INCREMENTAL MIGRATION FROM MONOLITHIC TO MICROSERVICES ARCHITECTURE

A SEMI-AUTOMATED APPROACH FOR INCREMENTAL MIGRATION FROM MONOLITHIC TO MICROSERVICES ARCHITECTURE

By HASSAN ZAKER ZAVERDEHI, MASc

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Applied Science

McMaster University

MASTER OF APPLIED SCIENCE (2024)

Hamilton, Ontario, Canada (Department of Computing and software)

TITLE:	A Semi-Automated Approach for Incremental Migration from Monolithic to Microservices Architecture
AUTHOR:	Hassan Zaker Zaverdehi MASc (Software Engineering), McMaster University, Hamilton, Canada
SUPERVISOR:	Richard Paige
NUMBER OF PAGES:	xv, 75

Lay Abstract

As software applications grow, maintaining and updating them becomes challenging. Traditionally, software is built using a **monolithic** architecture, where all parts are combined into a single unit. This can lead to problems as the software gets bigger. A newer approach, called **Microservice Architecture**, breaks down the application into smaller, independent services, each handling a specific function. This makes the software easier to manage and scale.

This thesis introduces a tool to help transition from monolithic to microservices architecture gradually, using the **Strangler Fig Pattern**. This approach involves incrementally creating microservices from the monolith, ensuring the system remains functional throughout the process. By combining different types of analysis, the tool accurately identifies potential microservices. It was tested on well-known projects and showed promising results in creating efficient and well-organized microservices. This work offers a practical solution for modernizing large software systems, making them easier to maintain and scale.

Abstract

As software applications grow in size and complexity, maintaining and scaling them becomes increasingly difficult. Traditional **monolithic** architectures, where all components are combined into a single unit, often face issues such as limited scalability, cumbersome maintenance, and problematic deployment. The **microservices** architecture has emerged as a solution, breaking down applications into smaller, loosely coupled services, each responsible for a specific business function. This transition process, known as migration, is challenging due to the difficulty in determining the optimal decomposition of the monolithic system.

This thesis presents a novel framework designed to facilitate the migration from monolithic to microservices architecture using the **Strangler Fig Pattern**. Unlike existing approaches that typically attempt to decompose the monolith in a single iteration, our framework supports a gradual, iterative migration process. This allows for smoother transitions, reduced risk, and better management of complexity.

Key contributions of this work include the development of a tool that leverages both static and dynamic analysis to identify microservice candidates. The tool integrates these heterogeneous data sources using the **Single Source of Truth** (SST) paradigm, ensuring consistency and reliability. The performance of the tool is evaluated on two well-known Java Spring projects, demonstrating its effectiveness in

creating well-modularized, cohesive, and loosely coupled microservices.

The results show that our approach not only meets the desired principles of microservice architectures but also compares favorably with other state-of-the-art methods. By providing a practical and systematic solution for gradual migration, this thesis addresses a significant gap in the existing literature and offers valuable insights for practitioners seeking to modernize large-scale software systems.

To my family,

for their unwavering support, encouragement,

and love throughout this journey.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Richard Paige, for his invaluable guidance, support, and encouragement throughout the course of this research. His insightful feedback and constant motivation were crucial to the completion of this thesis. I would also like to extend my heartfelt thanks to my co-supervisors, Dr. Vera Pantelic and Dr. Sebastien Mosser, for their mentorship and sound guidance. Their advice and expertise have greatly contributed to the development and success of this work.

I also want to extend my heartfelt thanks to my colleagues at McMaster University. Their collaboration, support, and camaraderie made this journey a memorable and enriching experience.

Finally, I am deeply indebted to my family and friends for their unwavering support and understanding throughout my studies. Their patience and encouragement have been my driving force.

Thank you all for your contributions and support.

Table of Contents

List of Figures

List of Tables

Notation, Definitions, and Abbreviations

Definitions

Granularity Granularity refers to the size or scope of individual microservices. It describes how finely a system is decomposed into separate services. A higher level of granularity means that services are smaller and more focused on specific tasks or functionalities, while a lower level of granularity indicates larger services that encompass multiple functionalities.

Bounded Context

In Domain-Driven Design (DDD), a bounded context defines the specific scope within which a domain model applies, encapsulating its models, language, and rules. It defines boundaries to manage complexity and ensure clarity and consistency within a specific domain or subdomain.

Community Detection Problem

In graph theory, the community detection problem refers to the task of identifying cohesive groups or communities within a network. These communities are subsets of nodes with dense internal connections and sparser connections between different communities. The goal is to partition the nodes of the graph into these distinct communities to reveal underlying structures or functional modules within the network.

Modularity Score

In community detection algorithms, the modularity score quantifies the quality of a division of a network into communities. It measures the density of connections within communities compared to connections between communities, aiming to identify densely connected groups of nodes.

Integer Linear Programming (ILP)

A mathematical optimization technique used to solve optimization problems where variables are required to be integer values. It involves formulating a mathematical model with linear relationships and integer variables, aiming to maximize or minimize an objective function while satisfying a set of constraints.

Multiway Cut Problem

The Multiway Cut Problem involves partitioning the nodes of a graph into multiple disjoint sets to minimize the total weight of edges

cut across the partitions.

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CSV	Comma-Separated Values
DDD	Domain-Driven Design
ILP	Integer Linear Programming
JSON	JavaScript Object Notation
SCA	Static Code Analysis
SST	Single Source of Truth
XML	eXtensible Markup Language

Declaration of Academic Achievement

I hereby declare that this thesis is the result of my own work and has not been submitted in any form for another degree or diploma at any university or other institution of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

I certify that this thesis is an original work of research, carried out under the supervision of Dr. Richard Paige and co-supervisors Dr. Vera Pantelic and Dr. Sebastien Mosser. This work was carried out by McSCert (McMaster Centre For Software Certification)[1].

This work is a reflection of my own efforts, contributions, and academic achievements in the field of software certification and analysis. The data and conclusions presented in this thesis are accurate and reliable to the best of my knowledge.

Hassan Zaker Zavardehi

[1] `https://mcscert.mcmaster.ca/`

Chapter 1

Introduction

1.1 Background and Motivation

A **monolithic architecture** refers to a software design pattern in which all components of an application are combined into a single, self-contained unit. This architectural style typically involves a single executable that includes the user interface, business logic, and data access layers. Monolithic applications can be straightforward to develop, test, and deploy, as all parts of the application are interconnected and run as a single process [10]. However, as these applications expand in size and complexity, they face several significant challenges. Scaling becomes difficult, as it requires the entire application to be scaled even if only a small portion needs additional resources. Maintenance becomes increasingly cumbersome due to the tightly coupled nature of the components, making it challenging to isolate and address issues. Furthermore, deployment becomes problematic because any modification, regardless of its size, necessitates redeploying the entire application, increasing the risk of downtime and deployment errors [7, 15].

To address these challenges, the **microservice** architecture has emerged as a viable solution. In contrast to the monolithic approach, microservice architecture breaks down an application into a collection of small, loosely coupled services, each responsible for a specific business function. These services can be developed, deployed, and scaled independently, allowing for greater flexibility and efficiency. By decoupling components, microservice architecture simplifies maintenance and enhances the scalability of individual services. Deployment is also streamlined, as updates to one service do not necessitate redeploying the entire application, thereby reducing the risk of downtime and deployment errors [24].

Not all applications with monolithic architecture experience scalability and maintainability challenges. Less complex monolithic applications have their own strengths, such as being easier to develop, test, deploy, and scale. This architecture has existed long before the advent of microservice architecture. Consequently, many companies have built their software using the monolithic approach. As their software grows in size and complexity, companies increasingly recognize the need to shift to microservice architecture to address the aforementioned challenges. This transformation process, where a monolithic application is restructured into a collection of microservices, is known as **migration**.

The primary challenge in transitioning from a monolithic system to a microservice architecture lies in determining an effective decomposition of the system. The objective is to decompose the monolithic application into several smaller components, each dedicated to a specific functionality, while minimizing the coupling between these microservices. Numerous researchers have proposed various Service Candidate

Identification approaches to facilitate or automate the decomposition process. Identifying services within the monolith is a complex task [26]. Migrating from a legacy monolithic system to a microservice architecture is inherently complex and time-consuming [9]. A survey conducted with 16 practitioners highlighted that identifying the appropriate decomposition approach is the primary challenge in the migration process [11]. Another study further emphasizes that a significant obstacle for companies with existing monolithic systems is the decomposition of these systems into cohesive microservice-based implementations [1].

Trabelsi et al. [28] aim to automate the microservice identification process by combining semantic analysis with machine learning algorithms. Gysel et al. [13] introduce **Service Cutter**, which employs a graph-based clustering approach to identify microservices. Kaminura et al. [16] propose a methodology for identifying microservice candidates within monolithic application code through static code analysis (SCA). Brito et al. [6] present a novel approach for identifying microservices within monolithic applications using topic modeling.

Filippone et al. [9] identify the main challenges of previous works on service identification as follows: the identification of highly cohesive and loosely coupled microservices with the right granularity and within the appropriate bounded context, and the requirement for user input. They assert that their proposed approach is fully automated, which means it does not require user input, and is capable of finding microservices with the right granularity.

Although the aforementioned approaches may achieve acceptable system decomposition, their primary limitation is that they attempt to identify microservices in a **single** iteration. In real-world industrial migration scenarios, it is uncommon to

transform a monolith into microservices in one step. A survey conducted by [19], involving 23 practitioners engaged in migrating their company's systems, revealed that none of the participants attempted to decompose the monolith into microservices in a single step. Instead, they adopted an incremental process, implementing new microservices one by one. Throughout this process, the monolith continues to function until the end, with new microservices being incrementally extracted from it. The monolith gradually reduces in size until it completely disappears, leaving only the microservices. This process aligns with the well-known **Strangler Fig Pattern** or **Strangler Pattern**, where new microservices **strangle** and replace the old monolith components over time. This incremental approach facilitates smoother migration, reduces risk, and better manages complexity compared to the **Big Bang Pattern**, where all microservices replace the monolith at once. The only study we found that employed the Strangler Pattern was conducted by Li et al. [18]. They propose an approach for incremental migration using the Strangler Pattern and Domain-Driven Design (DDD) principles for microservice identification. A key difference between their approach and ours is that they determine the decomposition of the monolithic project only once at the beginning of the process, before the first iteration. In contrast, our approach involves finding the decomposition of the monolithic project before each iteration, reassessing and refining the decomposition based on the latest state of the monolith.

In conclusion, as monolithic systems grow in size and complexity, maintaining and scaling them becomes increasingly challenging. Migrating to a microservices architecture can alleviate these issues. However, a significant challenge lies in identifying the appropriate decomposition during the migration process. Most approaches in the

literature follow the **big bang** pattern. Although Li et al. [18] adopt the **strangler** pattern, they determine the decomposition of the system only once, which is more aligned with the big bang pattern. In this work, we introduce a framework designed to facilitate the migration process from monolith to microservices in an incremental manner, where the decomposition of the system is determined in each iteration.

1.2 Problem Statement

Despite the advancements made by approaches such as those by Trabelsi et al., Li et al., Matias et al., Gysel et al., Kaminura et al., and Brito et al., which leverage semantic analysis, machine learning, graph-based clustering, and topic modeling respectively, they often overlook the iterative nature of practical migration processes. Filippone et al.'s automated approach, although innovative, still operates under a single iteration model, which does not align with the gradual, iterative process needed in industry settings.

Moreover, even methodologies that attempt to utilize the strangler pattern, such as the work by Li et al., are not fully committed to this pattern. Li et al. determine the decomposition of the monolith only once, before the first iteration. The challenge remains to develop an approach that can effectively guide the decomposition of monolithic systems into microservices through an iterative, step-by-step process, where the new microservices are introduced based on the latest monolith version, ensuring minimal disruption and maintaining system functionality throughout the migration.

This research aims to address these gaps by introducing a framework that supports the gradual migration of monolithic systems to microservices. The proposed

framework emphasizes iterative identification and implementation of microservices, ensuring that each step in the migration process is manageable and aligned with the overall goal of achieving a well-structured, efficient microservices architecture.

