	3.0 seconds	4.2 seconds	2.4 seconds
1000	7.1 seconds	8.4 seconds	5.9 seconds
10000	12.2 seconds	14.2 seconds	7.9 seconds

Table 8.7: Performance Analysis after the Refactoring of the *Pricing* Functionality

8.3 Conclusion

This Chapter presented a *quality-driven refactoring approach* and its application for the refactoring of microservices migrated from *BIM Italia*'s monolithic system. The goal was to perform a refactoring process to guarantee performance quality constraints satisfaction. In fact, the two migrated microservices, *Control* and *Pricing*, revealed performance issues once deployed. The applied *quality-driven refactoring approach* is based on 4 phases: i) antipattern analysis on monolith and microservice, ii) resolutive patterns selection, iii) code refactoring and assessment, and iv) microservice deployment or refactoring. The antipattern detection revealed the presence of the *ToB* on the monolith and the *DT* on the microservices. In Section 8.1.2 we reported how the strategy adopted to remove the presence of the problems concerned to the *ToB*, caused the presence of the *DT* in the microservices. Thus, in the studied case study presented, a correlation between these two antipatterns was displayed. This helped us in the selection of the resolutive patterns: *DAL*, *Iterator*, and *CA*. The application of our *quality-driven refactoring approach* results in till 86% of performance improvement on

the *Control* microservice. Then, the analysis of the *Pricing* refactored microservice showed a performance enhancement up to 81%.

The presented case study pointed out the need for a migration planning that carefully considers the quality aspects that the system should respect. In fact, a more accurate migration planning, can prevent the need of microservices refactoring. In this scenario, understanding the legacy system to be migrated is crucial. There are several reasons behind the need for legacy system understanding. The first is to evaluate if migration is really what the system needs. In fact, in some cases, it might even be enough to proceed with a simple legacy system refactoring rather than change the entire system architecture. Actually, the migration is a very expensive operation in terms of time, cost, and effort. Thus, the choice of migrating to microservices must be carefully agreed [20]. An accurate analysis of the legacy system, supports the creation of a new microservices-based system that is faithful to the original and that fully satisfies the customers requests. Moreover, by comprehension of the system, it is possible to measure the reusability of the legacy system, and whether external components may be needed. Finally, it is also useful to perform a thorough analysis of dependencies, especially if the company intends to perform an incremental migration. The legacy system failures are an important aspect to consider during its understanding. In fact, the migration should be planned to ensure that these problems do not recur within the migrated system. Consequently, a system satisfying quality constraints can be obtained.

Chapter 9

Implementation of the *Diagnosis Related Group* functionality

In this Chapter, a revised version of the quality-driven refactoring method introduced in Chapter 6 is applied for the migration of the functionality revealed to be equivalent to *Controls* and *Pricing*: the *DRG*. We will carefully analyze each step of the method and explore the decisions made and their effect on the microservice.

9.1 The *Diagnosis Related Group* Functionality

The *DRG* system, which groups patients with similar clinical characteristics and diagnoses for payment purposes, was developed in the United States in the 1980s by researchers from Yale University and the University of California [138]. To address concerns about the rising healthcare costs and the need for more transparent and standardized payment for hospital services, the Medicare program first implemented the *DRG* system in 1983. The *DRG* system determines the reimbursement amount that hospitals will receive for inpatient stays, incentives them to provide effective and efficient care. The *DRG* system criteria for patient grouping and payment amounts may vary by country. In Italy, the system is known as *DRG* Italia, and the Italian National Health Service (*SSN*) uses it to reimburse hospitals for inpatient services. *DRG* Italia groups patients based on their clinical characteristics, diagnoses, and procedures, and each group is assigned a *DRG* code with a predetermined payment amount. *DRG* Italia employs the International Classification of Diseases, Ninth Revision, Clinical Modification (*ICD-9-CM*) coding system to classify patients based on their medical conditions and diagnoses. Patient factors like age, comorbidities, and length of stay are also taken into account. Each *DRG* is assigned a relative weight based on the average cost of treating patients with similar clinical characteristics. Hospitals are reimbursed based on the *DRG* code assigned to each patient, with adjustments made for regional and hospital-specific factors. In addition to *ICD-9-CM*, the *DRG* system utilizes Major

Diagnostic Categories (MDC) to group patients with similar diagnoses and medical conditions. These 25 distinct groups are based on the *ICD-9-CM* codes and provide further categorization of patients into *DRG*s, which is essential for determining reimbursement amounts for hospitals. Overall, *DRG* Italia has helped standardize and streamline the payment process for hospital services in Italy, incentives hospitals to provide high-quality care while promoting efficiency and controlling costs. The system serves as a benchmark for evaluating hospital performance and resource allocation in the Italian healthcare system. Figure 9.1 highlights the phases of the *DRG* calculation as reported in the World Health Organization (WHO) guidelines [1].

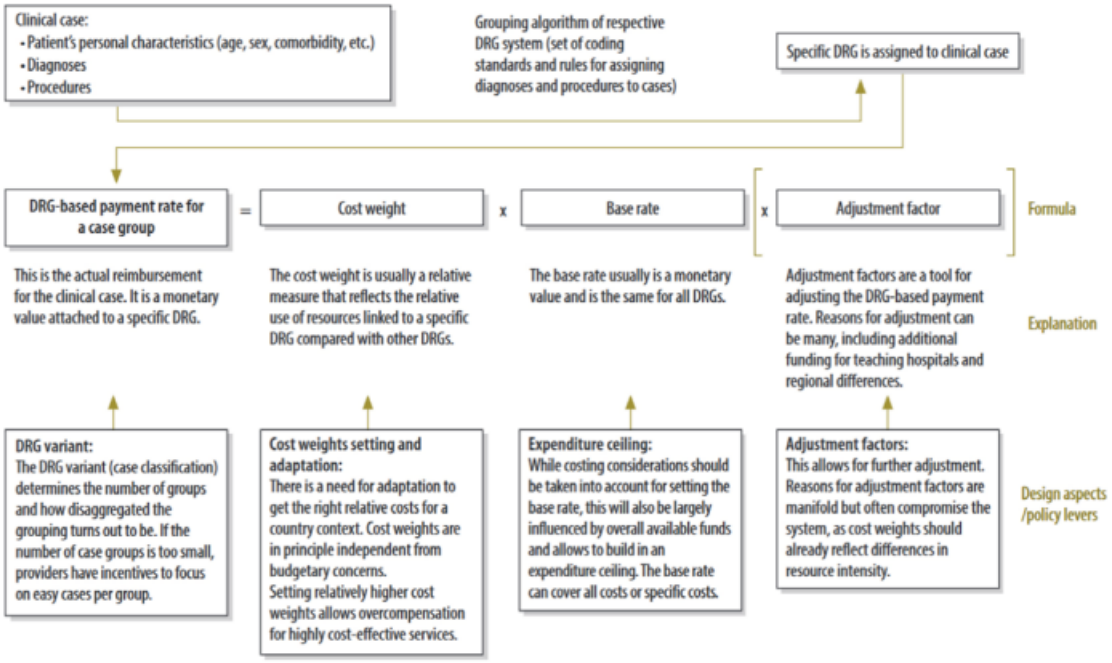

Figure 9.1: *DRG* Calculation Schema.

9.1.1 *Diagnosis Related Group* Components

The functionality of *DRG* has been broken down into two sub-components. We will going into detail of the implemention of only the more performance-intensive subcomponent. To understand how these sub-components were identified, it's essential to understand how this functionality is implemented in the monolith.

Figure 9.2 shows the general functioning of the *DRG*. When calculating the *DRG*, a query is made to the database, and the data from the *HDR* is extracted. The data considered in this phase includes sex, age, principal diagnosis, secondary diagnoses, procedures and interventions, and the discharge status. This data undergoes a coding process and is transformed into a text file that will then be passed to the grouper for

[1] https://tinyurl.com/drgwho

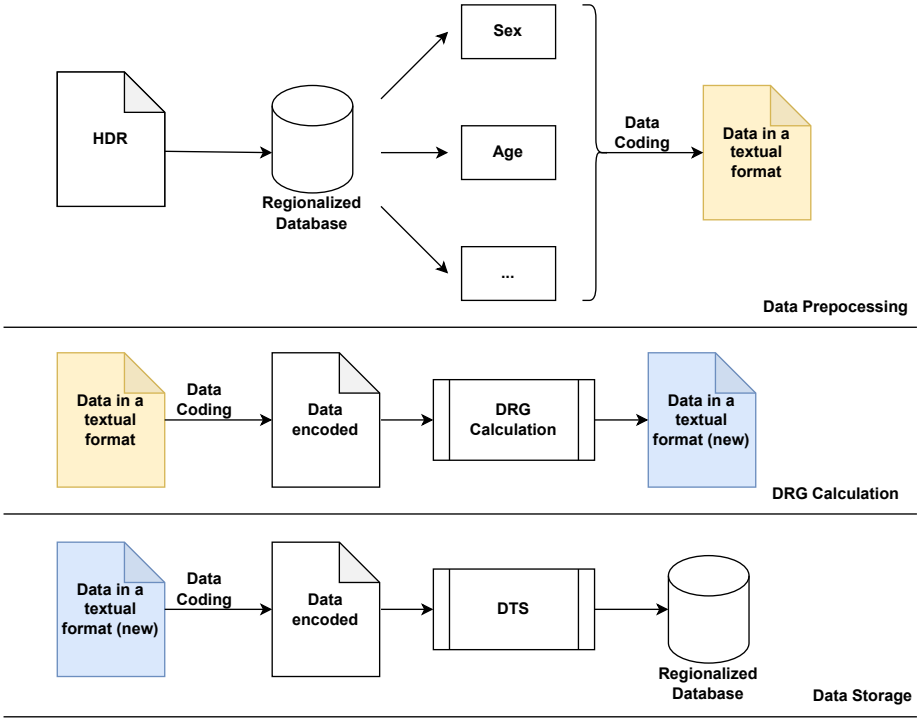

Figure 9.2: *DRG* Functionality.

the necessary calculations. To manipulate the data, the grouper converts it to assign the correct data type item. Once the calculation is done, the grouper generates a new text file that will be used by the *DTS* to insert the calculated values into the source database.