1.3 Contributions

This research aims to address the challenges associated with migrating from monolithic architectures to microservices using the Strangler approach. The primary contributions of this study are as follows:

1. **Develop a Tool for Incremental Migration**:
    - Designed and implemented a tool that supports the **incremental** decomposition of monolithic systems into microservices. This tool complements the expertise of practitioners, facilitating a smoother and more manageable migration process. The tool reduces the complexity and risk associated with monolith to microservices migration, and provides a framework for migrating in an incremental manner.
2. **Integrate Static and Dynamic Data**:
    - Extended existing state-of-the-art methodologies that primarily rely on static code analysis by incorporating dynamic analysis data. This integration enhances the robustness and accuracy of the decomposition process, providing a more comprehensive view of system behavior and interactions, ultimately leading to better-designed microservices.
3. **Ensure Data Consistency and Reliability**:

- Addressed the challenge of ensuring the reliability and consistency of heterogeneous data sources by employing the Single Source of Truth (SST) paradigm. Demonstrated the application and effectiveness of SST in maintaining data integrity during the migration process. This approach ensures that decisions are based on accurate and up-to-date information, minimizing errors and manual effort. It also provide data accessibility through the migration process.

4. **Evaluate Tool Performance**:

   - Assessed the performance of the developed tool on benchmark projects from the literature. Conducted a comparative analysis with other state-of-the-art approaches to demonstrate the effectiveness and improvements achieved by the proposed methodology. This evaluation provides evidence of the tool's practical benefits, such as improved decomposition quality.

By achieving these objectives, this research contributes a practical and theoretically sound framework for the incremental migration of monolithic systems to microservices. The proposed framework addresses a crucial gap in existing literature, offering a solution that is both effective and accessible to practitioners, ultimately improving the scalability, maintainability, and resilience of software systems.

1.4 Thesis Structure

This thesis is organized into five chapters, each addressing different aspects of the research:

Chapter 1: Introduction Provides the background, problem statement, research objectives, scope, methodology, and thesis structure.

Chapter 2: Previous Works Reviews related work in the field of microservice migration and identifies gaps in the existing research.

Chapter 3: Technical Chapter Details the algorithm and enhancements made to the tool, the integration of static and dynamic analysis, and the challenges encountered.

Chapter 4: Evaluation Presents the experimental setup, metrics used, and results obtained from applying the tool to case studies.

Chapter 5: Conclusion Summarizes the findings, discusses the implications of the research, and suggests directions for future work.

Chapter 2

Related Work

2.1 Introduction

The term **microservices** was first popularized by Fowler and Lewis. In their influential article "Microservices: a definition of this new architectural term," published on Martin Fowler's blog in March 2014[1], they provided a formal definition and outlined the principles and characteristics of microservice architecture. While the concept of breaking down applications into smaller, more manageable components existed prior to their work, it was Fowler and Lewis who brought widespread attention to the term microservices and formalized the architectural style associated with it [10].

The transition from monolithic to microservice architecture has emerged as a critical area of research in software engineering. This shift is driven by the need to enhance the scalability, maintainability, and flexibility of large and complex software systems. As monolithic applications grow, they face scalability and maintainability issues. To address these issues, the microservices architectural style has been proposed, which

[1] https://martinfowler.com/articles/microservices.html

allows for decomposing a monolithic application into smaller, loosely coupled services that can be developed, deployed, and scaled independently [8].

Dragoni *et al.* [8] provide a detailed analysis of how microservices can be designed and implemented to address the limitations of traditional monolithic architectures. They introduce scalability and maintainability as the most important features provided by the microservice paradigm. Their work highlights the benefits of adopting a microservice architecture, including improved fault isolation, continuous delivery, and the ability to deploy components independently.

This chapter reviews the key contributions in the field of microservice migration, focusing on the challenges and solutions associated with transitioning from monolithic to microservice architectures. The related works are organized into several sections: **Challenges in Migration**, which discusses the technical and organizational challenges faced during migration; **Approaches for Microservice Identification**, which reviews various methodologies for identifying microservice candidates; **Gradual Migration**, which explores works which have studied incrementally transitioning to microservices; and **Single Source of Truth**, which addresses the importance of maintaining a unified data store.

2.2 Challenges in Migration

Migrating from monolithic systems to microservice architectures presents numerous challenges that organizations must consider to achieve successful transitions. These challenges encompass both technical and organizational aspects, requiring careful planning and execution [8, 13, 16].

Fritzsch *et al.* [11] conduct an empirical study on the intentions, strategies, and

challenges faced by organizations transitioning from monolithic systems to microservice architectures. Surveying 23 companies, the study identifies the primary motivations for migration as achieving greater scalability, improving maintainability, and enabling faster development cycles. The study defines two migration strategies: (i) the Big Bang approach and (ii) the iterative approach. Notably, none of the surveyed organizations used the Big Bang approach due to its complexity and time-consuming nature. Instead, the iterative approach, or Strangler pattern, is commonly adopted and is the basis for the migration strategy in our tool. The study also highlights several challenges, including the complexity of decomposition, managing data consistency across distributed services, cultural and organizational changes, and technical debt. Our research, along with other studies mentioned in the following section, focuses on addressing the first challenge, decomposition complexity.

Salii *et al.* [25] provide an extensive analysis of the benefits and challenges associated with migrating from monolithic architectures to microservices through real-world case studies from various organizations. The study identifies key benefits, including improved scalability, better fault isolation, enhanced development and deployment agility, and the ability to perform independent updates, leading to quicker release cycles and reduced time-to-market. However, the migration process also presents significant challenges, such as increased complexity in managing a distributed system, ensuring data consistency and transactional integrity, and requiring substantial organizational and cultural changes. Additionally, dealing with existing technical debt and the performance overhead of inter-service communication are noted as obstacles.

2.3 Approaches for Microservice Identification

The identification of microservices within existing software systems has been a significant focus of research in recent years. Numerous approaches have been proposed to extract microservice candidates from monolithic architectures, each employing various techniques to achieve this goal [9, 13, 16, 19, 28]. These methodologies aim to identify cohesive, loosely coupled services that align with business functionalities and can be independently developed, deployed, and scaled. The following sections will explore several key approaches for microservice identification, highlighting their methodologies, advantages, and limitations.

Filippone *et al.* [9] propose a method for identifying microservices within a monolithic system using graph clustering and combinatorial optimization. They utilize static code analysis (SCA) to create a knowledge graph that includes entities and methods as nodes. The Louvain algorithm is then applied to find communities within the knowledge graph and a subgraph that consists only of entities as nodes. Subsequently, they optimize the microservice architecture using an Integer Linear Programming (ILP) model. In this thesis, we build upon the work of Filippone *et al.* [9] by employing their approach as a foundation for iteratively identifying microservices. Unlike their single-iteration method for identifying microservice candidates, we adopt a gradual migration pattern for the identification of new microservices.

Trabelsi *et al.* [28] introduce a type-based approach for identifying microservices from legacy monolithic applications. The main contribution of this paper is the combination of machine learning and semantic analysis to automate the microservice identification process. By analyzing the types and relationships within the legacy codebase, the proposed method uses machine learning models to detect patterns and

dependencies that suggest potential microservice boundaries. Similar to our approach, they utilize a graph-based representation to identify microservices. However, unlike our iterative method, their migration is executed in a single iteration.

Gysel *et al.* [13] introduce a systematic approach for identifying microservice candidates from monolithic applications, called **Service Cutter**. Similar to our method, Service Cutter utilizes a graph-based clustering approach to identify microservices. However, it differs by using software specification artifacts (SSAs) such as use cases and Entity-Relationship models as input, whereas our approach relies on source code as the input. Their methodology applies a set of 16 coupling criteria to construct an undirected weighted graph, which is subsequently analyzed to identify potential microservice candidates.

Kaminura *et al.* [16] present a methodology for identifying microservice candidates within monolithic application code through static code analysis (SCA). Similar to our approach, this paper leverages SCA to extract semantic information for clustering using proximity measures or topic modeling. Their method employs a mix of annotations and a naming convention to classify different types of nodes; for example, methods containing *set* or *add* in their names are identified as being responsible for writing data. In contrast, our approach uses a comprehensive list of annotations to determine node types, which provides a more standardized identification process. A notable limitation of Kaminura *et al.*'s work is the lack of consideration for the granularity of microservices during the decomposition process. In comparison, our methodology explicitly addresses microservice granularity to ensure that the services are appropriately sized.

Li *et al.* [19] proposed a novel method for extracting microservices from monolithic applications by constructing a knowledge graph. Similar to our approach, they build a knowledge graph, but their graph focuses solely on entity-entity relationships. In contrast, our method includes method-entity and method-method relationships, providing a more detailed representation. Additionally, unlike our approach, Li *et al.*'s method requires manual input from users.

Brito *et al.* [6] present a novel approach for identifying microservices within monolithic applications using topic modeling. They create a knowledge graph based on semantic analysis. By extracting topics that represent coherent clusters of functionalities, their approach helps to identify potential microservice boundaries that are aligned with the business logic and domain-specific concepts of the application. In this approach, the weight of nodes in the graph is determined by the topic distribution similarity between nodes. Similar to our method, they use graph representation and clustering algorithms to identify microservices.

In terms of using static and dynamic data at the same time, Matias *et al.* [20] present a case study focused on determining microservice boundaries through the combined use of static and dynamic software analysis techniques. The main contribution of this paper is the demonstration of how this combined approach can effectively identify cohesive microservice candidates within a monolithic application by leveraging both structural dependencies and runtime interactions. The study reveals that combining static and dynamic analysis provides a comprehensive view of the system, aiding in the identification of potential microservices by analyzing code dependencies, communication patterns, and runtime behavior. Despite its effectiveness, the approach is complex and resource-intensive, generating a large volume of data that

can be challenging to process and interpret.

2.4 Gradual Migration

Although there are various approaches in the literature, only a limited number have employed the Strangler Fig Pattern as a primary strategy for incremental decomposition. This pattern involves the gradual replacement of parts of the monolithic system with microservices, providing a practical and less disruptive pathway for migration. In this section, we review studies that have leveraged this pattern, examining their methodologies and outcomes.

Volynsky *et al.* [29] introduced the Architect framework, which facilitates the migration process and enables the creation of a reliable architecture for distributed applications. They transitioned from a shared database pattern to a database-per-microservice pattern using an iterative approach rather than a big bang approach.

Li *et al.* [18] present a case study on the migration of a monolithic application to a microservice architecture using the Strangler Fig pattern. The main contribution of this paper is the detailed examination of the practical application of the Strangler Fig pattern, highlighting its benefits such as reduced risk, continuous delivery of new features, and improved maintainability through an incremental migration process. The study finds that the pattern allows for a smoother transition by isolating changes to specific parts of the system. However, it also identifies challenges like the initial complexity of setup, managing dependencies, and ensuring data synchronization. While their work utilized the Strangler Fig pattern for a specific case study with an emphasis on the migration process, this thesis extends the works of this paper by developing a tool for microservice identification that uses the Strangler Fig pattern,

facilitating the identification of microservices during the migration.

2.5 Single Source of Truth

In this work, we utilize data from various sources, both static and dynamic. Therefore, it is essential to ensure the consistency of the data before using it.

Müller *et al.* [21] introduces an open stack framework designed to create a unified data source for software analysis and visualization. The main contribution of this paper is the development of a comprehensive framework that integrates various tools and technologies to gather, store, and visualize software data from multiple sources. This framework aims to facilitate better understanding and management of software systems by providing a holistic view of the software's structure and behavior. In this thesis, we leverage the concept of a unified data store [21] to ensure the reliability and consistency of data before using it.

In this work, we integrate data from various sources, including both static and dynamic data analysis. The heterogeneity of these data sources presents challenges in ensuring their accurate and consistent use. To avoid inaccuracies and potential discrepancies in our results, it is crucial to establish a unified data store. This approach ensures that all data, regardless of its origin, is processed, stored, and analyzed in a coherent manner. By implementing this unified data store, we aim to enhance the reliability and consistency of our analysis and improve the overall accuracy of our findings.

Chapter 3

Technical Approach

3.1 Motivation

As monolithic systems grow, they face increasing challenges in scalability and maintainability, often necessitating a shift to a microservices architecture. However, identifying the appropriate services within a monolith is complex and time-consuming, particularly when migrating incrementally rather than using a big bang approach. The Strangler Fig Tree pattern, which supports gradual migration, offers a more manageable and less risky transition by progressively replacing monolithic components with microservices.

Existing approaches to microservice identification often rely on a single decomposition at the beginning of the process, which may not adapt well to ongoing changes in the monolith. In contrast, our framework iteratively reassesses and refines the decomposition before each migration step, addressing the limitations of current tools that do not fully accommodate the incremental nature of most real-world migrations.

This chapter will explore the technical details and challenges involved in implementing our approach.

3.2 Overview of the Approach

The ultimate goal of this framework is to assist users in identifying and extracting microservices from a monolithic architecture in a gradual and systematic manner. This process is structured to facilitate a smooth transition, allowing for the progressive migration of components while ensuring the system remains functional throughout. The approach can be summarized in the following steps.

3.2.1 Step 1: Initial Setup

The initial setup involves configuring the foundational infrastructure required to support the microservices architecture. This setup is essential for enabling the seamless operation and communication of services within the system. This phase emphasizes the importance of establishing core infrastructure elements that will vary depending on the specific needs of the system and the chosen architectural patterns. In this phase, users are encouraged to identify and implement the infrastructural components that best suit their architecture.

Below, we outline some of the most commonly used infrastructural elements in microservices architecture. While implementing these components is not mandatory, they offer several advantages in terms of efficiency, stability, and scalability. Users can select from these or incorporate other elements according to their specific requirements.

- **Service Registry**: Manages service discovery, allowing microservices to find and communicate with each other dynamically, thus decoupling the system components.
- **API Gateway**: Acts as a single entry point for all client requests, routing them to the appropriate microservices. It helps in load balancing, security, and monitoring, and simplifying the interaction between clients and services.
- **Configuration Server**: Centralizes configuration management, allowing microservices to retrieve their configuration settings from a central location. This ensures consistency and simplifies the management of configuration changes across the system.
- **Circuit Breaker**: Protects the system from cascading failures by monitoring and limiting the number of failed requests between microservices. If a service fails, the circuit breaker can stop further attempts to call it, allowing the system to maintain overall stability.
- **Load Balancer**: Distributes incoming network traffic across multiple microservices, ensuring no single service becomes overwhelmed. This helps in achieving high availability and reliability by efficiently managing resource utilization.
- **Message Broker**: Facilitates asynchronous communication between microservices by allowing them to exchange information through messages. This decouples services and improves system resilience by enabling non-blocking communication, often in an event-driven architecture.

3.2.2 Step 2: Decomposing the Monolith

The core function of our framework is to determine the decomposition of the monolithic project. This process can be divided into three distinct phases:

Analysis Phase

In this phase, static code analysis and dynamic analysis are conducted on the monolithic project to create a knowledge graph. This graph models the entire monolithic system, transforming the problem of microservice identification into a problem of finding cohesive and loosely coupled communities within the graph [19]. The knowledge graph serves as the foundational input for the subsequent phases.

Community Detection Phase

Using the knowledge graph, the Louvain algorithm [5] is applied to detect communities within the graph and one of its subgraphs. Although these communities may appear to be potential microservice candidates, they still lack some key features required for effective microservices, such as the appropriate granularity. Therefore, further refinement is necessary to ensure these communities meet the essential characteristics and constraints of microservices. This refinement process is addressed in the next phase.

Optimization Phase

The communities identified in the previous phase are further refined using an Integer Linear Programming (ILP) model, which addresses a variant of the Multiway Cut problem [4]. This model incorporates specific constraints to ensure the identified

communities meet the criteria for effective microservices. The output of this phase is a list of microservice candidates along with a few metrics associated with each of them. We will discuss further the algorithm in Section 3.3.

3.2.3 Step 3: Selecting the Next Microservice

The user must select one of the microservice candidates for implementation from the decomposition suggested in the previous step. Experts believe that complete automation of this process is not feasible [18]. While metrics such as cohesion and coupling provided in the previous step can aid in this decision, empirical evidence suggests that user knowledge of the system's bounded context often yields better results. Therefore, users are encouraged to make their selection based on both metric comparisons and their expertise.

3.2.4 Step 4: Implementing the Microservice

In this step, the selected microservice candidate is implemented. This involves extracting the microservice from the monolithic system and ensuring it interacts correctly with the remaining parts of the system. The goal is to maintain overall system functionality without disruption.

3.2.5 Step 5: Evaluating the Remaining Monolith

After implementing the microservice, the user assesses whether the remaining monolithic system is small enough to be considered a microservice itself. If it is, the migration process concludes with the remaining monolith becoming the final microservice. If not, the remaining system undergoes further decomposition. The process iterates

by treating the remaining monolith as the new system to be analyzed and decomposed, returning to Step 2.

3.3 Monolith Decomposition

The core of our framework is designed to identify and extract microservice candidates from a monolithic project using static and dynamic analysis, graph-based community detection, and optimization techniques. This section offers a detailed explanation of the decomposition process, which comprises three primary phases: Analysis, Graph Decomposition, and Optimization.[1]

3.3.1 Analysis

The goal of this phase is to represent the system as a knowledge graph. While previous works such as Filippone et al. [9] primarily use static code analysis (SCA) to construct the graph, our approach expands on this by integrating both static and dynamic data. Additionally, unlike Filippone et al.'s work, which does not account for the frequency of each relationship, our method incorporates the number of occurrences of a relationship into its final weight.

Static Code Analysis

In our system analysis, we leverage the common use of layered architectures in web service development, where methods in the presentation layer handle incoming requests and call methods in the lower layers down to the persistence layer and database [30].

[1] This section is largely based on the work of Filippone et al. [9]

We abstract these layers into two main layers: a "logic" layer, comprising both presentation and business layers, and a "persistence" layer. This abstraction allows us to create a graph representation that captures the relationships among components, considering the role of each layer in the system.

Our analyzer tool parses the code to classify the system's business logic and domain entity classes into these layers. It utilizes framework annotations (e.g., $@Controller, @Service, @Repository, @Entity$) to identify the technical role of each class and determine whether it represents a domain entity. This automated classification can be manually refined to enhance accuracy, providing a detailed and structured view of the system's architecture.

After allocating the classes into their respective layers, the tool inspects each class to identify the declared methods. It then recursively traverses the code's syntax tree to collect method calls and references to entities. For the graph representation, a node is created for (i) each method from classes in the logic and repository layers, and (ii) each entity class. Relationships between methods and entities are represented as directed arcs in the graph. We define five types of arcs, each representing a specific type of relationship:

- **Calls**: Arcs are added between method nodes when one method calls another method.
- **Uses**: Arcs are created between methods and entities if a method references an entity, such as instantiating an object or accessing its values.
- **Persists**: Arcs connect methods and entities if a method in the persistence layer reads from or writes to a database entity.

- **References**: Arcs link entities if one entity references another, representing association, aggregation, and composition relationships in the domain model.
- **Extends**: Arcs denote a generalization relationship if one entity extends another entity.

While this concept can be applied to layered architecture projects in any language, our tool currently only includes a static code analyzer for Java Spring framework projects. However, our framework provides an environment for developing static code analyzers for other languages. When developing a new static code analyzer for other languages, it is crucial to account for the differences in annotations across various programming languages and frameworks.

Dynamic Analysis

Dynamic data is obtained through dynamic analysis, which involves monitoring and profiling the system's runtime behavior. This analysis provides insights into how the system operates in real-time. Monitoring and profiling tools are used to observe and record the method call stacks during the system's execution. Dynamic data used in this project is essentially a call tree. However, different tools may represent the call tree in various formats and structures, generally, call trees are represented by a list of methods as nodes and a list of pairwise relationships between these nodes.

To collect dynamic data, the profiling tool must be integrated with the running project. After integration, we execute a series of predefined scenarios on the project, ensuring these scenarios capture all the functionalities provided by the tool. This approach ensures comprehensive coverage of the system. Once profiling is complete,

the output is stored as either a CSV or XML file. [2]

After extracting the dynamic data, we process it to generate a list of methods and their call relationships, referred to as **DCalls**.

Building the Knowledge Graph

At this stage, we have a list of method and entity nodes, as well as a list of arcs between these nodes. These arcs can be categorized into six different types, each with a specific weight coefficient used for calculating their final weight. Table 3.1 shows the coefficient for each arc type, sorted in ascending order by their value. Similar to the system layers, these coefficients can be manually adjusted to meet specific application domains and requirements.

Arc Type	**Weight Coefficient**
Extends	0.0
References	0.2
Uses	0.6
Calls	0.8
DCalls	0.9
Persists	1.0

Table 3.1: Weight coefficients for each arc type.

Although the coefficients provided in Table 3.1 are set as default values for each arc type, they are not rigid and can be refined by users according to their specific needs. These values, derived from experimental results, may not represent the absolute best possible values but have demonstrated good practical outcomes across different projects. The weights assigned to the arcs indicate the significance of the

[2] As discussed in Chapter 4, we use two different profiling tools to store the results of dynamic analysis on two separate projects. Further details on this process can be found in Section 3.4.

relationships between nodes: the higher the weight of an arc, the stronger the likelihood that the methods and/or entities at its endpoints should be grouped within the same microservice.

The order of these values is partially based on the effectiveness of each arc type in grouping nodes into a microservice. For instance, the **Persists** arc type is given the highest weight of 1, which is logical because a **Persists** arc represents a relationship between a method and an entity in the database. Since the method has direct access to the entity, they are more likely to reside within the same microservice, justifying the high weight assigned to **Persists**. Similarly, **DCalls** has a higher weight than **Calls** because runtime method calls (captured by DCalls) are more indicative of the true relationships between methods. For example, a **Calls** relationship identified through static code analysis might never actually occur during runtime, making it less reliable than a **DCalls** relationship that actually happens in runtime. The **Extends** arc has a coefficient of 0, meaning it does not influence whether two entities should be grouped into the same microservice.

As mentioned before, there is another input that affects the final weight of each arc: the frequency of occurrence of each relationship. Let's explain this with an example:

```
public void A(){
    B();
    C();
    B();
}
```

In this example, method `A()` calls method `B()` twice and method `C()` only once.

Therefore, the frequency of the **Calls** relationship between `A` and `B` is 2, and between `A` and `C` is 1. We count the number of occurrences of each type of relationship and use it as a parameter for calculating the final weight of the graph.

The output of this phase is a directed graph $G = (V, A)$. Each node $i \in V$ is assigned a type $t_i \in \{Method,\ Entity\}$, and each arc $(i, j) \in A$ is assigned a relationship type $r_{ij} \in \{DCalls,\ Calls,\ Uses,\ Persists,\ References,\ Extends\}$. Additionally, each arc (i, j) is assigned a weight w_{ij}.

To calculate the final weight of the edges, we use Equation (3.3.1), where frequency_{ij} represents the number of occurrences of an arc.

$$w_{ij} = \text{coefficient}(\text{r}_{ij}) \times \left(1 + \frac{\text{frequency}_{ij} - 1}{10}\right) \tag{3.3.1}$$

Regarding Formula (3.3.1), the intention was to account for the importance of relationships that appear multiple times in our graph without exaggerating their significance. For example, a call that occurs twice does not necessarily imply that it is twice as important as a call that happens only once.

The graph built in this step forms the foundation for our subsequent computations. While Filippone et al. [9] construct their knowledge graph instantly in memory, making it easily accessible for the next steps, this approach is feasible primarily because they rely solely on static data. However, our approach incorporates both static and dynamic data, introducing new challenges that must be addressed.

To tackle these challenges, we make use of the Single Source of Truth (SST) paradigm. In Section 3.4, we will discuss these challenges in detail, along with the technical aspects of the SST. By building our knowledge graph on the SST, we ensure that we can efficiently fetch the required parts of the graph for the next steps of our

process.

3.3.2 Graph Decomposition

As previously mentioned, when representing the system as a graph, the challenge of identifying microservices transforms into the task of finding loosely coupled and highly cohesive communities within the graph [19]. Consequently, accurately identifying these communities is vital for determining high-quality microservice candidates.

Several community detection algorithms exist, including the Girvan-Newman (GN) algorithm [12], [22], the Kernighan-Lin (KL) algorithm [17], the Spectral Bisection algorithm [2], and the Louvain algorithm [5]. Among these, the Louvain algorithm stands out due to its superior features, such as the lowest time complexity, higher cohesion, and stability [19]. Stability is crucial as it ensures the algorithm does not produce entirely new communities with each run.

The Louvain algorithm detects communities by maximizing a modularity score, which measures how densely connected the nodes are within a single community while maintaining fewer connections between different communities [23]. Equations (3.3.2) and (3.3.3) define the modularity heuristic function [23] that the Louvain algorithm optimizes to evaluate the partitioning of a community:

$$Q = \frac{1}{2m} \sum_{(i,j)} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j), \tag{3.3.2}$$

$$\delta(u, v) = \begin{cases} 1 & \text{if } u = v \\ 0 & \text{if } u \neq v \end{cases} \tag{3.3.3}$$

where A_{ij} represents the weight on the edge connecting nodes i and j. m is the sum of the weights of all edges in the graph, and c_i is the community to which node i currently belongs. The term k_i represents the sum of the weights of all edges connected to node i:

$$k_i = \sum_j A_{ij}, \tag{3.3.4}$$

Being unsupervised, the Louvain algorithm does not require the number of communities to be predefined or the size of the communities [6]. In this work, we use the Python implementation of the Louvain algorithm provided by the **communities** Python package[3]. In this implementation, the Louvain method includes a property called *resolution*. The *resolution* parameter influences the granularity of the detected communities: higher values of **resolution** lead to the identification of more, smaller communities, whereas lower values result in fewer, larger communities. The default value for **resolution** is set to 1. Users have the option to adjust this variable if they believe it will yield better results.