The two sub-components of *DRG* functionality were identified based on their performance requirements. The more performance-intensive sub-component involves the calculation process that occurs within the grouper, which is resource-intensive. Therefore, we will implement this sub-component only to reduce the overall computational load on the system. The two sub-components identified are *code conversion* and *DRG attribution*. For our analysis, we will consider the *code conversion* sub-component as a black box, while we will implement the *DRG attribution* part. The *DRG attribution* sub-component involves verifying the input data based on certain classifications and sets of procedures and diagnoses to assign the correct type of *DRG*. This process is computationally expensive, but it is a crucial step in the calculation of the *DRG*. The company decided to use the Strangler Pattern to gradually migrate the *DRG* functionality to a new microservice. As a result, it will also be possible to measure the performance (response time) and compare it to that of the monolith.

9.1.2 Equivalence Relationship

To ensure the successful implementation of the *DRG* microservice version using our approach, it is important to conduct a comprehensive analysis of the three functionalities

- *Pricing*, *Control*, and *DRG* - and determine their equivalence. In our analysis, we have identified two main points of equivalence that need to be addressed for a successful implementation. Firstly, there is the need to perform parsing and encoding of the data at every step of the process, with a massive use of *DTS* to ensure the correct formatting and manipulation of data. This is crucial to ensure that the data remains accurate and consistent throughout the entire *DRG* calculation process, as even small errors or inconsistencies can result in incorrect *DRG* calculations. Secondly, there is the application of a series of checks and rules based on the actual *DRG* algorithm. These checks and rules ensure that the *DRG* calculations are accurate and in line with the standard *DRG* methodology. These rules and checks are applied at various points throughout the *DRG* calculation process to ensure that the calculations are consistent and accurate.

By identifying this equivalence, we can apply a modified version of the approach used for refactoring *Pricing* and *Control* to the *DRG* microservice. This means that we can leverage the knowledge and experience gained from those previous efforts to ensure the successful implementation of the *DRG* microservice. Furthermore, this equivalence also allows us to compare the migration efforts of the three functionalities - *Pricing*, *Control*, and *DRG* - in terms of time, costs, and effort. This will help us to identify any potential challenges or bottlenecks in the migration process and optimize our approach accordingly.

9.2 Application of the Approach

Regarding the *DRG* functionality, the refactoring approach proposed was not applied in its original form. Specifically, the analysis of antipatterns on the related microservice was not performed as it had not yet been implemented. However, we proceeded with the migration by performing an the antipatterns analysis on the monolith and selecting appropriate patterns for implementing the *DRG* sub-component. In the upcoming sections, we will outline the steps we took in more details.

9.2.1 Antipatterns Analysis

Given the equivalence relationship between functionalities shown in Section 9.1.2, and in order to speed up development times, the company has decided to perform an antipatterns analysis by considering only the subset already identified for the *Pricing* and *Control* functionalities on the monolith. Therefore, the company searched for the *ToB* antipattern within the monolith, where operations related to the *DRG* calculation functionality are performed. As already explained, this antipattern refers to the proliferation of parsing and serialization code throughout an application, resulting in the application becoming increasingly complex and difficult to maintain. The identification

of this antipattern within the monolith is likely due to the repeated parsing operations that are required for the *DRG* calculation functionality.

As it happened with the refactoring of the two microservices *Control* and *Pricing*, since the monolith was implemented in Visual Basic, the only way to perform analysis of anti-patterns was to rely on the software system knowledge possessed by the developers. This highlights the importance of having knowledgeable developers who understand the intricacies of the software system they are working on, especially when dealing with legacy code. While modern tools and techniques for identifying anti-patterns can be helpful, they are not always sufficient in the face of complex and poorly documented systems. Therefore, investing in the expertise of developers and encouraging knowledge sharing within the team can be crucial for maintaining and improving software quality over time.

9.2.2 Patterns Selection

In contrast to the approach taken for the refactoring of the *Pricing* and *Control* microservices, the selection of patterns for migrating the *DRG* functionality has two objectives. Firstly, the migration aims to solve the *ToB* antipattern detected on the monolith, and secondly, it aims to avoid the occurrence of *DT* in the implemented microservice as occurred in the *Control* and *Pricing* microservices. To achieve these goals, the company decided to use the same patterns used for the refactoring of *Pricing* and *Control*, namely *Iterator*, *DAL*, and *CA*. Based on our experience with the refactoring of *Pricing* and *Control*, we have identified these patterns as effective solutions to address the challenges of the *DRG* migration. We believe that the *Iterator* pattern will help in traversing through the data structures efficiently and effectively, while the *DAL* pattern will allow to reuse the common code across multiple *DRG* functions. Moreover, the *CA* pattern will enable the implementation of an high-performance data cache that can significantly reduce the load on the database and improve the overall system performance. By adopting these patterns, we aim to reduce the development time and complexity of the *DRG* migration process.

For the implementation of the subcomponent of the *DRG* functionality, the patterns have been applied as follows:

> **DAL**: the *DRG* requires the application of a series of rules to associate the correct code number to all the *HDR* parts. Therefore, every data check is carried out using the same structure. The algorithm related to the *DRG* calculation has been reimplemented by inserting an abstract class whose task is to manage the common parts, leaving the more specific tasks to the subclasses.
>
> ***Iterator***: a class responsible for retrieving the data necessary for the *DRG* calculation has been implemented. This reduces the need for continuous requests to the database.

CA: the pattern allows avoiding the execution of continuous requests to the database that are similar to each other. Instead of being lost, the data is saved in the cache where it will be accessible for similar requests. Even in this case, the chosen format is JSON.

Applying the *DAL* and *Iterator* pattern has allowed solving the *ToB* issue, ensuring that it did not occur again in the subcomponent of the implemented microservice. The *CA* pattern has allowed overcoming the presence of the *DT* within the new microservice, an antipattern that had instead emerged in the first implementation of the two microservices related to *Control* and *Pricing* functionalities.

9.3 Results Discussion

Various parameters were taken into account during the analysis of the results obtained from implementing the modified refactoring approach. The goal was not solely to compare the response time performance of the monolith and the implemented functionality. The time, cost, and effort involved in implementing the *Pricing* and *Control* microservices were also analyzed and compared to those required for implementing *DRG*.

9.3.1 Performance Analysis

As already mentioned, due to time constraints for development, the company implemented a subcomponent of *DRG* and adopted the Strangler Pattern, which allowed for an empirical evaluation of the increase in performance. To test this, a set of ten different tests was constructed based on real customer requests. Those tests were performed on both the monolith and the microservice, which consisted of a black box monolithic subcomponent and the implemented microservice. The testing scenario is represented in Figure 9.3.

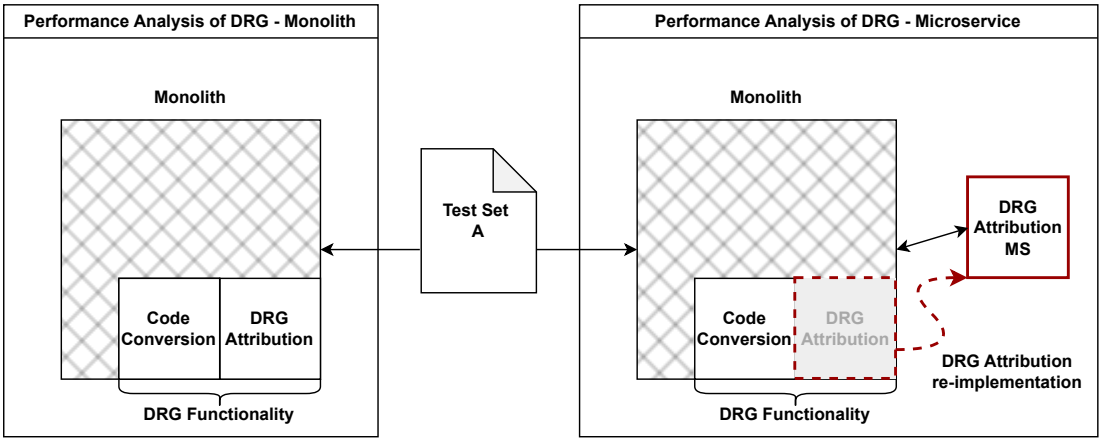

Figure 9.3: *DRG* Testing Scenario.

Tables 9.1 and 9.2 present the response time data for the average and worst-case scenarios, respectively, after the partial migration of the *DRG* functionality to a microservice architecture. From the analysis of Table 9.1, it can be observed that the microservice architecture generally outperforms the monolithic architecture in terms of average response time. For instance, with a single request, the microservice architecture had an average response time of 0.47 seconds, which is significantly better than the monolithic architecture's average response time of 3.32 seconds. Even with 10,000 requests, the microservice architecture had an average response time of 7.45 seconds, which is still better than the monolithic architecture's average response time of 18.02 seconds. On average, the microservice architecture improved the response time by approximately 85%.

Number of Requests	**Response Time - Average**	
	Monolith	**Microservice**
1	3.32 seconds	0.47 seconds
10	4.17 seconds	1.62 seconds
100	5.29 seconds	2.92 seconds
1000	8.13 seconds	4.38 seconds
10000	18.02 seconds	7.45 seconds

Table 9.1: Average response time after the partial migration of the *DRG* Functionality

Similarly, in Table 9.2, the microservice architecture generally outperforms the monolithic architecture in terms of worst-case response time as well. For example, with only one request, the microservice architecture had a worst-case response time of 2.20 seconds, which is better than the monolithic architecture's worst-case response time of 3.48 seconds. Even with 10,000 requests, the microservice architecture had a worst-case response time of 12.52 seconds, which is significantly better than the monolithic architecture's worst-case response time of 18.17 seconds. On average, the microservice architecture improved worst-case response time by approximately 53%.

Number of Requests	**Response Time - Worst Case**	
	Monolith	**Microservice**
1	3.48 seconds	2.20 seconds
10	4.28 seconds	3.12 seconds
100	5.43 seconds	4.09 seconds
1000	8.26 seconds	6.12 seconds
10000	18.17 seconds	12.52 seconds

Table 9.2: Worst case response time after the partial migration of the *DRG* Functionality

In conclusion, the results demonstrate that the microservice architecture generally outperforms the monolithic architecture in terms of response time, both in terms of average response time and worst-case response time, with improvements ranging from approximately 53% to 85%.

9.3.2 Time, Effort and Costs Analysis

When evaluating the effectiveness of our quality-driven refactoring process, it is not sufficient to only consider response time data. It is also important to examine whether the process helps optimize the time, cost, and effort required for developing a microservice from a monolithic system. Response time data provides valuable insights into the performance of the system after refactoring. However, it does not provide a complete picture of the benefits of the refactoring process.

Time Analysis Table 9.3 summarizes the details of the migration time analysis for the three implemented microservices: *Pricing*, *Control* and *DRG*. The first table column describes the parameters considered for calculating the migration times. The next three columns refer to the three implemented microservices. For each microservice, the following information is provided: the time required for migration, the time required for refactoring the microservice, the total time required. Since only the 20% of *DRG* has been implemented, the table reports also a row estimating the total time required to develop the entire microservice.

Parameter	***Pricing***	***Control***	***DRG***
Time required for migration	7.50 weeks	3,75 weeks	0,75 weeks
Time required for refactoring	6,25 weeks	4,00 weeks	0,00 weeks
Total time required	13,75 weeks	7,75 weeks	3,75 weeks

Table 9.3: Time Analysis

The data shows that the migration for *Pricing* and *Control* took 7.50 and 3.75 weeks, respectively, while the migration for *DRG* took only 0.75 weeks. The refactoring for *Pricing* and *Control* took 6.25 and 4.00 weeks, respectively, while the *DRG* microservice was only partially migrated (20%), resulting in a refactoring time of 0.00 weeks.

To calculate the total time required, the migration and refactoring times are added. The total time required for *Pricing* and *Control* was 13.75 and 7.75 weeks, respectively, while the total time required for *DRG* was only 3.75 weeks. However, it is important to note that the total time for *DRG* was multiplied by a factor of 5 because it was only partially migrated.

Overall, the data in the table suggests that the quality-driven migration process applied for the *DRG* microservice development made it more efficient compared to *Pricing*

and *Control*. However, it is important to keep in mind that the *DRG* microservice was only partially migrated and may require more time in the future.

Effort Analysis Table 9.4 provides an overview of the effort analysis that was necessary to migrate the three microservices. As with the time analysis, the table is composed of four columns, with the first column describing the parameters that were considered during the effort calculation. Similarly, the other three columns of the table are dedicated to each of the implemented microservices. The table presents the total number of people involved in the migration of each microservice, including the number of people employed for refactoring the *Pricing* and *Control* microservices. Additionally, the table presents the total time spent on the complete migration of the three microservices. The total effort is expressed in terms of men/months. This value was computed using the data mentioned previously. As previously stated, the estimation of the total effort for implementing the *DRG* microservice is reported in a separate row, owing to the reasons explained in the preceding paragraph.

Parameter	***Pricing***	***Control***	***DRG***
People required for migration	3	2	2
People required for refactoring	6	1	0
Total number of people required	9	3	2
Total time required	13,75 weeks	7,75 weeks	0,75 weeks
Total effort (men/months)	1,53	2,58	1,88

Table 9.4: Effort Analysis

The data shows that the total effort required for *Pricing* and *Control* is higher than the *DRG* microservice. The table indicates that three people were required for the migration of *Pricing*, two people for the migration of *Control*, and two people for the migration of *DRG*. On the other hand, six people were required for the refactoring of *Pricing*, one person for the refactoring of *Control*, and zero people for the refactoring of *DRG*. To calculate the total number of people required, the number of people required for migration and refactoring are added. The total number of people required for *Pricing* and *Control* is 9 and 3, respectively, while the total number of people required for *DRG* is 2. Lastly, based on the provided data, we evaluated that the total effort required for *Pricing*, *Control*, and *DRG* is 1.53, 2.58, and 1.88 men/months, respectively. Note that, as for the time analysis, the total effort required for *DRG* was multiplied by a factor of 5 due to it being only partially migrated.

Costs Analysis The cost analysis for the migration of three microservices is presented in Table 9.5. Similar to previous tables, it contains four columns, with the first column

providing details on the cost calculation parameters. The subsequent three columns represent each microservice: *Pricing*, *Control*, and *DRG*. The parameters considered for cost calculation are presented as rows in the table and include the number of people involved in the migration of the three microservices, the total migration time in weeks, and the extimate average wage hour per person. The table includes also an estimation of the cost required to develop the entire *DRG* microservice, as explained in the first paragraph. Notably, the cost calculation does not include any software or hardware purchases. The primary goal of the analysis is to evaluate the migration process's quality in terms of time, effort, and cost, rather than the overall migration process.

Parameter	***Pricing***	***Control***	***DRG***
Total number of people required	9	3	2
Total time required	13,75	7,75	0,75
Total time (hours)	1320	744	72
Average wage/hour	17,00 €	17,00 €	17,00 €
Total average cost	201960,00 €	37944,00 €	12249,00 €

Table 9.5: Costs Analysis

As previously mentioned, the *Pricing* and *Control* microservices had to be refactored due to poor performance in terms of response time. In contrast, only 20% of the *DRG* microservice was developed following our guidelines based on antipatterns. After taking into account the time and personnel involved in implementing and refactoring the *Pricing* and *Control* microservices, as well as developing 20% of the *DRG* microservice, we calculated the costs for the company to be 201,906 € for *Pricing*, 37,944 € for *Control*, and 12,249 € for *DRG*. It should be noted that the cost for *DRG* was multiplied by a factor of five to arrive at the total cost.

Based on the economic data reported earlier, it can be inferred that the development and refactoring of the *Pricing* and *Control* microservices were relatively more expensive compared to the development of the *DRG* microservice. Specifically, the cost for *Pricing* was 15 times higher than the cost for *DRG*, while the cost for *Control* was over 3 times higher than the cost for *DRG*. However, it should be noted that additional investment may be required to complete the *DRG* microservice and ensure its functionality. As a result, this cost analysis serves as a preliminary assessment and should not be regarded as a definitive measure of the project's success.

9.4 Conclusion

This Chapter has demonstrated the successful application of the quality-driven refactoring process, specifically the antipattern-based approach, in implementing the *DRG*

microservice. After explaining the purpose of the *DRG* and how it was split into two sub-components, it was found that the more complex sub-component, representing only 20% of the total functionality, was the most performance-intensive. Antipattern detection was applied to this sub-component, using the same subset of antipatterns previously identified in the analysis of the two microservices *Pricing* and *Control*. The application of the corresponding patterns resulted in estimated response time improvements ranging from 53% to 83%. Furthermore, the analysis of time, cost, and effort resulting from the adoption of our guidelines for implementing *DRG* resulted in more than 50% time savings, requiring slightly less effort than *Pricing* and *Control*, with a reduced economic cost of approximately 50%. Overall, this demonstrates the effectiveness of the quality-driven refactoring approach in improving performance, reducing costs, and optimizing the implementation of complex microservices like *DRG*.

Part IV

Conclusions

Chapter 10

Conclusions

The process of migrating from a monolithic application to a microservices-based architecture involves refactoring the large, monolithic codebase into smaller, independent services that can be developed, tested, and deployed separately. This migration process requires careful planning and execution to ensure a smooth transition to the new architecture while minimizing disruption to the existing application. Failing to consider software quality attributes during the migration process can lead to added time, effort, and cost, as the system may require further refactoring to meet the necessary requirements. The primary focus of this Thesis is to answer the research question: *what strategies can be implemented to ensure that the migration to microservices leads to the achievement or improvement of a predefined set of software qualities?* To achieve this objective, a quality-driven migration approach to microservices is proposed, which aims to help organizations achieve their desired outcomes while maintaining or improving quality attributes throughout the migration process. This approach places non-functional requirements as a first-class entity in all defined steps.

10.1 Thesis Findings

This Section summarizes the findings of the research objectives presented in the Introduction Chapter and addressed with this Thesis.