Through my experiments with the Louvain algorithm, I observed that increasing the resolution parameter from 1 to 1.5 generally resulted in the detection of one additional entity cluster, and consequently, one more microservice. However, working with higher resolution values also introduced the risk of producing microservices that were too small to be practical. Conversely, using lower resolution values tended to result in fewer, but larger, microservice candidates. After testing a range of resolution settings, I decided to keep the default resolution value at 1 for our approach. Given that our method is iterative, with users implementing one microservice per iteration,

[3] `https://pypi.org/project/communities/`

setting the resolution to 1 strikes a balance between granularity and manageability. Additionally, I provided users with the flexibility to adjust the resolution parameter according to their specific needs, allowing them to fine-tune the results based on the context of their project.

By applying the Louvain algorithm to our system representation, we obtain a cohesive set of communities comprising nodes from all layers of the system. While this approach is beneficial as it considers relationships between nodes with different roles (i.e., entities, controller interfaces, business logic) which is required to offer complete functionalities, these communities cannot generally be considered acceptable microservice candidates. The absence of structural constraints in the candidate microservices leads to the formation of communities that may consist solely of entity nodes or solely of method nodes, thus failing to form complete and standalone microservices. Additionally, there is no assurance regarding the appropriate granularity of these communities. Without constraints on the number of microservices to be identified, they are likely to be too fine-grained, especially when derived from graphs representing large systems with a high number of methods. Consequently, these communities may not include all the nodes necessary to realize a business functionality.

To address these issues, we applied the Louvain community detection algorithm to the information graph in two different ways: (i) on the complete information graph, including all architectural layers, to obtain cohesive sets of nodes; and (ii) on a subgraph containing only the entities related to the persistence layer, which plays a crucial role in maintaining the domain context [7].

Graph Communities from the Complete Graph The execution of the Louvain algorithm on the complete information graph yields a set C of n communities

$C_1, C_2, \ldots, C_n$, which include nodes representing entities and methods across all architectural layers. These communities are not sufficient to be considered complete microservices due to their fine granularity. For instance, methods related to a single entity might be scattered across multiple communities. This suggests that each community may lack the comprehensive set of functionalities necessary to fulfill a business capability, thereby failing to define a bounded context. This may result in the communities identified in this phase being highly coupled, despite their cohesiveness. The inherent cohesiveness of these communities can be leveraged to improve cohesion in the final microservice architecture.

Entity Clusters We derive the subgraph of domain entities by considering the nodes of type *Entity* that have relationships with the persistence layer and the arcs of type *References* and *Extends*. Applying the Louvain algorithm to this graph yields a set K of m communities $K_1, K_2, \ldots, K_m$ of Entity nodes. For clarity, we will refer to the communities obtained from the entire graph as **communities**, and those obtained from the domain entities subgraph as entity **clusters**.

In the optimization phase, we use both communities and entity clusters to achieve cohesion and determine the appropriate granularity of the microservices.

3.3.3 Optimization

To optimize the microservice architecture, we employ an Integer Linear Programming (ILP) model that addresses a variant of the Multiway Cut problem as formulated in [3]. We have introduced specific constraints to define the structure of the microservices to be identified. The primary goal of this optimization is to partition the nodes of the information graph into m microservices, where m represents the number of entity

clusters determined in the previous step. The objective is to minimize the coupling between microservices. Coupling is quantified as the sum of the weights of the arcs that connect different microservices.

We formalize the problem as follows:

Given a graph $G = (V, E)$, with nodes V and edges E, and a type $t_i \in \{\textit{Method, Entity}\}$ for each node $i \in V$, and an associated weight w_{ij} for each edge $(i, j) \in E$, the goal is to partition V into m sets $\{M_1, M_2, \ldots, M_m\} = M$ such that: (i) $\bigcap M_i = \emptyset$;

(ii) $\bigcup M_i = V$;

(iii) Each $M_k \in M$ contains at least one node i where $t_i \neq$ Entity;

(iv) The sum of weights w_{ij}, where $i \in M_h$ and $j \in M_k$ with $h \neq k$, is minimized.

Each set $M_1, M_2, \ldots, M_m$ represents a potential microservice. The problem is modeled using an ILP formulation with the following decision variables, as described in [3]:

$$x_{ik} = \begin{cases} 1 & \text{if node } i \text{ is in microservice } M_k \\ 0 & \text{otherwise} \end{cases} \tag{3.3.5}$$

$$y_{ij} = \begin{cases} 1 & \text{if edge } (i, j) \text{ has its endpoints within the same microservice} \\ 0 & \text{otherwise} \end{cases} \tag{3.3.6}$$

$$z_{ij}^k = \begin{cases} 1 & \text{if edge } (i,j) \text{ has both its endpoints in microservice } M_k \\ 0 & \text{otherwise} \end{cases} \tag{3.3.7}$$

We define the following constraints:

$$z_{ij}^k - x_{ik} \leq 0 \quad \forall M_k \in M,\, \forall (i,j) \in E \tag{3.3.8}$$

$$z_{ij}^k - x_{jk} \leq 0 \quad \forall M_k \in M,\, \forall (i,j) \in E \tag{3.3.9}$$

$$x_{ik} + x_{jk} - z_{ij}^k \leq 1 \quad \forall M_k \in M,\, \forall (i,j) \in E \tag{3.3.10}$$

$$y_{ij} = \sum_{M_k \in M} z_{ij}^k \quad \forall (i,j) \in E \tag{3.3.11}$$

$$\sum_k x_{ik} = 1 \quad \forall i \in V \tag{3.3.12}$$

$$\sum_{i \in V | t_i = \text{Method}} x_{ik} \geq 1 \quad \forall M_k \in M \tag{3.3.13}$$

Constraints (3.3.8) to (3.3.12) originate from the Multiway Cut problem formulation, while constraint (3.3.13) is newly added to ensure methods are included in the microservices, as they are crucial for exposing interfaces, offering functionalities, and implementing microservice logic.

The objective function (3.3.14) aims to minimize coupling among microservices, defined as the sum of the weights of arcs connecting different microservices:

$$\min \sum_{(i,j)\in E} w_{ij}(1 - y_{ij}) \tag{3.3.14}$$

However, the solution to this optimization model may produce microservices with low coupling that exhibit several issues. For example, domain entities might be improperly assigned to a single all-containing microservice, while others might consist of a few loosely related nodes, compromising both architectural integrity and functional completeness.

To achieve domain-driven decomposition and build microservices within bounded contexts, we assign entities to microservices based on the entity clusters identified earlier. Thus, for each entity i in the entity cluster $K_k \in K$, i must belong to microservice k. This is enforced by the following variables:

$$x_{ik} = 1 \quad \forall i \in V \text{ if } i \text{ is in entity cluster } K_k \tag{3.3.15}$$

To ensure cohesion, we group all nodes from the same community into the same microservice, resulting in highly cohesive microservices. Thus, we fix the y variables as follows:

$$y_{ij} = 1 \quad \forall (i,j) \in E \text{ if } i \text{ and } j \text{ are in the same community} \tag{3.3.16}$$

This approach may lead to an infeasible model if nodes from the same community belong to different entity clusters. To address this, we use a Fix-and-Relax strategy. First, we fix the x variables based on entity clusters. Then, for each community

$C_c \in C$, we fix the y variables and check model feasibility. If infeasible, we relax the fixed y variables for nodes in the community C_c to restore feasibility.

Algorithm 1 Fix-and-Relax algorithm for x and y variables

```
for entity cluster K_k ∈ K do
  for i ∈ k do
    Fix x_ik = 1
  end for
end for
for community C_c ∈ C do
  for (i, j) ∈ E do
    if i ∈ C_c and j ∈ C_c then
      Fix y_ij = 1
    end if
  end for
  if model is infeasible then
    for (i, j) ∈ E do
      if i ∈ C_c and j ∈ C_c then
        Relax y_ij
      end if
    end for
  end if
end for
```

The results of this optimization allow the assignment of nodes in the information graph (i.e., methods and entities) to microservices, thereby defining the architecture

with the necessary domain-driven design principles.

The output of this algorithm is a list of microservice candidates and metrics assessing their cohesion and coupling which will be used in the next step as described in Section 3.2.3.

I would like to highlight two key differences between our approach and that of Filippone et al. First, their work offers a tool that decomposes the entire system in a single iteration, resulting in a collection of potential microservice candidates. In contrast, our approach provides a comprehensive framework that facilitates the incremental migration process. It offers a clear roadmap for users and suggests potential microservice candidates at each step of the migration.

The second technical difference lies in how the information graph is constructed. Our approach extends the work of Filippone et al. by incorporating dynamic data into the graph and defining a weight criterion for each edge based on their frequency of occurrence. The algorithms used for community detection and optimization in both approaches are largely similar.

3.4 Single Source of Truth

Incorporating dynamic data introduces the challenge of ensuring data consistency. Static data, derived from static code analysis, can be generated using various tools and packages. For instance, in this work, we use the *javalang* Python library[4] to parse the source code. Conversely, dynamic data results from the project's runtime behavior, gathered using different profiling tools. For our project, we used VisualVM[5]

[4]https://github.com/c2nes/javalang
[5]https://visualvm.github.io/

and JProfiler[6] for the two different projects that we have tested in Chapter 4.

For a specific project under study, although the static code analyzer and dynamic data analyzer are analyzing the same project, their outputs may not be consistent. Let's consider an example. The raw output of the static code analysis for the Spring Petclinic project includes two CSV files: one with entries representing nodes and another representing edges between these nodes. Figures 3.1 and 3.2 display a portion of these files. We use VisualVM to profile the Spring Petclinic project, with Figure 3.3 illustrating the data extracted from VisualVM. The node numbered 13 in Figure 3.1 corresponds to the same method as the second entry in Figure 3.3. When creating the graph, it is crucial to ensure that we do not create two distinct nodes for these data points. Creating separate nodes for data that essentially represent the same node can lead to contaminated data and consequently, inaccurate results.

```
id, type, name
1,Entity,Vet
2,Entity,Specialty
3,Entity,Vets
4,Entity,Pet
5,Entity,Owner
6,Entity,Visit
7,Entity,PetType
8,Entity,BaseEntity
9,Entity,Person
10,Entity,NamedEntity
11,Method,processNewVisitForm,VisitController
12,Method,initCreationForm,PetController
13,Method,findOwner,OwnerController
```

Figure 3.1: Nodes from Static Code Analysis

These data sources have different structures. Despite these differences, they share

[6]https://www.ej-technologies.com/

```
from, type, to, count
20,Calls,38,1
20,Calls,13,1
38,Calls,22,1
45,Calls,41,1
11,Calls,37,1
44,Calls,41,1
43,Calls,37,1
32,Calls,25,1
```

Figure 3.2: Edges from Static Code Analysis

```
"Name","Total Time","Total Time (CPU)","Invocations"
" org.springframework.samples.petclinic.model.NamedEntity.toString ()","0.005 ms (-0%)","0.004 ms (-0%)","1"
" org.springframework.samples.petclinic.owner.OwnerController.findOwner (Integer)","5.18 ms (-0%)","4.34 ms (-0%)","2"
```

Figure 3.3: Data from Dynamic Analysis using VisualVM

a commonality: both represent the same components. This similarity forms the foundation for combining these data sources. In our proposed framework, the result of static analysis establishes the backbone of the knowledge graph. Dynamic data, collected at various times, can be incrementally added to the graph to provide a more accurate representation of the system.

Furthermore, our methodology operates incrementally, resulting in working with different versions of the software over a period of time. The migration process here is not merely about decomposing the graph into a few subgraphs. Instead, our approach involves a systematic process. If we build the graph in memory, we would need to recreate it each time we need it for further computations. Since the migration process may take several months to complete, it is not computationally efficient to recompute the graph each time it is needed. Therefore, it is essential to store the information graph in non-volatile memory, allowing instant access without the need for heavy computations. Additionally, as mentioned in the previous paragraph, having the

graph in non-volatile memory enables us to add new information to the existing graph, thereby updating and enriching it.