RO1: analysis of the state-of-the-art in quality-driven migration to microservices.
We conducted the *SLR* presented in Chapter 3 to investigate how researchers have considered quality aspects in the migration process. Our search yielded 58 papers published in the last six years, out of over 2000 results. Our research questions (RQs) were:

> RQ1: What is the trend in system migration to microservices from 2015 till now?
>
> RQ2: Are there any studies that address the problem of migration to microservices considering software qualities?

RQ3: Which of the three steps for the migration to microservices did the researchers focus on?

Our results showed that the trend of considering software quality attributes during microservices migration is increasing. 67.24% of the papers included in our SLR considered software qualities during migration. However, quality improvement was not the primary goal of migration, as only a limited number of works (5.12%) considered the same set of quality attributes in at least two phases of the migration. The most commonly considered quality aspects during the comprehension phase were coupling (36.36%), cohesion (27.27%), performance (54.54%), and scalability (27.27%). Similarly, during the microservices identification phase, the most studied qualities were coupling (57.89%), cohesion (52.63%), scalability (21.05%), availability (15.79%), and modularity (15.79%). When assessing the identified microservices, the most studied quality attributes were cohesion (35.71%), coupling (35.71%), and performance (28.57%).

Although many approaches are based on the static and dynamic structure of the system, the relationship between quality aspects and the techniques used in each phase is not consistent. In the comprehension and microservices identification phases, Data-Driven, Domain-Driven, Dynamic Analysis, and Static Analysis approaches are used primarily to focus on system functionality. Clustering algorithms are often applied to identify microservices based on the system's static and dynamic structure. However, there is no evidence of qualities related to the packaging phase since researchers consider this phase in very few cases. The results indicate a great interest in attributes such as coupling and cohesion, while other quality aspects are not given much attention. Nonetheless, performance-related aspects are gaining increasing attention from researchers.

RO2: definition of a quality-drvien migration to microservices approach aiming to sutisfy non-functional requirements.

Chapter 4, presented two distinct quality-driven migration approaches. The first approach involves three phases, namely system comprehension, microservice identification and assessment, and microservices packaging. Each phase considers quality attributes, and the output of each phase is carried over to the next. During the system comprehension phase, antipattern analysis is performed to eliminate identified antipatterns and apply related patterns and tactics for precise decomposition strategies. The most optimal decomposition strategy is selected. However, this process is complex and time-consuming, which led to the second quality-driven approach based on antipattern analysis. This approach involves three phases: migration planning, system decomposition, and migration execution. In migration planning, quality constraints are defined, and the system is analyzed. Antipattern detection is performed during system decomposition, and an augmented graph is used to visualize the system's classes, methods, dependencies, and antipatterns. The output of this phase is a unique system decompo-

sition. During migration execution, microservices are implemented, and an assessment is conducted. If the microservice(s) comply with the quality constraints, they are implemented. Otherwise, the system undergoes a refactoring process explained in Chapter 6.

RO3: graph-based representation for antipatterns detection.
We provided a matemathical description of the graphs that represents the legacy system. The graph representation is one of the task that is part of the microservices identification phase of the quality-driven migration approach shown in Chapeter 4. The graph representation: i) permits to represent the software, ii) permits the detection of possible antipatterns in the system. The matemathical formulation for the graph representation and antipatterns detection is shown in Chapter 5. The process for developing this graph-based representation and the mathematical formulation allowing to detect antipatterns have been carefully crafted and are currently under submission as a conference paper.

RO4: definition of a qualtiy-driven refactoring approach of microservices derived from legacy, monolithic-based, software.
Chapter 6 introduced the quality-driven refactoring process as part of the quality-driven migration approach presented Chapter 4. The pocess comprises three phases. The first phase involves the analysis of antipatterns in both the monolith and microservices. The Chapter discusses the different relationships that may exist between the antipatterns found in the monolith and those related to microservices. Three potential relationships were identified, and each was analyzed in detail.

The second phase involves selecting resolution patterns based on the relationships found in the previous phase. If there are common antipatterns in the monolith and microservices, the strategy involves resolving only the common antipatterns. Alternatively, if the antipatterns in the monolith cause the emergence of antipatterns in microservices, both antipatterns are resolved. Finally, if the sets of antipatterns are disjoint, only the antipatterns present in microservices are resolved.

The third phase of the quality-driven refactoring process involves the actual refactoring of microservices using the selected patterns. The microservices are then assessed, and the Chapter explains the two possible outputs of this assessment: either the microservices comply with the required system quality, or they do not. Based on the assessment's results, the system is either deployed in the fourth phase, or the process is repeated.

RO5: analysis of a real-world case study to investigate the migration process adopted and monitor non-functional requirements pre and post migration.
The challenges and opportunities associated with transitioning from a monolithic ar-

chitecture to microservices are exemplified by BIM Italia's flagship product QuaniSDO, which has been examined in 7. BIM Italia made the decision to adopt microservices due to market demands and the difficulty of maintaining the monolithic system. To prepare for the transition, BIM Italia analyzed the development team's skills and provided training on MSA and related technologies.

The company then identified the functionalities and dependencies of the monolithic system to determine the microservices required. Prioritizing microservices led to the selection of controls and pricing as the first two functionalities to implement. The equivalence relationship between these functionalities resulted in the same implementation choices for both microservices.

Despite extensive testing before release, customers reported a significant decline in system performance, particularly in terms of response time, underscoring the importance of continuous monitoring and optimization of microservices to meet customer expectations.

BIM Italia's experience highlights the significance of careful planning, prioritization, and testing when migrating to MSA, as well as the ongoing need for monitoring and optimization to ensure optimal system performance.

RO6: application of the quality-driven refactoring approach on the real-world case study to refactor two already implemented microservices suffering of performance issues

In Chapter 8, the quality-driven refactoring approach has been validated by applying it to the Pricing and Control microservices derived from the QuaniSDO monolithic system. The aim of the refactoring process was to ensure that performance quality constraints were met, as the two migrated microservices, Control and Pricing, exhibited performance issues once deployed. The quality-driven refactoring approach consisted of four phases: antipattern analysis on the monolith and microservice, selection of resolutive patterns, code refactoring and assessment, and microservice deployment or refactoring. The antipattern detection revealed the presence of the TOB on the monolith and the DT on the microservices. The resolutive patterns selected were DAL, Iterator, and CASIDE. The application of the quality-driven refactoring approach resulted in up to 86% performance improvement on the Control microservice, while the Pricing microservice showed a performance enhancement of up to 81%.

Finally, Chapter 9 has illustrated how the quality-driven refactoring process, specifically the antipattern-based approach, was successfully applied to implement the *DRG* microservice. After breaking down the *DRG* into two sub-components, it was discovered that the more complex sub-component, which accounted for only 20% of the total functionality, was the most performance-intensive. Antipattern detection was applied to this sub-component, using the same subset of antipatterns that were previously identified in the analysis of the two microservices related to *DRG*. The application of the

corresponding patterns resulted in estimated response time improvements ranging from 53% to 83%. Furthermore, an analysis of time, cost, and effort suggested that implementing *DRG* using our guidelines should result in more than 50% time savings, requiring slightly less effort than Pricing and Control, and reducing economic costs by approximately 50%. Overall, these results demonstrate the effectiveness of the quality-driven refactoring approach in improving performance, reducing costs, and optimizing the implementation of complex microservices such as *DRG*.

10.2 Future work

In this Thesis, we have presented: i) the conceptual framework for the quality-driven migration to microservices, ii) a graph-based modeling of the system for antipatterns detection, and iii) a quality-driven refactoring approach for the refactoring of microservices derived from a monolithic system. The quality-driven refactoring approach has been adopted by a national company, BIM Italia, and demonstrated promising results in improving performance, reducing costs, and optimizing the implementation of complex microservices. However, there are still some future works that could be explored to further enhance our approach.

Firstly, we plan to expand the graph augmentation technique used in our antipattern-based approach to incorporate dynamic analysis of software. By doing so, we believe that we can capture more accurate and diverse antipatterns and improve the overall effectiveness of our approach.

Secondly, preprocessing strategies should be implemented to address the antipatterns identified in the graph. By resolving these issues prior to decomposition, the quality of the resulting microservices can be improved. This can lead to better performance, scalability, and maintainability of the system.

Thirdly, machine learning-based clustering strategies should be implemented to improve the quality and efficiency of the decomposition process. This could result in more optimal and effective decompositions, which would in turn lead to better microservice architectures.

Finally, the set of quality criteria that the migration intends to meet should be expanded, with a particular focus on software sustainability. To this end, we have already initiated a discovery phase for identifying relevant sustainability patterns and antipatterns, which will be incorporated into the framework to ensure that the resulting microservices are not only functional and efficient, but also sustainable over the long term.

10.3 List of Pubblications

C1 Roberta Capuano. Enhancing System Quality Attributes via Microservices Adoption (short paper). In Robert Heinrich, Raffaela Mirandola, and Danny Weyns, editors, ECSA 2021 Companion Volume, Virtual (originally: Vaxjo, Sweden), 13-17 September, 2021, volume 2978 of CEUR Workshop Proceedings. CEUR-WS.org, 2021

C2 Roberta Capuano and Henry Muccini. A Systematic Literature Review on Migration to Microservices: a Quality Attributes perspective. In IEEE 19th International Conference on Software Architecture Companion, ICSA Companion 2022, Honolulu, HI, USA, March 12-15, 2022, pages 120–123. IEEE, 2022

C3 Roberta Capuano and Fabio Vaccaro. The Quality-Driven Refactoring Approach in BIM Italia. In IEEE 20th International Conference on Software Architecture Companion, ICSA Companion 2023, L'Aquila, IT, March 14-17, 2023. IEEE, 2023

C4 Roberta Capuano and Hnery Muccini. A Graph-based Java Projects Representation for Antipatterns Detection. In IEEE 17th International Conference on Software Architecture, ECSA 2023, Istanbul, Turkey, September 18-22, 2023. IEEE, 2023

Bibliography

[1] Bruno Góis Mateus, Matias Martinez, and Christophe Kolski. Learning migration models for supporting incremental language migrations of software applications. *Information and Software Technology*, 153:107082, 2023.