To ensure the data is reliable and consistent, we leverage the **Single Source of Truth** (SST) paradigm. SST emphasizes maintaining a single, authoritative source for all data within a system. By centralizing data storage and access, SST ensures data consistency and reliability. With SST, all data for all versions are stored in one location, eliminating the need to process the data to build the graph each time. We can simply fetch the required parts of the system, reducing memory usage by avoiding unnecessary data retrieval. Additionally, since the migration process in a large project may take several months, SST guarantees the availability of our data throughout this process.

By implementing SST, we not only streamline data management but also enhance the efficiency and accuracy of our migration process. This centralized approach allows for seamless updates and integration of new data into the existing graph, ensuring that our system remains up-to-date and reflective of the current state of the software. As a result, we can maintain a coherent and comprehensive view of the system, facilitating better decision-making and more effective project management.

3.4.1 Technical Aspects of SST

This section explains how to use SST from a user perspective. The core of SST is a Neo4j[7] database wrapped by a Node.js Express server. As users of SST, we do not delve deeply into its implementation. Instead, we focus on the two endpoints provided for building our graph through these APIs. SST is a general-purpose paradigm. In

[7]https://neo4j.com/

our work, we define two small programs called **Probes** to prepare and transform the raw output of static code analysis and dynamic data analysis into a format that SST can accept as a standard input.

We define a **Static Code Analyzer** probe, responsible for transforming static code analysis output into a JSON file compatible with SST. The other probe developed in this work is the **Dynamic Data Analyzer** probe, which processes and transforms profiling tool data into the standard format accepted by SST.

The standard format for SST's APIs is a JSON file containing two properties: a list of nodes and a list of edges. The node types that we are going to add to SST need to be defined before inputting them into SST. We use the **/api/types** API endpoint to define our node types. This endpoint is a REST POST request that accepts a JSON file in its body. In this file, we define two node types: Entity and Method. For each node type, we define a set of properties and a *MergeRule*, which specifies how to merge two nodes if they refer to the same component.

After defining the node types, we run the Static Code Analyzer probe to generate a JSON file containing a list of nodes and edges. Using another REST POST request, we send this file to the **/api/upload-graph** endpoint to input it into SST. In the next step, after extracting the dynamic data using a profiling tool, we use the Dynamic Data Analyzer probe to generate a new JSON file with the same structure. This JSON file contains a list of nodes and edges derived from dynamic analysis. Finally, this data is input into SST to build our comprehensive knowledge graph. At the end, we fetch the graph from the SST server for further computations.

Chapter 4

Evaluation

In this chapter, we aim to evaluate the effectiveness, accuracy, and correctness of our approach by comparing it to other state-of-the-art approaches. Our evaluation focuses on the following criteria:

1. **Independence of Functionality:** This criteria follows the *Single Responsibility Principle* (SRP), which asserts that a service should offer well-defined, independent, and coherent functionalities to its external clients. We assess this criterion by calculating the *IFN* (Interface Number) metric. Independence of functionality is crucial as it ensures that each microservice is focused on a specific business capability, making the system easier to maintain and evolve.

2. **Modularity:** A well-modularized system should have high cohesion within its components and low coupling between them. We evaluate modularity using two metrics: *average cohesion* and *average coupling*. High cohesion indicates that the internal entities of a service are closely related and work well together, while low coupling suggests that entities across service boundaries are minimally

dependent on each other. Modularity is essential for reducing complexity and enhancing the maintainability and scalability of the system.

These criteria were selected because they directly relate to the fundamental principles of microservice architecture, which emphasize creating services that are independent, modular, and loosely coupled. Evaluating our approach based on these criteria allows us to determine its effectiveness in achieving these core architectural goals. Additionally, these criteria are commonly used in similar works in the literature [6, 9, 13, 15, 16, 30], providing a standardized basis for comparison and validation of our approach.

4.1 Evaluation Metrics

IFN Metric Regarding the independence of functionality, we use the *IFN* metric, which measures the average number of interfaces exposed by a microservice. It is defined as follows:

$$IFN = \frac{1}{|M|} \sum_{M_k \in M} ifn_k, \tag{4.1.1}$$

where $|M|$ is the set of identified microservices and ifn_k is the number of interfaces exposed by the microservice k. According to our system representation, we define an interface as a set of methods of the logic layer, without incoming edges, that are connected to the same set of entity nodes. A microservice focused on the single responsibility principle is expected to have only one interface, i.e., to offer functionalities related to a single entity. Hence, the lower the value of *IFN* (down to 1), the more likely the microservice architecture is to follow the single responsibility principle.

Average Cohesion *Average cohesion* shows how cohesive the whole set of identified microservices are. In the graph representation of the system, *cohesion* for each microservice is defined as the ratio between the sum of the weights of inner arcs (i.e., those arcs with endpoints in the same microservice) and the sum of the weights of all arcs outgoing from the nodes of a microservice (essentially all the arcs that have at least one endpoint in a microservice). *Average cohesion* is the average of *cohesion* over all identified microservices:

$$AverageCohesion = \frac{1}{|M|} \sum_{M_k \in M} \frac{inner_k}{outer_k}, \tag{4.1.2}$$

where M is the set of identified microservices. The values of $inner_k$ and $outer_k$ are obtained as follows:

$$inner_k = \sum_{(i,j) \in E | i,j \in M_k} w_{ij}, \tag{4.1.3}$$

$$outer_k = \sum_{(i,j) \in E | i \in M_k} w_{ij}. \tag{4.1.4}$$

The ideal value for average cohesion in a microservice architecture is 1, indicating that the elements within each microservice are highly interrelated. A high cohesion value signifies that all methods within a microservice have strong relationships with the entities included in the same microservice. This implies that a microservice with high cohesion is likely well-aligned with a specific bounded context. Conversely, a poor microservice decomposition may exhibit low cohesion, as many methods may need to reference entities located in other microservices.

Average Coupling *Average Coupling* measures the degree of coupling between the identified microservices. In the graph representation, it is calculated as the average weight of the arcs whose endpoint nodes belong to different microservices.

$$AverageCoupling = \frac{1}{|M|} \sum_{(i,j) \in E | i \in M_h, j \in M_k, h \neq k} w_{ij}, \tag{4.1.5}$$

where M is the set of identified microservices, $M_h, M_k \in M$, and w_{ij} is the weight of the arc (i, j). The lower the average coupling, the less coupled a microservice architecture is.

4.2 Performed Experiments

To evaluate our approach, we selected two publicly available projects implemented in the Java language[1]: the Spring Petclinic Project[2] and MyBatis JPetStore[3]. These monolithic projects have previously been used in the evaluation of related works on microservice decomposition. They are suitable choices for our study as they allow for direct comparison with other state-of-the-art approaches. Their moderate size and representative features make them ideal for demonstrating and testing decomposition methodologies. Table 4.1 shows the size of each tested project.

Project	Number of Classes	Lines of Code	Number of Entities
Petclinic	24	805	10
JPetStore	25	1475	9

Table 4.1: Comparison of Projects

The Spring Petclinic Project is a sample application designed to demonstrate

[1]https://www.java.com/
[2]Obtained in March 2024 from https://github.com/spring-projects/spring-petclinic
[3]Obtained in April 2024 from https://github.com/mybatis/jpetstore-6

the use of the Spring framework[4] in building a simple web-based clinic for managing veterinarians, pets, and their owners. It showcases best practices for building a layered architecture with Spring and includes functionalities such as adding, editing, and viewing information about veterinarians, pet owners, and their pets. This project is commonly used in related works due to its clear structure and comprehensive feature set, which are representative of real-world applications.

The MyBatis JPetStore Project is an open-source application that demonstrates the use of MyBatis[5], an object-relational mapping framework, in a web-based pet store. The project includes features such as product browsing, order processing, and customer management, providing a comprehensive example of a monolithic application with a well-defined domain model. Its moderate complexity and variety of functionalities make it a valuable benchmark for testing decomposition approaches.

In the following sections, we will evaluate our approach on these two projects. Unlike other decomposition approaches, which can be assessed after a single iteration, evaluating our approach is more complex. Other methods can be evaluated based on the result of a single migration iteration. However, our approach requires the entire migration process to be completed before the final decomposition is achieved. Therefore, we need to use the framework throughout the entire migration process to obtain a complete result for comparison with other approaches. In real-world scenarios, a team of experts typically oversees the migration process, and the extent of their impact on the final outcome is closely tied to the depth of their knowledge about the system and its context. For this work, since the projects are used solely for testing purposes and lack real-world application, I will assume the role of the expert

[4]https://spring.io/
[5]https://blog.mybatis.org/

during the migration process in the following experiments. For both experiments, the process and each decision made are documented. After the final iteration, the results will be compared with those of other approaches.

All the artifacts and source code of our project are accessible from:
https://github.com/ace-papers/hassan-masc

4.3 Experiment 1: Spring Petclinic

In this section, we elaborate on the migration process for the **Petclinic** project. Given that our approach performs migration iteratively, each iteration is detailed in its own subsection. At the end, we compare our approach with other approaches.

4.3.1 Initial Setup

In the initial setup phase (iteration 0), we do not start decomposing the monolithic project yet. Instead, we focus on tasks that are essential and beneficial for the entire process.

Static Data The initial step is to create the graph in our database. To achieve this, we run the *Static Code Analyzer* probe on the project to obtain the static data as a JSON file. This data is then added to the SST to construct the knowledge graph.

Dynamic Data Next, we use the VisualVM tool to capture the dynamic call stack of the Petclinic project. By setting the tool to profiling mode and running a comprehensive set of scenarios, we aim to achieve high code coverage. After executing all scenarios, we take a snapshot of the *forward calls* as a CSV file. Then, we use the

Dynamic Data Analyzer probe to process this data into a JSON file, which is added to the SST to enrich the existing knowledge graph.

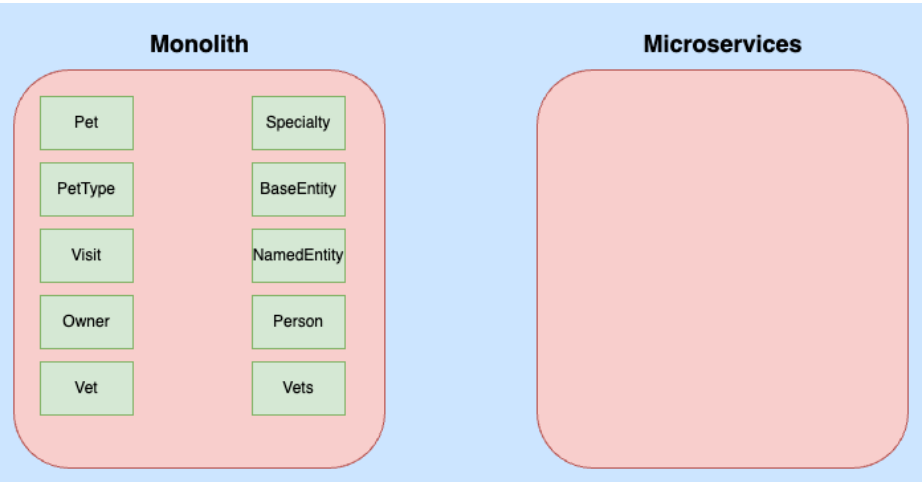

Figure 4.1: Monolith and Microservices projects before iteration 0.

Figure 4.1 illustrates the state of the monolithic and microservices projects before iteration 0. The monolithic project comprises 10 entities, three of which are superclasses inherited by other classes and do not exist as separate tables in the database. These three entities are: *BaseEntity*, *NamedEntity*, and *Person*. We will observe the progression of the monolithic and microservices projects over time, following each iteration.

Initial Setup In this step, we are ready to start the migration using our framework. Before proceeding, we implement a **configuration server** and a **service registry server** as the first two microservices of the project, as described in Section 3.2.1. The service registry is responsible for managing service discovery, which allows microservices to dynamically find and communicate with each other, effectively decoupling

the system components. The configuration server centralizes configuration management, enabling microservices to retrieve their configuration settings from a central location. This ensures consistency and simplifies the management of configuration changes across the system.

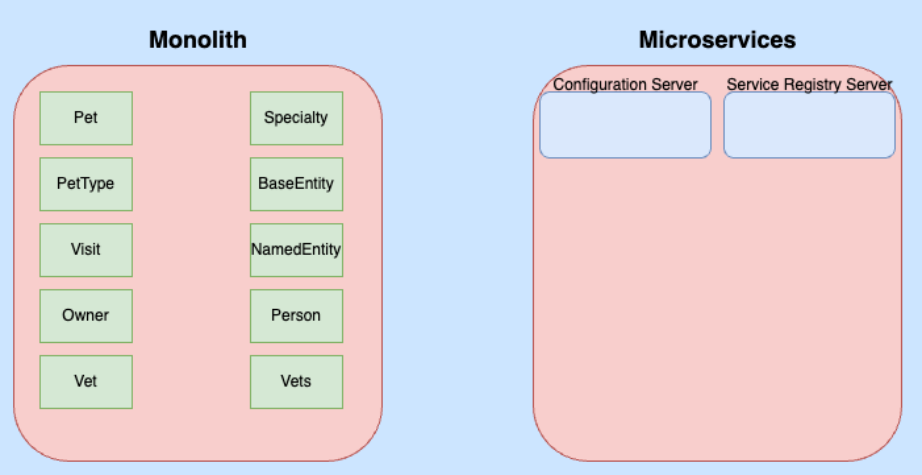

Figure 4.2: Monolith and Microservice projects after iteration 0.

4.3.2 Iteration 1

At this stage, we run the tool on the monolithic project. Figure 4.3 shows the output of the tool for the first iteration. As illustrated, the tool suggests a decomposition consisting of two microservice candidates. According to the metrics, the two candidates are equal. Both have perfect cohesion values. The coupling is 0, indicating no edges between these two clusters. Based on our knowledge of the system, we determine that *Candidate 0* is the best option to be implemented as a microservice first. Consequently, a new microservice named **Vet Service** is implemented. Figure 4.4 shows the architecture of the monolithic and microservice projects after iteration 1.

```
Found 2 candidates
Total coupling value: 0.0               Average Coupling 0.0
Cohesion value: 1.0

Entities in Candidate 0:  Cohesion: 1.0
-Vet
-Specialty
-Vets
Entities in Candidate 1:  Cohesion: 1.0
-Pet
-Owner
-Visit
-PetType
-BaseEntity
-Person
-NamedEntity

# of method calls crossing microservice candidates: 0
# of entity references crossing microservice candidates: 0
# of entity usages crossing microservice candidates: 0
# of methods persisting entities of other microservice candidates: 0
```

Figure 4.3: Output of the tool for the main petclinic project. Total Coupling is the sum of the weights that have endpoints in two candidates. The sum of the four numbers in the last four lines shows the unweighted coupling.

4.3.3 Iteration 2

Filippone et al. [9] identify two microservices for the Petclinic project. In our approach, the decision of when to finish the migration process is left to the expert. At this stage, I chose to perform one more iteration to identify an additional microservice. When there is more than one, our tool finds multiple decompositions for the system, which is the case for the second iteration of this project. Figure 4.8 displays the output after running the tool on the remaining part of the Petclinic project. Based on the coupling and cohesion values, Decomposition 2 in Figure 4.6 appears to be the best choice among the three. However, this decomposition suggests implementing **PetType** as the entity in the new microservice. Based on my knowledge of the system, this is not a good choice since **Pet** and **PetType** are conceptually related, and separating them into two different microservices does not seem rational. Based on

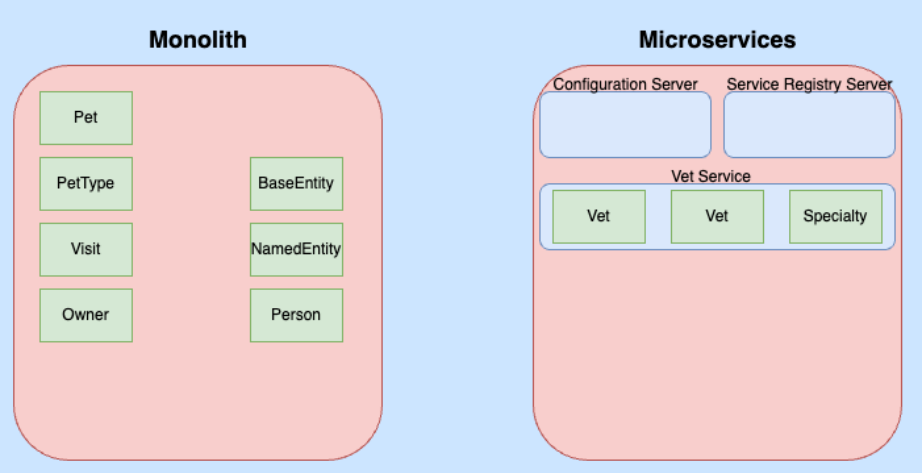

Figure 4.4: Monolith and Microservice projects after iteration 1.

my knowledge of the system, I decided to go with Decomposition 3. I implemented the new microservice with **Visit** as the only entity in it. Figure 4.9 shows the system architecture after implementing this new microservice.

```
Found 2 candidates
Total coupling value: 12.919            Average Coupling 6.4595
Cohesion value: 0.926

Entities in Candidate 0:  Cohesion: 0.862
-Pet
-Visit
-PetType
Entities in Candidate 1:  Cohesion: 0.99
-Owner
-BaseEntity
-Person
-NamedEntity

# of method calls crossing microservice candidates: 1
# of entity references crossing microservice candidates: 1
# of entity usages crossing microservice candidates: 8
# of methods persisting entities of other microservice candidates: 1
```

Figure 4.5: Decomposition 1

```
Found 2 candidates
Total coupling value: 0.38              Average Coupling 0.19
Cohesion value: 0.998

Entities in Candidate 0:  Cohesion: 0.997
-Pet
-Visit
-Owner
Entities in Candidate 1:  Cohesion: 1.0
-PetType
-BaseEntity
-Person
-NamedEntity

# of method calls crossing microservice candidates: 0
# of entity references crossing microservice candidates: 0
# of entity usages crossing microservice candidates: 1
# of methods persisting entities of other microservice candidates: 0
```

Figure 4.6: Decomposition 2

```
Found 2 candidates
Total coupling value: 21.78             Average Coupling 10.89
Cohesion value: 0.837

Entities in Candidate 0:  Cohesion: 0.803
-Pet
-PetType
-Owner
Entities in Candidate 1:  Cohesion: 0.871
-Visit
-BaseEntity
-Person
-NamedEntity

# of method calls crossing microservice candidates: 2
# of entity references crossing microservice candidates: 1
# of entity usages crossing microservice candidates: 3
# of methods persisting entities of other microservice candidates: 0
```

Figure 4.7: Decomposition 3

Figure 4.8: Output of the tool for the remaining part of petclinic project.

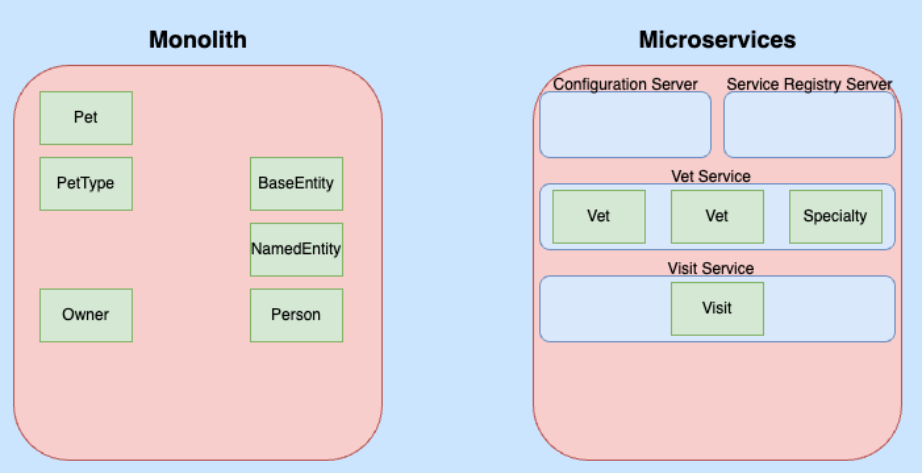

Figure 4.9: Monolith and Microservice projects after implementing the new microservice in iteration 2.

Before concluding iteration 2, we must decide whether to end the process. I believe the remaining part of the monolithic project is small enough to be considered a new microservice itself. Therefore, I refactor the remaining monolith project to be the last microservice defined in this process. As shown in Figure 4.10, there is nothing left in the monolithic project at the end of the migration process. We now have three main microservices that perform the core functionalities of the system. We use these three microservices as the final decomposition of the system and compare the results with those obtained from other approaches. Additionally, we have two other microservices: *Service Registry* and *Configuration Server*, which are not considered in the final evaluation of our approach. Notably, the **BaseEntity**, **NamedEntity**, and **Person** classes do not appear in Figure 4.10. This is because they were deleted during the refactoring process, though their functionalities are preserved.

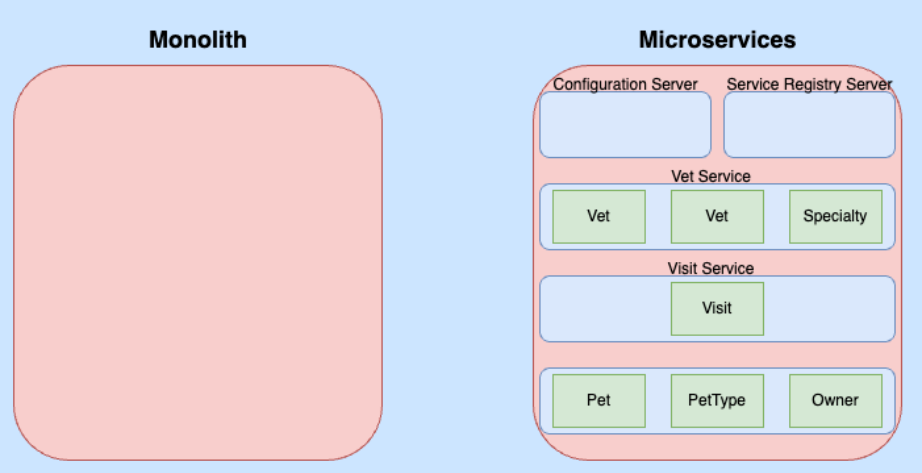

Figure 4.10: Monolith and Microservice projects at the end of migration process.

4.3.4 Final Result

Table 4.2 presents the comparison of the results obtained from our approach with other approaches for the Petclinic project. Filippone et al.'s approach serves as the baseline. For this project, there exists a gold standard, the **Spring Petclinic Microservices**[6], which was developed by the same team that created the **Monolithic Spring Petclinic** to demonstrate the process of splitting a sample Spring application into microservices. As shown in the table, the *Average Coupling* of the decomposition obtained by our approach is 2, which is higher than both the baseline and the gold standard. Our results would have matched the baseline if we had concluded the migration at the end of the first iteration. Notably, our approach achieves better *Average Cohesion* compared to the gold standard.

[6]https://github.com/spring-petclinic/spring-petclinic-microservices

Approach	# Microservices	Average Coupling	Average Cohesion	IFN
Kamimura et al. [16]	3	2.07	0.76	1.7
Baseline	2	0	1	2
Gold Truth	3	1.2	0.75	1.7
Our approach	3	2	0.89	1.7

Table 4.2: Result comparison for the Spring Petclinic project.

In this experiment, given the existence of a gold standard, we can compare our approach's decomposition with the decomposition of entities in the gold standard to calculate precision and recall. Precision in this context refers to the proportion of entities correctly assigned to a microservice out of all entities assigned by our approach to that microservice. Recall refers to the proportion of entities correctly identified for a microservice out of all entities that belong to that microservice in the gold standard. High values of precision and recall indicate that a given microservice owns the complete set of entities required to fully realize the bounded context, without including "spurious" entities that may pertain to different bounded contexts. Since the **Spring Petclinic Microservices** project has the same decomposition as ours, the precision and recall for each microservice are both equal to 1.

4.4 Experiment 2: MyBaits JPetStore

In this section, we detail the migration process for the **JPetStore** project. Similar to the previous experiment, we provide a comprehensive explanation of each iteration and the decisions made throughout the process. Finally, we compare our approach with other state-of-the-art methodologies.

4.4.1 Initial Setup

For this experiment, the initial setup closely follows the pattern established in the first experiment. We utilize the *Static Code Analyzer* and *Dynamic Data Analyzer* to construct the information graph in the SST. The only difference is that, in this project, we use the JProfiler tool to extract dynamic data instead of VisualVM. While VisualVM provides a CSV file, JProfiler generates an XML file. In both cases, we use the same *Dynamic Data Analyzer*, which processes the data and produces a JSON file containing a list of nodes and edges to be added to the graph. This JSON file is then uploaded to the SST to enrich the information graph.