[2] M.M. Lehman, D.E. Perry, and J.F. Ramil. Implications of evolution metrics on software maintenance. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 208–217, 1998.

[3] Ravi Khadka, Belfrit V Batlajery, Amir M Saeidi, Slinger Jansen, and Jurriaan Hage. How do professionals perceive legacy systems and software modernization? In *Proceedings of the 36th International Conference on Software Engineering*, pages 36–47, 2014.

[4] Kent Beck, Martin Fowler, and Grandma Beck. Bad smells in code. *Refactoring: Improving the design of existing code*, 1(1999):75–88, 1999.

[5] Linda H Rosenberg and Lawrence E Hyatt. Software re-engineering. *Software Assurance Technology Center*, pages 2–3, 1996.

[6] Edmund C Arranga and Frank P Coyle. Cobol: Perception and reality. *Computer*, 30(3):126–128, 1997.

[7] Niels Veerman. Revitalizing modifiability of legacy assets. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(4-5):219–254, 2004.

[8] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and Giuseppe A Di Lucca. Decomposing legacy programs: A first step towards migrating to client–server platforms. *Journal of Systems and Software*, 54(2):99–110, 2000.

[9] Andrea De Lucia, Rita Francese, Giuseppe Scanniello, and Genoveffa Tortora. Developing legacy system migration methods and tools for technology transfer. *Software: Practice and Experience*, 38(13):1333–1364, 2008.

[10] Janet Lavery, Cornelia Boldyreff, Bin Ling, and Colin Allison. Modelling the evolution of legacy systems to web-based systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(1-2):5–30, 2004.

[11] Eleni Stroulia, Mohammad El-Ramly, and Paul Sorenson. From legacy to web through interaction modeling. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 320–329. IEEE, 2002.

[12] John Erickson and Keng Siau. Web services, service-oriented computing, and service-oriented architecture: Separating hype from reality. *Journal of database management (JDM)*, 19(3):42–54, 2008.

[13] Luciano Baresi and Martin Garriga. Microservices: the evolution and extinction of web services? *Microservices: Science and Engineering*, pages 3–28, 2020.

[14] Martin Fowler and James Lewis. Microservices. 2014. URL `http://martinfowler.com/articles/microservices.html`.

[15] Sam Newman. *Building microservices.* " O'Reilly Media, Inc.", 2021.

[16] Olaf Zimmermann. Do microservices pass the same old architecture test? or: Soa is not dead-long live (micro-) services. In *Microservices Workshop at SATURN conference, SEI*, 2015.

[17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216, 2017.

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

[19] 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, 2017.

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

[21] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. From monolith to microservices: A classification of refactoring approaches. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, March 5-6, 2018, Revised Selected Papers 1*, pages 128–141. Springer, 2019.

[22] Alexis Henry and Youssef Ridene. Migrating to microservices. *Microservices: Science and Engineering*, pages 45–72, 2020.

[23] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: A systematic approach to service decomposition. In *Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings 5*, pages 185–200. Springer, 2016.

[24] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information and software technology*, 131:106449, 2021.

[25] James D Herbsleb and Rebecca E Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE software*, 16(5):63–70, 1999.

[26] Martin Fowler and James Lewis. Strangler fig application. 2004. URL `https://martinfowler.com/bliki/StranglerFigApplication.html`.

[27] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information and software technology*, 131:106449, 2021.

[28] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Microservices in industry: insights into technologies, characteristics, and software quality. In *2019 IEEE international conference on software architecture companion (ICSA-C)*, pages 187–195. IEEE, 2019.

[29] Iso / iec 25010 : 2011 systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. 2013.

[30] Lianping Chen. Microservices: architecting for continuous delivery and devops. In *2018 IEEE International conference on software architecture (ICSA)*, pages 39–397. IEEE, 2018.

[31] Justas Kazanavičius and Dalius Mažeika. Migrating legacy software to microservices architecture. In *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–5. IEEE, 2019.

[32] David Jaramillo, Duy V Nguyen, and Robert Smart. Leveraging microservices architecture by using docker technology. In *SoutheastCon 2016*, pages 1–5. IEEE, 2016.

[33] Roberta Capuano and Henry Muccini. A systematic literature review on migration to microservices: a quality attributes perspective. In *2022 IEEE 19th*

International Conference on Software Architecture Companion (ICSA-C), pages 120–123. IEEE, 2022.

[34] Roberta Capuano. Enhancing system quality attributes via microservices adoption. 2021.

[35] Henry Muccini Roberta Capuano. A graph-based java projects representation for antipatterns detection. In *ECSA2023*, 2023.

[36] Roberta Capuano and Fabio Vaccaro. The Quality-Driven Refactoring Approach in BIM Italia. In *IEEE 20th International Conference on Software Architecture Companion, ICSA Companion 2023, L'Aquila, IT, March 14-17, 2023*. IEEE, 2023.

[37] 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. IEEE, 2018.

[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. IEEE, 2019.

[39] Thelma Colanzi, Aline Amaral, Wesley Assunção, Arthur Zavadski, Douglas Tanno, Alessandro Garcia, and Carlos Lucena. Are we speaking the industry language? the practice and literature of modernizing legacy systems with microservices. In *15th Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 61–70, 2021.

[40] Xin Zhou, Shanshan Li, Lingli Cao, He Zhang, Zijia Jia, Chenxing Zhong, Zhihao Shan, and Muhammad Ali Babar. Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry. *Journal of Systems and Software*, 195:111521, 2023.

[41] Luiz Carvalho, Alessandro Garcia, Wesley KG Assunção, Rafael de Mello, and Maria Julia de Lima. Analysis of the criteria adopted in industry to extract microservices. In *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, pages 22–29. IEEE, 2019.

[42] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: a systematic mapping study. In *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018*. SciTePress, 2018.

[43] Staffs Keele et al. Guidelines for performing systematic literature reviews in software engineering, 2007.

[44] Shahbaz Ahmed Khan Ghayyur, Abdul Razzaq, Saeed Ullah, and Salman Ahmed. Matrix clustering based migration of system application to microservices architecture. *International Journal of Advanced Computer Science and Applications*, 9(1):284–296, 2018.

[45] Luiz Carvalho, Alessandro Garcia, Wesley KG Assunção, Rodrigo Bonifácio, Leonardo P Tizzei, and Thelma Elita Colanzi. Extraction of configurable and reusable microservices from legacy systems: An exploratory study. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*, pages 26–31, 2019.

[46] Justas Kazanavičius and Dalius Mažeika. Migrating legacy software to microservices architecture. In *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–5. IEEE, 2019.

[47] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. Migrating from monolithic architecture to microservices: A rapid review. In *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7. IEEE, 2019.

[48] 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. IEEE, 2021.

[49] Santonu Sarkar, Gloria Vashi, and PP Abdulla. Towards transforming an industrial automation system from monolithic to microservices. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 1256–1259. IEEE, 2018.

[50] James Thomas and Angela Harden. Methods for the thematic synthesis of qualitative research in systematic reviews. *BMC medical research methodology*, 8(1): 1–10, 2008.

[51] Harris Cooper, Larry V Hedges, and Jeffrey C Valentine. *The handbook of research synthesis and meta-analysis*. Russell Sage Foundation, 2019.

[52] Antonin Smid, Ruolin Wang, and Tomas Cerny. Case study on data communication in microservice architecture. In *Proceedings of the Conference on Research in Adaptive and Convergent Systems*, pages 261–267, 2019.

[53] Nuno Santos, Carlos E Salgado, Francisco Morais, Mónica Melo, Sara Silva, Raquel Martins, Marco Pereira, Helena Rodrigues, Ricardo J Machado, Nuno Ferreira, et al. A logical architecture design method for microservices architectures. In *Proceedings of the 13th European Conference on Software Architecture-Volume 2*, pages 145–151, 2019.

[54] Hugo Henrique S da Silva, Glauco de F. Carneiro, and Miguel P Monteiro. An experience report from the migration of legacy software systems to microservice based architecture. In *16th International Conference on Information Technology-New Generations (ITNG 2019)*, pages 183–189. Springer, 2019.

[55] Adambarage Anuruddha Chathuranga De Alwis, Alistair Barros, Colin Fidge, and Artem Polyvyanyy. Availability and scalability optimized microservice discovery from enterprise systems. In *On the Move to Meaningful Internet Systems: OTM 2019 Conferences: Confederated International Conferences: CoopIS, ODBASE, C&TC 2019, Rhodes, Greece, October 21–25, 2019, Proceedings*, pages 496–514. Springer, 2019.

[56] Christoph Schröer, Sven Wittfoth, and Jorge Marx Gómez. A process model for microservices design and identification. In *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*, pages 1–8. IEEE, 2021.

[57] Mohsen Ahmadvand and Amjad Ibrahim. Requirements reconciliation for scalable and secure microservice (de) composition. In *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*, pages 68–73. IEEE, 2016.

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

[59] 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. IEEE, 2017.

[60] Sinan Eski and Feza Buzluca. An automatic extraction approach: Transition to microservices architecture from monolithic application. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pages 1–6, 2018.