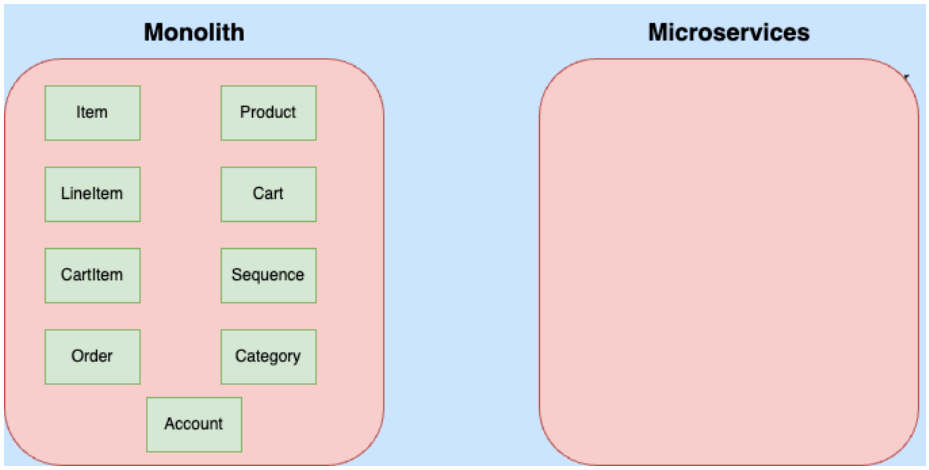

Figure 4.11: Monolith and Microservices projects before iteration 0.

Figure 4.11 depicts the state of the monolithic and microservices projects prior to iteration 0. The monolithic project consists of 9 entities: *Item, Product, LineItem, Sequence, Order, Category, Cart, CartItem, Account*. We will observe the progression of the monolithic and microservices projects over time, following each iteration.

In this experiment, we follow the same initial setup as described in the first experiment. This involves implementing a **configuration server** and a **service registry**

server as the first two microservices of the project, as outlined in Section 3.2.1. The service registry manages service discovery, allowing microservices to dynamically find and communicate with each other, thus decoupling the system components. The configuration server centralizes configuration management, ensuring consistency and simplifying the management of configuration changes across the system.

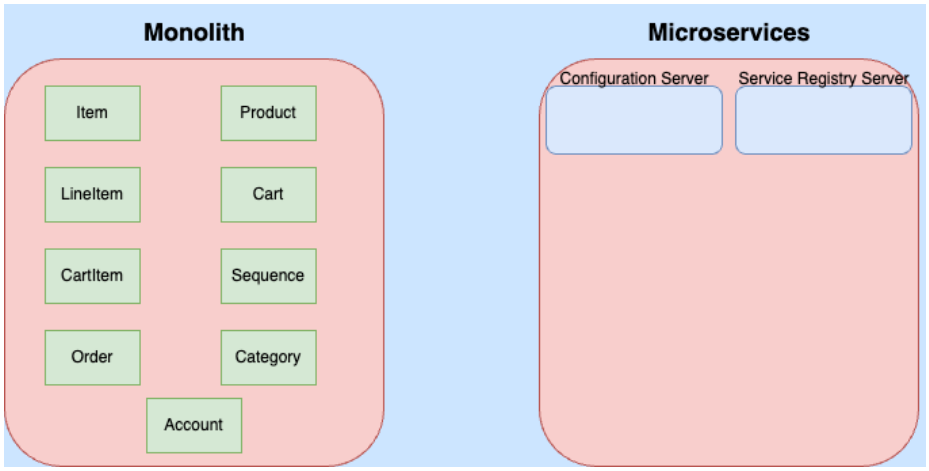

Figure 4.12: Monolith and Microservice projects after iteration 0.

4.4.2 Iteration 1

In the first iteration of the JPetStore project, we aim to identify and implement the initial microservice. Figure 4.13 illustrates the output of the tool for this iteration. The tool suggests several microservice candidates. Initially, candidates 2 and 3 appear suitable due to their perfect cohesion scores. However, based on my understanding of the bounded context, candidate 5, which focuses solely on the *Account* entity, emerges as the ideal choice. This entity handles critical functions such as user authentication, signup, and sign-in, making it suitable to operate as an independent microservice. Despite its lower cohesion score due to interactions with other methods, its core

functionality justifies its selection. Consequently, I implemented a new microservice named **Account Service**. Figure 4.14 shows the architecture of the monolithic and microservice projects after iteration 1.

```
------------------------------------------------------------  Found 6 microservices
Total coupling value: 5.499999999999986 (0.9166666666666643 avg.)
Cohesion value: 0.9345114678876182

Entities in Microservice 0:  Cohesion: 0.9546578730420444
- Order
- LineItem
Entities in Microservice 1:  Cohesion: 0.83271375464684
- Sequence
Entities in Microservice 2:  Cohesion: 0.9705449189985272
- Product
- Item
Entities in Microservice 3:  Cohesion: 1.0
- Category
Entities in Microservice 4:  Cohesion: 0.946808510638298
- CartItem
- Cart
Entities in Microservice 5:  Cohesion: 0.90234375
- Account


# of method calls crossing microservices: 3
# of entity references crossing microservices: 2
# of entity usages crossing microservices: 0
# of methods persisting entities of other microservices: 0
------------------------------------------------------------------------------
```

Figure 4.13: Output of the tool for the initial JPetStore project.

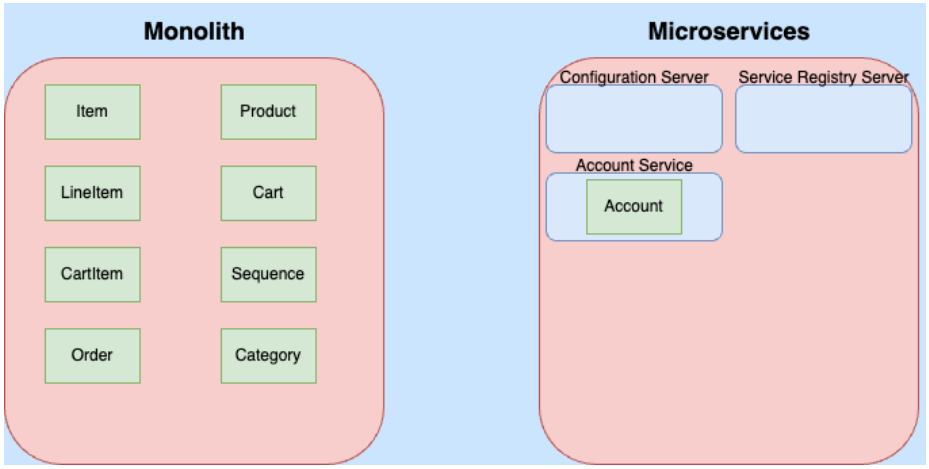

Figure 4.14: Monolith and Microservice projects after iteration 1.

We claimed that by incorporating the frequency of relationships between nodes

and integrating dynamic data, we have enhanced the accuracy of service identification compared to state-of-the-art approaches. This is logical, as adding more data allows us to model the system more comprehensively in the graph. While comparing average coupling is not ideal in this example—since the inclusion of frequency affects the weights—we instead focus on unweighted coupling, which is the total number presented at the end of the output. As shown in Figure 4.13, the output for the JPetstore project when considering relationship frequency and dynamic data results in a coupling value of 5 (3 + 2 + 0 + 0). In contrast, Figure 4.15 illustrates the output without these features, resulting in a coupling value of 8 (5 + 2 + 1 + 0). Although the average cohesion is slightly higher when these features are not included, this example, while limited, demonstrates how the inclusion of these features can lead to a more decoupled and accurate system decomposition. A more extensive assessment on a larger project would likely provide a clearer illustration of this improvement.

```
------------------------------------------------------------  Found 6 microservices
Total coupling value: 4.999999999999972 (0.8333333333333286 avg.)
Cohesion value: 0.9466439457228931

Entities in Microservice 0:  Cohesion: 0.9523809523809522
- Order
- LineItem
Entities in Microservice 1:  Cohesion: 1.0
- Sequence
Entities in Microservice 2:  Cohesion: 0.9539473684210525
- Product
- Item
Entities in Microservice 3:  Cohesion: 0.9090909090909091
- Category
Entities in Microservice 4:  Cohesion: 0.9444444444444445
- CartItem
- Cart
Entities in Microservice 5:  Cohesion: 0.9199999999999999
- Account



# of method calls crossing microservices: 5
# of entity references crossing microservices: 2
# of entity usages crossing microservices: 1
# of methods persisting entities of other microservices: 0
------------------------------------------------------------------------------
```

Figure 4.15: Output of the tool for the initial JPetStore project without considering the frequency of relationships and dynamic data.

4.4.3 Iteration 2

In the second iteration, based on the comparison in Figure 4.16, the *Category* service seems the most suitable candidate for extraction. However, considering the bounded context, *Category* is tightly connected to *Product* and *Item* entities. Therefore, I decided to develop a new microservice that combines candidates 2 and 3. This new microservice, **Product Service**, includes *Product*, *Item*, and *Category* entities. Figure 4.17 shows the system architecture after this implementation.

```
------------------------------------------------------------  Found 5 microservices
Total coupling value: 2.999999999999986 (0.5999999999999972 avg.)
Cohesion value: 0.9409450114651419

Entities in Microservice 0:  Cohesion: 0.9546578730420444
- Order
- LineItem
Entities in Microservice 1:  Cohesion: 0.83271375464684
- Sequence
Entities in Microservice 2:  Cohesion: 0.9705449189985272
- Product
- Item
Entities in Microservice 3:  Cohesion: 1.0
- Category
Entities in Microservice 4:  Cohesion: 0.946808510638298
- CartItem
- Cart


# of method calls crossing microservices: 1
# of entity references crossing microservices: 2
# of entity usages crossing microservices: 0
# of methods persisting entities of other microservices: 0
------------------------------------------------------------------------------
```

Figure 4.16: Output of the tool for the second iteration on JPetStore project.

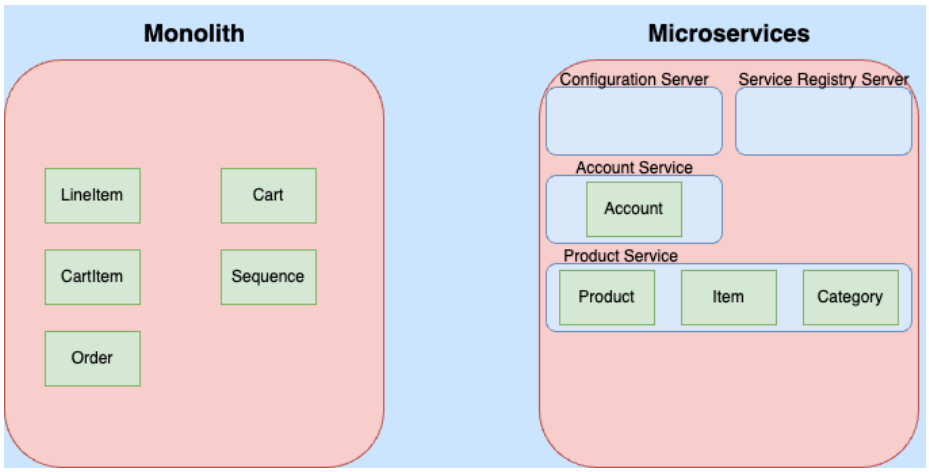

Figure 4.17: Monolith and Microservice projects after iteration 2.

4.4.4 Iteration 3

In this step, as shown in Figure 4.18, the best candidate is the one containing the *Cart* entity. However, considering the bounded context, *Cart* and *CartItem* are conceptually connected. Thus, a new microservice, **Cart Service**, is introduced containing both *Cart* and *CartItem* entities. Figure 4.19 illustrates the system architecture after this iteration.

```
-----
------------------------------------------------------------  Found 4 microservices
Total coupling value: 2.400000000000002 (0.6000000000000005 avg.)
Cohesion value: 0.8787333597701539

Entities in Microservice 0:  Cohesion: 0.9625935162094762
- Order
- LineItem
Entities in Microservice 1:  Cohesion: 0.83271375464684
- Sequence
Entities in Microservice 2:  Cohesion: 0.719626168224299
- CartItem
Entities in Microservice 3:  Cohesion: 1.0
- Cart


# of method calls crossing microservices: 0
# of entity references crossing microservices: 0
# of entity usages crossing microservices: 1
# of methods persisting entities of other microservices: 0
--------------------------------------------------------------------------------
```

Figure 4.18: Output of the tool for the third iteration on JPetStore project.