[61] Manabu Kamimura, Keisuke Yano, Tomomi Hatano, and Akihiko Matsuo. Extracting candidates of microservices from monolithic application code. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 571–580. IEEE, 2018.

[62] Adambarage Anuruddha Chathuranga De Alwis, Alistair Barros, Artem Polyvyanyy, and Colin Fidge. Function-splitting heuristics for discovery of microservices in enterprise systems. In *Service-Oriented Computing: 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings 16*, pages 37–53. Springer, 2018.

[63] Andreas Christoforou, Lambros Odysseos, and Andreas S Andreou. Migration of software components to microservices: Matching and synthesis. 2019.

[64] Shanshan Li, He Zhang, Zijia Jia, Zheng Li, Cheng Zhang, Jiaqi Li, Qiuya Gao, Jidong Ge, and Zhihao Shan. A dataflow-driven approach to identifying microservices from monolithic applications. *Journal of Systems and Software*, 157:110380, 2019.

[65] Omar Al-Debagy and Peter Martinek. A new decomposition method for designing microservices. *Periodica Polytechnica Electrical Engineering and Computer Science*, 63(4):274–281, 2019.

[66] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Aymen Saied. Towards automated microservices extraction using muti-objective evolutionary search. In *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings 17*, pages 58–63. Springer, 2019.

[67] Ilaria Pigazzini, Francesca Arcelli Fontana, and Andrea Maggioni. Tool support for the migration to microservice architecture: An industrial case study. In *Software Architecture: 13th European Conference, ECSA 2019, Paris, France, September 9–13, 2019, Proceedings 13*, pages 247–263. Springer, 2019.

[68] Luís Nunes, Nuno Santos, and António Rito Silva. From a monolith to a microservices architecture: An approach based on transactional contexts. In *Software Architecture: 13th European Conference, ECSA 2019, Paris, France, September 9–13, 2019, Proceedings 13*, pages 37–52. Springer, 2019.

[69] Davide Taibi and Kari Systä. From monolithic systems to microservices: A decomposition framework based on process mining. 2019.

[70] Hugo HOS da Silva, Glauco F de Carneiro, and Miguel P Monteiro. Towards a roadmap for the migration of legacy software systems to a microservice based architecture. In *CLOSER 2019-Proceedings of the 9th International Conference on Cloud Computing and Services Science*, pages 37–47. SciTePress-Science and Technology Publications, 2019.

[71] Alexander Krause, Christian Zirkelbach, Wilhelm Hasselbring, Stephan Lenga, and Dan Kröger. Microservice decomposition via static and dynamic analysis of

the monolith. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 9–16. IEEE, 2020.

[72] Marx Haron Gomes Barbosa and Paulo Henrique M Maia. Towards identifying microservice candidates from business rules implemented in stored procedures. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 41–48. IEEE, 2020.

[73] 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 *Software Architecture: 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14–18, 2020, Proceedings 14*, pages 315–332. Springer, 2020.

[74] Justas Kazanavičius and Dalius Mažeika. Analysis of legacy monolithic software decomposition into microservices. 2020.

[75] 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. IEEE, 2020.

[76] Luiz Carvalho, Alessandro Garcia, Thelma Elita Colanzi, Wesley KG Assunção, Juliana Alves Pereira, Baldoino Fonseca, Márcio Ribeiro, Maria Julia de Lima, and Carlos Lucena. On the performance and adoption of search-based microservice identification with tomicroservices. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 569–580. IEEE, 2020.

[77] Chia-Yu Li, Shang-Pin Ma, and Tsung-Wen Lu. Microservice migration using strangler fig pattern: A case study on the green button system. In *2020 International Computer Symposium (ICS)*, pages 519–524. IEEE, 2020.

[78] Mohamed Daoud, Asmae El Mezouari, Noura Faci, Djamal Benslimane, Zakaria Maamar, and Aziz El Fazziki. Automatic microservices identification from a set of business processes. In *Smart Applications and Data Analysis: Third International Conference, SADASC 2020, Marrakesh, Morocco, June 25–26, 2020, Proceedings 3*, pages 299–315. Springer, 2020.

[79] 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, 2019.

[80] 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, 2021.

[81] Wesley KG Assunção, Thelma Elita Colanzi, Luiz Carvalho, Juliana Alves Pereira, Alessandro Garcia, Maria Julia de Lima, and Carlos Lucena. A multi-criteria strategy for redesigning legacy features as microservices: An industrial case study. In *2021 IEEE International conference on software analysis, evolution and reengineering (SANER)*, pages 377–387. IEEE, 2021.

[82] Mayank Mishra, Shruti Kunde, and Manoj Nambiar. Cracking the monolith: Challenges in data transitioning to cloud native architectures. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, pages 1–4, 2018.

[83] Alex Kaplunovich. Tolambda–automatic path to serverless architectures. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 1–8. IEEE, 2019.

[84] Teguh Prasandy, Dina Fitria Murad, Taufik Darwis, et al. Migrating application from monolith to microservices. In *2020 International Conference on Information Management and Technology (ICIMTech)*, pages 726–731. IEEE, 2020.

[85] 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, 2020.

[86] Athar Sheikh and Anisha Bs. Decomposing monolithic systems to microservices. In *2020 3rd International Conference on Computer and Informatics Engineering (IC2IE)*, pages 478–481. IEEE, 2020.

[87] Zhongshan Ren, Wei Wang, Guoquan Wu, Chushu Gao, Wei Chen, Jun Wei, and Tao Huang. Migrating web applications from monolithic structure to microservices architecture. In *Proceedings of the 10th Asia-Pacific Symposium on Internetware*, pages 1–10, 2018.

[88] Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde Lilia Bouziane, Christophe Dony, and Rahina Oumarou Mahamane. Re-architecting oo software into microservices: A quality-centred approach. In *Service-Oriented and Cloud Computing: 7th IFIP WG 2.14 European Conference, ESOCC 2018, Como, Italy, September 12-14, 2018, Proceedings 7*, pages 65–73. Springer, 2018.

[89] Atsushi Shimoda and Tsubasa Sunada. Priority order determination method for extracting services stepwise from monolithic system. In *2018 7th International Congress on Advanced Applied Informatics (IIAI-AAI)*, pages 805–810. IEEE, 2018.

[90] Alireza Goli, Omid Hajihassani, Hamzeh Khazaei, Omid Ardakanian, Moe Rashidi, and Tyler Dauphinee. Migrating from monolithic to serverless: A fintech case study. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, pages 20–25, 2020.

[91] Fola-Dami Eyitemi and Stephan Reiff-Marganiec. System decomposition to optimize functionality distribution in microservices with rule based approach. In *2020 IEEE International Conference on Service Oriented Systems Engineering (SOSE)*, pages 65–71. IEEE, 2020.

[92] 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. IEEE, 2020.

[93] Jakob Löhnertz and Ana-Maria Oprescu. Steinmetz: Toward automatic decomposition of monolithic software into microservices. In *SATToSE*, 2020.

[94] Deepali Bajaj, Urmil Bharti, Anita Goel, and SC Gupta. Partial migration for re-architecting a cloud native monolithic application into microservices and faas. In *Information, Communication and Computing Technology: 5th International Conference, ICICCT 2020, New Delhi, India, May 9, 2020, Revised Selected Papers*, pages 111–124. Springer, 2020.

[95] 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. IEEE, 2020.

[96] Holger Knoche. Sustaining runtime performance while incrementally modernizing transactional monolithic software towards microservices. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 121–124, 2016.

[97] Chen-Yuan Fan and Shang-Pin Ma. Migrating monolithic mobile application to microservice architecture: An experiment report. In *2017 ieee international conference on ai & mobile services (aims)*, pages 109–112. IEEE, 2017.

[98] Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde Lilia Bouziane, Rahina Oumarou Mahamane, Pascal Zaragoza, and Christophe Dony. From monolithic architecture style to microservice one based on a semi-automatic approach. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 157–168. IEEE, 2020.

[99] Heimo Stranner., Stefan Strobl., Mario Bernhart., and Thomas Grechenig. Microservice decompositon: A case study of a large industrial software migration in the automotive industry. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE,*, pages 498–505, 2020. doi: 10.5220/0009564604980505.

[100] Chang-ai Sun, Jing Wang, Jing Guo, Zhen Wang, and Li Duan. A reconfigurable microservice-based migration technique for iot systems. In *Service-Oriented Computing–ICSOC 2019 Workshops: WESOACS, ASOCA, ISYCC, TBCE, and STRAPS, Toulouse, France, October 28–31, 2019, Revised Selected Papers 17*, pages 142–155. Springer, 2020.

[101] Gabor Kecskemeti, Attila Kertesz, and Attila Csaba Marosi. Towards a methodology to form microservices from monolithic ones. In *Euro-Par 2016: Parallel Processing Workshops: Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers*, pages 284–295. Springer, 2017.

[102] Davide Taibi and Kari Systä. A decomposition and metric-based evaluation framework for microservices. In *Cloud Computing and Services Science: 9th International Conference, CLOSER 2019, Heraklion, Crete, Greece, May 2–4, 2019, Revised Selected Papers 9*, pages 133–149. Springer, 2020.

[103] Luciano Baresi, Martin Garriga, and Alan De Renzis. Microservices identification through interface analysis. In *Service-Oriented and Cloud Computing: 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings 6*, pages 19–33. Springer, 2017.

[104] Gojko Adzic and Robert Chatley. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 884–889, 2017.

[105] Jean-Philippe Gouigoux and Dalila Tamzalit. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *2017 IEEE international conference on software architecture workshops (ICSAW)*, pages 62–65. IEEE, 2017.

[106] Michel Cojocaru, Alexandru Uta, and Ana-Maria Oprescu. Microvalid: A validation framework for automatically decomposed microservices. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 78–86. IEEE, 2019.

[107] Andrea Janes and Barbara Russo. Automatic performance monitoring and regression testing during the transition from monolith to microservices. In *2019*

IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2019. doi: 10.1109/ISSREW.2019.00067.

[108] Luiz Carvalho, Alessandro Garcia, Thelma Elita Colanzi, Wesley K. G. Assunção, Maria Julia Lima, Baldoino Fonseca, Mãrcio Ribeiro, and Carlos Lucena. Search-based many-criteria identification of microservices from legacy systems. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 2020. doi: 10.1145/3377929.3390030.

[109] C Trubiani. Automated generation of architectural feedback from software performance analysis results. *Unpublished PhD thesis). Universita di L'Aquila. Retrieved from http://www. di. univaq. it/catia. trubiani/phDthesis/PhDThesis-CatiaTrubiani. pdf*, 2011.

[110] Connie U Smith and Lloyd G Williams. Software performance antipatterns. In *Proceedings of the 2nd international workshop on Software and performance*, pages 127–136, 2000.

[111] Connie U Smith and Lloyd G Williams. New software performance antipatterns: More ways to shoot yourself in the foot. In *Int. CMG Conference*, pages 667–674, 2002.

[112] Connie U Smith and Lloyd G Williams. Software performance antipatterns; common performance problems and their solutions. In *Int. CMG Conference*, pages 797–806. Citeseer, 2001.

[113] Mary Jean Harrold and Gregg Rothermel. A coherent family of analyzable graphical representations for object-oriented software. *Department of Computer and Information Science, The Ohio State University, Technical Report OSU-CISRC-11/96-TR60*, 1996.

[114] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proceedings of IEEE 18th international conference on software engineering*, pages 495–505. IEEE, 1996.

[115] Brian Mallo, John D McGregor, Anand Krishnaswamy, and Murali Medikonda. An extensible program representation for object-oriented software. *ACM Sigplan Notices*, 29(12):38–47, 1994.

[116] Rothermel and Harrold. Selecting regression tests for object-oriented software. In *Proceedings 1994 International Conference on Software Maintenance*, pages 14–25. IEEE, 1994.

[117] ESF Najumudheen, Rajib Mall, and Debasis Samanta. A dependence graph-based representation for test coverage analysis of object-oriented programs. *ACM SIGSOFT Software Engineering Notes*, 34(2):1–8, 2009.

[118] ESF Najumudheen, Rajib Mall, and Debasis Samanta. A dependence representation for coverage testing of object-oriented programs. *J. Object Technol.*, 9(4): 1–23, 2010.

[119] Jian-Jun Zhao. Applying program dependence analysis to java software. 1998.

[120] Neil Walkinshaw, Marc Roper, and Murray Wood. The java system dependence graph. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 55–64. IEEE, 2003.

[121] Gang Shu, Boya Sun, Tim AD Henderson, and Andy Podgurski. Javapdg: A new platform for program dependence analysis. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 408–415. IEEE, 2013.

[122] Tim AD Henderson and Andy Podgurski. Sampling code clones from program dependence graphs with graple. In *Proceedings of the 2nd international workshop on software analytics*, pages 47–53, 2016.

[123] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Microart: A software architecture recovery tool for maintaining microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 298–302. IEEE, 2017.

[124] Vittorio Cortellessa, Daniele Di Pompeo, Vincenzo Stoico, and Michele Tucci. Software model refactoring driven by performance antipattern detection. *ACM SIGMETRICS Performance Evaluation Review*, 49(4):53–58, 2022.

[125] Yasser A Khan and Mohamed El-Attar. Using model transformation to refactor use case models based on antipatterns. *Information systems frontiers*, 18:171–204, 2016.

[126] Jens Dietrich, Catherine McCartin, Ewan Tempero, and Syed M Ali Shah. On the existence of high-impact refactoring opportunities in programs. In *Proceedings of the Thirty-fifth Australasian Computer Science Conference-Volume 122*, pages 37–48, 2012.

[127] Catia Trubiani, Alexander Bran, André van Hoorn, Alberto Avritzer, and Holger Knoche. Exploiting load testing and profiling for performance antipattern detection. *Information and Software Technology*, 95:329–345, 2018.

[128] Catia Trubiani, Riccardo Pinciroli, Andrea Biaggi, and Francesca Arcelli Fontana. Automated detection of software performance antipatterns in java-based applications. *IEEE Transactions on Software Engineering*, 2023.

[129] Raed Shatnawi and Wei Li. An empirical assessment of refactoring impact on software quality using a hierarchical quality model. *International Journal of Software Engineering and Its Applications*, 5(4):127–149, 2011.

[130] Mohammad Alshayeb. Empirical investigation of refactoring effect on software quality. *Information and software technology*, 51(9):1319–1326, 2009.

[131] Konstantinos Stroggylos and Diomidis Spinellis. Refactoring–does it improve software quality? In *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*, pages 10–10. IEEE, 2007.

[132] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[133] Antonio Brogi, Davide Neri, and Jacopo Soldani. Freshening the air in microservices: resolving architectural smells via refactoring. In *Service-Oriented Computing–ICSOC 2019 Workshops: WESOACS, ASOCA, ISYCC, TBCE, and STRAPS, Toulouse, France, October 28–31, 2019, Revised Selected Papers 17*, pages 17–29. Springer, 2020.

[134] Rafik Tighilt, Manel Abdellatif, Naouel Moha, Hafedh Mili, Ghizlane El Boussaidi, Jean Privat, and Yann-Gaël Guéhéneuc. On the study of microservices antipatterns: A catalog proposal. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pages 1–13, 2020.

[135] Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. *CLOSER (1)*, pages 137–146, 2016.

[136] Robert B Fetter and Jean L Freeman. Diagnosis related groups: product line management within hospitals. *Academy of management Review*, 11(1):41–54, 1986.

[137] Julie A Schoenman, Janet P Sutton, Anne Elixhauser, and Denise Love. Understanding and enhancing the value of hospital discharge data. *Medical Care Research and Review*, 64(4):449–468, 2007.

[138] Robert B Fetter, Youngsoo Shin, Jean L Freeman, Richard F Averill, and John D Thompson. Case mix definition by diagnosis-related groups. *Medical care*, 18(2): i–53, 1980.

[139] Connie U Smith and Lloyd G Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer Measurement Group Conference*, pages 717–725. Citeseer, 2003.

[140] C. Aucion. How anti-patterns can stifle microservices adoption in the enterprise., 2018. URL `https://www.appdynamics.com/blog/engineering/how-to-avoid-antipatterns-with-microservices/`.

[141] J. Kanjilal. 4 microservices antipatterns that ruin migration., 2020. URL `https://www.techtarget.com/searchapparchitecture/tip/4-deadly-microservices-antipatterns-that-ruin-migration`.

[142] J. Kanjilal. Overcoming the common microservices anti-patterns., 2021. URL `https://www.appdynamics.com/blog/engineering/how-to-avoid-antipatterns-with-microservices/`.

[143] Shuai Jiang and Huaxin Mu. Design patterns in object oriented analysis and design. In *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, pages 326–329. IEEE, 2011.

[144] Erich Gamma, Ralph Johnson, Richard Helm, Ralph E Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[145] Narendra Babu Pamula, K Jairam, and B Rajesh. Cache-aside approach for cloud design pattern. *International Journal of Computer Science and Information Technologies*, 5(2):1423–1426, 2014.

[146] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Microservices anti-patterns: A taxonomy. pages 111–128, 2020.

[147] Rafik Tighilt, Manel Abdellatif, Naouel Moha, Hafedh Mili, Ghizlane El Boussaidi, Jean Privat, and Yann-Gaël Guéhéneuc. On the study of microservices antipatterns: A catalog proposal. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, EuroPLoP '20. Association for Computing Machinery, 2020.

[148] Andreas Grabner. Locating common micro service performance anti-patterns. *InfoQ*, 2016. URL `https://www.infoq.com/articles/Diagnose-Microservice-Performance-Anti-Patterns`.

Appendix A

SLR on Quality-Driven Migration to Microservices: Parameters

A.1 General parameters and Values

ID	Attribute Name	Type	Description
GEN-1	ID	Open	Unique Identifier of the study.
GEN-2	Item Type	Single	Type of the study associated by the Digital Library.
GEN-3	Title	Open	Title of the downloaded study.
GEN-4	Authors	Open	List of authors of the study.
GEN-5	Institution	Open	List of the institutions of the study as reported in the article itself.
GEN-6	Venue	Open	Acronym and complete name of the venue in which the study has been published.
GEN-7	Page Count	Open	Number of pages of the study.
GEN-8	Keywords	Open	Keywords as they appear in the downloaded study or in the Digital Library.
GEN-9	Digital Library	Multiple	Source library of the study.
GEN-10	DOI	Open	DOI of the paper.

Table A.1: *SLR*: General Parmeters and Descriptions.

A.2 RQ1-related Parameters and Values

ID	Attribute Name	Type	Description
MT-1	Publication Year	Single	Year of publication of the study as reported by the Digital Library.
MT-2	Old Software Architecture	Open	The architectural style of the system to be migrated.
MT-3	New Architecture	Multiple	
MT-4	Old Programming Language	Open	The programming language of the system to be migrated.
MT-5	Application Domain	Open	Describe the domain of the system that generates the approach.

Table A.2: *SLR*: *RQ1* Parmeters and Descriptions.

A.3 RQ2-related Parameters and Values

ID	Attribute Name	Type	Description
QA-1	Quality Attributes	Open	List of Quality Attributes considered in the work.
QA-2	QA in Migration Phase	Multiple	Phase of the migration in which Quality Attributes are considered.

Table A.3: *SLR*: *RQ2* Parmeters and Descriptions.

QA-2 Value	Description
Comprehension	Quality Attributes are considered in the Comprehension Phase
MS Identification	Quality Attributes are considered in the Micoservices Identification Phase
MS Packaging	Quality Attributes are considered in the Packaging Phase
MS Assessment	Quality Attributes are considered after the microservices implementation

Table A.4: *SLR*: QA-2 - Quality Attributes in Migration Phase Values.

A.4 RQ3-related Parameters and Values

ID	Attribute Name	Type	Description
MS-1	System Comprehansion Approach	Multiple	Describes the techniques used for the comprehension of the system.
MS-2	System Comprehansion Tools	Open	Describes the tools used for the comprehension of the system.
MS-3	MS Identification Approach	Multiple	Describes the techniques used for the microservices identification.
MS-4	MS Identification Tools	Open	Describes the tools used for the microservices identification.
MS-5	System Packaging Approach	Open	Describes the techniques used for the packaging of the system.
MS-6	System Packaging Tools	Open	Describes the tools used for the packaging of the system.

Table A.5: *SLR*: *RQ3* Parmeters and Descriptions.

MS-1 Value	Description
Static Analysis	The comprehension has been done analysing: code, dependences, classes, methods, packages, files, directories, text analysis
Dynamic Analysis	The comprehension has been done analysing: logs, traces, use case testing, user experience testing, user stories testing
Model-Driven Analysis	The comprehension has been done using models and metamodels of the system
Data-Driven Analysis	The comprehension has been done analysing data
Domain-Driven Analysis	The comprehension has been done analysing: boundend context, business process, requirements analysis

Table A.6: *SLR*: MS-1 - System Comprehension Approach Values

MS-3 Value	Description
Static Analysis	The microservices identification approach is based on: code, dependences, classes, methods, packages, files, directories, text analysis
Dynamic Analysis	The microservices identification approach is based on: logs, traces, use case testing, user experience testing, user stories testing
Model-Driven Analysis	The microservices identification approach is based on: models and meta-models of the system
Data-Driven Analysis	The microservices identification approach is based on data
Domain-Driven Analysis	The microservices identification approach is based on: boundend context, business process, requirements analysis
Machine Learning/Optimization Driven	The microservices identification approach is based on clustering techniques

Table A.7: *SLR*: MS-3 - System Comprehension Approach Values

A.5 Other Parameters and Values

ID	Attribute Name	Type	Description
OP-1	Main Topic	Single	Describes the main topic of the study.
OP-2	Evaluation	Single	Describes if the approach shown in the study has been evaluated.
OP-3	Automated Process	Single	Describes if the migration strategy is automated in the study.
OP-4	Recommendation	Single	

Table A.8: *SLR*: Other Parmeters and Descriptions.

OP1 Value	Description
Migration to Microservices	The paper refers to migration to microservices
Migration to Cloud	The paper refers to migration to cloud
Migration to Microservices and Quality	The paper refers to migration to microservices considering quality attributes in the comprehension or in the microservices identification phase
Migration to Cloud and Quality	The paper refers to migration to cloud considering also quality aspects

Table A.9: *SLR*: OP1 - Main Topic - Values

OP2 Value	Description
Empirical	Authors shows the approach for migration
Case Study	In two cases: i) authors show the approach and apply it on a case study; ii) authors show the lesson learned from migration

Table A.10: *SLR*: OP2 - Validation Type - Values

Appendix B

Antipatterns Detected in the QuaniSDO Software in BIM Italia

B.1 Antipatterns Detected on the Monolith - Control Functionality

God Class	
Description	Occurs when there is one (or more) class(es) that performs most of the work. The associated other classes are relegated to minor, supporting role. A variation is described as a class that contains all the system's data [139].
Causes	Poorly distributed system intelligence, i.e. a poor design that splits data from the relative processing logic [110].
Identification	The responsibility to perform the check is assigned mostly to two classes.

Table B.1: God Class - Control Functionality - Monolith

Circuitous Treasure Hunt	
Description	Occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each "look," performance will suffer [112].
Causes	Use of an inadequate architecture. Subdivision of tasks and structures into small parts that could be merged into the same structure [110].
Identification	Each check performed needs data and structures that are scattered in different parts of the code.

Table B.2: Circuitous Treasure Hunt - Control Functionality - Monolith

Concurrent Processing	
Description	Occurs when processing cannot make use of available processors [139].
Causes	i) non-balanced assignment of tasks to processors ii) single-threaded code [111].
Identification	The monolith process the check one *HDR* at a time. This causes an increase in response time while more than one *HDR* has to be checked with respect to the locar authority rules.

Table B.3: Concurrent Processing - Control Functionality - Monolith

Pipe and Filter	
Description	Occurs when the slowest filter in a "pipe and filter" architecture causes the system to have unacceptable throughput [139].
Causes	There is a stage in a pipeline which is significantly slower than all the others [111].
Identification	Each *HDR* is filtered more than once based on the check to be performed to retreive the data required.

Table B.4: Pipe and Filter - Control Functionality - Monolith

Extensive Processing	
Description	Occurs when extensive processing in general impedes overall response time [139].
Causes	A long running process monopolizes a processor and prevents a set of other jobs to be executed until it finishes its computation [111].
Identification	The check of each *HDR* showed an extensive use of CPUs to retreive the data required.

Table B.5: Extensive Processing - Control Functionality - Monolith

Onle-Lane Bridge	
Description	Occurs at a point in execution where only one, or a few, processes may continue to execute concurrently (e.g., when accessing a database). Other processes are delayed while they wait for their turn [139].
Causes	Concurrent systems when the mechanisms of mutual access to a shared resource are badly designed [139].
Identification	The access to the database is not parallelized.

Table B.6: Onle-Lane Bridge - Control Functionality - Monolith

Excessive Dynamic Allocation	
Description	Occurs when an application unnecessarily creates and destroys large numbers of objects during its execution [139].
Causes	Poor design solutions adopted to address flexibility ("Just-in-time" approach) [110].
Identification	The data structure are created on-the-fly and destroyed once each query has been performed.

Table B.7: Excessive Dynamic Allocation - Control Functionality - Monolith

Tower of Babel	
Description	Occurs when processes excessively convert, parse, and translate internal data into a common exchange format [139].
Causes	The same information is often translated into an exchange format (by a sending process) and then parsed and translated into an internal format (by the receiving process) [139].
Identification	The data has been converted at least two times: from data to *DTS* and viceversa.

Table B.8: Tower of Babel - Control and Pricing Functionalities - Monolith

The Ramp	
Description	Occurs when processing time increases as the system is used [139].
Causes	It is due to the growing amount of data the system stores and, as the time goes on, the data grow and the processing time required to perform an operation on such data becomes unacceptable [139].
Identification	The control functionality on the monolith suffers from memory leaks causing a growing memory utilization.

Table B.9: The Ramp - Control Functionality - Monolith

More is Less	
Description	Occurs when a system spends more time thrashing than accomplishing real work because there are too many processes relative to available resources [139].
Causes	i) running too many processes over time causes too much paging and too much overhead for servicing page faults ii) too many database connections are created by causing a significant performance loss iii) too many internet connections or too many pooled resources are allowed [139].
Identification	For each *HDR* to be checked, the monolith performs different database connections causing performance loss.

Table B.10: More is Less - Control Functionality - Monolith

B.2 Antipatterns Detected on the Microservices

Data Taffy	
Description	All services have full access to all objects in the database. This is also referred as Entangled Data [140, 141]. This antipatterns is also known as Shared Persistence [146, 147].
Causes	Lots of stored procedures, embedded complex queries, and object relationship managers all accessing the database [142].
Identification	The access to data is always complete regardless the invocation.

Table B.11: Data Taffy - Control and Pricing Functionalities - Microservice

High Service Network Payload	
Description	Lack of bandwidth in the cloud environment that causes high latency in the communication [148].
Causes	The size of the data transferred between internal service calls is bigger than the data sent to the user [148].
Identification	The data required to perform the checks on a single *HDR* are bigger than the data representing the check outcome.

Table B.12: High Service Network Payload - Control Functionality - Microservice

N+1 Service Call	
Description	Occurs when the results of a service requires an additional numbers of different calls [148].
Causes	The same operation, e.g. db query, is performed more than once to require all the data needed [148].
Identification	Instead of performing a single database request, many different queries are created to retreive data required for each check on the same *HDR*.

Table B.13: N+1 Service Call - Control Functionality - Microservice

Traffic Jam	
Description	Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared [139].
Causes	Large amount of work is scheduled within a relatively small interval. Eg. when a huge number of processes are originated at approximately the same time [112].
Identification	Checks are not bound. This implies that when a check references the null pointer, the entire *HDR* check process slows down.

Table B.14: Traffic Jam - Control Functionality - Microservice

La borsa di dottorato è stata cofinanziata con risorse del Programma Operativo Nazionale 2014-2020 (CCI 2014IT16M2OP005), Fondo Sociale Europeo, Azione I.1 “Dottorati Innovativi con caratterizzazione industriale”