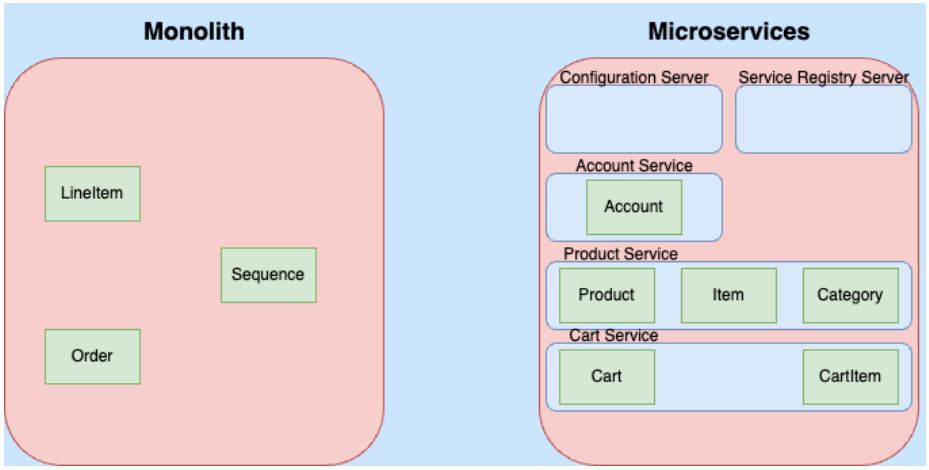

Figure 4.19: Monolith and Microservice projects after iteration 3.

4.4.5 Iteration 4

In the final iteration, we are left with two microservice candidates. These candidates are interconnected in the graph and conceptually close together. Therefore, developing *Sequence* as a new microservice is not advisable due to its close relation with *Order* and *LineItem* entities. Thus, the last microservice, **Order Service**, includes

Order, *LineItem*, and *Sequence* entities. Figure 4.21 shows the architecture at the end of the migration process.

```
%%%%%
------------------------------------------------------------  Found 2 microservices
Total coupling value: 1.8000000000000007 (0.9000000000000004 avg.)
Cohesion value: 0.8976536354281581

Entities in Microservice 0:  Cohesion: 0.9625935162094762
- Order
- LineItem
Entities in Microservice 1:  Cohesion: 0.83271375464684
- Sequence


# of method calls crossing microservices: 0
# of entity references crossing microservices: 0
# of entity usages crossing microservices: 0
# of methods persisting entities of other microservices: 0
-------------------------------------------------------------------------------
```

Figure 4.20: Output of the tool for the fourth iteration on JPetStore project.

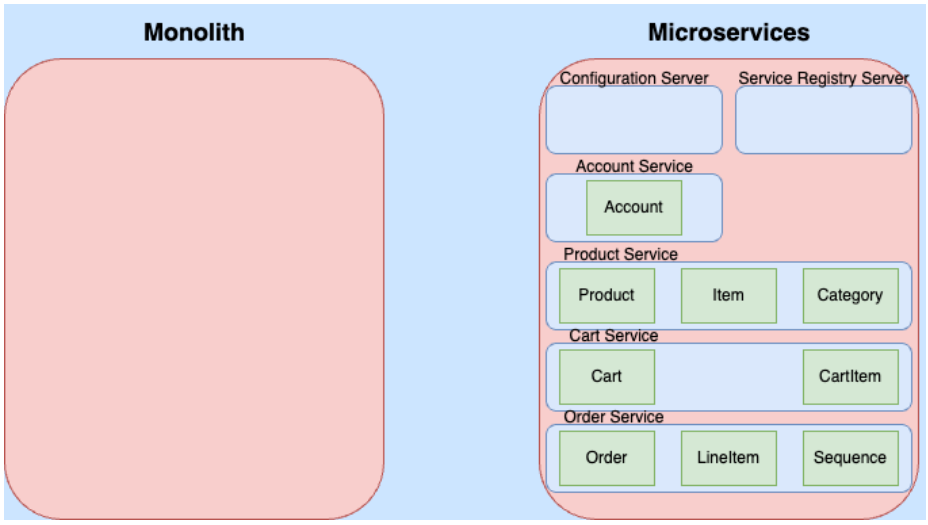

Figure 4.21: Monolith and Microservice projects at the end of migration process.

4.4.6 Final Result

Table 4.3 presents the comparison of the results obtained from our approach with other approaches for the JPetStore project. Filippone et al.'s approach serves as

the baseline. The table shows the *Average Coupling* and *Average Cohesion* metrics, along with the *IFN* (Interface Number) for different approaches. Our approach demonstrates competitive results in terms of cohesion and coupling, confirming the effectiveness of the proposed methodology.

Approach	**# Microservices**	**Average Coupling**	**Average Cohesion**	**IFN**
Zaragoza et al. [30]	3	3.13	0.90	2
Selmadji et al. [27]	2	3.8	0.93	2.5
Jin et al. [14]	4	2.25	0.82	1.75
Brito et al. [6]	4	3.1	0.85	1.75
Filippone (BaseLine)	3	0.87	0.97	1.7
Our approach	4	0.75	0.94	1.75

Table 4.3: Result comparison for the JPetStore project.

4.5 Conclusion

In this chapter, we evaluated our approach against other state-of-the-art methodologies using two publicly available Java projects: Spring Petclinic and MyBatis JPetStore. The evaluation criteria focused on the independence of functionality, measured by the IFN metric, and modularity, assessed through average cohesion and coupling metrics.

Our approach demonstrated competitive results across both projects. The iterative and incremental process allowed for a systematic decomposition of the monolithic systems into microservices, effectively balancing cohesion and coupling. This was particularly evident in the JPetStore experiment, where our approach achieved the lowest average coupling and high average cohesion, confirming its effectiveness in creating

well-modularized microservices.

Moreover, the application of dynamic analysis data, along with static analysis, provided a comprehensive understanding of the system's behavior, enhancing the accuracy of the microservice identification process. The use of the SST paradigm ensured data consistency and reliability throughout the migration, contributing to the correctness of the results.

Overall, the evaluations indicate that our approach not only meets but often exceeds the performance of existing methodologies in achieving the fundamental principles of microservice architecture. This reaffirms the potential of our framework as a robust solution for the gradual and systematic migration of monolithic systems to microservices, ensuring minimal disruption and maintaining system functionality throughout the process.

Chapter 5

Conclusion

5.1 Summary of Contributions

This thesis addresses a significant gap in the existing literature regarding the migration from monolithic architectures to microservices using the Strangler Fig Pattern. The primary contribution of this work is the development of a semi-automated tool that aids in the gradual decomposition of monolithic systems into microservices. Unlike existing tools that typically aim for a single iteration decomposition, our approach leverages a gradual migration strategy, aligning with real-world practices where such transitions are implemented step-by-step to minimize risk and ensure continuity.

Our framework builds on the state-of-the-art methodology proposed by Filippone et al., which uses static code analysis for service identification. We enhanced this approach by incorporating dynamic analysis data, addressing the challenge of data consistency and reliability through the Single Source of Truth (SST) paradigm. The tool developed as part of this research was evaluated using two well-known Java Spring projects: Spring Petclinic and MyBatis JPetStore. The results demonstrate

that our tool can effectively identify microservice candidates with high cohesion and low coupling, validating the practicality and effectiveness of the proposed approach.

The main objectives of this research were to:

- Develop a tool that facilitates the gradual migration from monolith to microservices using the Strangler Fig Pattern.
- Integrate static and dynamic analysis data to improve the accuracy of service identification.
- Ensure data consistency and reliability through the SST paradigm.
- Evaluate the tool on real-world projects and compare its performance with existing approaches.

All these objectives have been successfully achieved. The developed tool not only assists in identifying and extracting microservices but also supports the gradual migration strategy, which is more aligned with industry practices. The integration of static and dynamic data has shown to enhance the accuracy of the decomposition, and the use of the SST paradigm has ensured the reliability of the data used in the analysis.

The evaluation of the tool on the Spring Petclinic and MyBatis JPetStore projects showed promising results. Our approach outperformed some of the existing methods in terms of achieving higher cohesion and lower coupling in the identified microservices. Specifically, our tool's ability to operate in a gradual manner and refine the decomposition iteratively is a significant advantage over the single iteration approaches prevalent in the literature.

5.2 Threats to Validity

In this section, we discuss potential threats to the validity of our approach and its evaluation.

User Knowledge Dependency One of the main threats to validity is the high dependency on the user's knowledge. In the evaluation of our approach, I acted as the expert and used my knowledge of the system under study to find the final decomposition. This introduces a potential bias, as the results are influenced by my personal understanding and decisions.

Heuristic Nature of Metrics The metrics used for evaluation, such as cohesion and coupling, are heuristic and might not capture all aspects of an optimal microservice. Different metrics or a combination of metrics could yield different results, potentially influencing the perceived effectiveness of our approach.

External Validity The generalizability of our findings to real-world industrial settings might be limited. The controlled environment of the experiments may not fully capture the complexities and challenges faced in actual migration projects. To address this, future assessments should consider larger and more complex projects that closely resemble real-world enterprise systems. These projects should feature substantial codebases, diverse technologies, and intricate interdependencies between services, thereby providing a more rigorous test of the tool's capabilities.

Language Dependency Another threat to validity is that the current implementation of the tool only works for Java projects. While the underlying concepts can be

applied to other programming languages, the tool's applicability and effectiveness in non-Java environments remain untested.

Acknowledging these threats provides a comprehensive understanding of the limitations of our study and highlights areas for further research.

5.3 Future Work

While this research has made significant strides in improving the migration process from monolithic to microservice architectures, there are several areas for future work:

- Expanding the tool to support more programming languages and frameworks.
- Enhancing the dynamic analysis component to capture more complex runtime behaviors.
- Incorporating machine learning techniques to further automate the service identification process.
- Conducting more extensive evaluations on a wider range of projects to validate the generalizability of the approach.

5.4 Final Thoughts

In conclusion, this thesis has presented a novel approach to the challenging problem of migrating from monolithic to microservice architectures. By adopting a gradual migration strategy and leveraging both static and dynamic analysis data, the developed tool provides a practical and effective solution for industry practitioners. The

promising results from the evaluation underscore the potential of this approach to significantly ease the transition to microservices, paving the way for more scalable and maintainable software architectures.

Bibliography

[1] Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley, and P. Clarke. Decomposition of monolith applications into microservices architectures: A systematic review. *IEEE Transactions on Software Engineering*, 49(8):4213–4242, 2023. doi: 10.1109/TSE.2023.3287297.

[2] E. R. Barnes. An algorithm for partitioning the nodes of a graph. In *1981 20th IEEE Conference on Decision and Control including the Symposium on Adaptive Processes*, pages 303–304, 1981. doi: 10.1109/CDC.1981.269534.

[3] D. Bertsimas, C.-P. Teo, and R. Vohra. Analysis of lp relaxations for multiway and multicut problems. *Networks*, 34(2):102–114, 1999. doi: https://doi.org/10.1002/(SICI)1097-0037(199909)34:2⟨102::AID-NET3⟩3.0.CO;2-X. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-0037%28199909%2934%3A2%3C102%3A%3AAID-NET3%3E3.0.CO%3B2-X`.

[4] D. Bertsimas, C.-P. Teo, and R. V. Vohra. Analysis of lp relaxations for multiway and multicut problems. *Networks*, 34:102–114, 1999. URL `https://api.semanticscholar.org/CorpusID:7584343`.

[5] V. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding

of communities in large networks. *Journal of Statistical Mechanics Theory and Experiment*, 2008, 04 2008. doi: 10.1088/1742-5468/2008/10/P10008.

[6] M. Brito, J. Cunha, and J. a. Saraiva. Identification of microservices from monolithic applications through topic modelling. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 1409–1418, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450381048. doi: 10.1145/3412841.3442016. URL `https://doi.org/10.1145/3412841.3442016`.

[7] R. Chen, S. Li, and Z. Li. From monolith to microservices: A dataflow-driven approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475, 2017. doi: 10.1109/APSEC.2017.53.

[8] N. Dragoni, I. Lanese, S. Larsen, M. Mazzara, R. Mustafin, and L. Safina. Microservices: How to make your application scale, 02 2017.

[9] G. Filippone, N. Qaisar Mehmood, M. Autili, F. Rossi, and M. Tivoli. From monolithic to microservice architecture: an automated approach based on graph clustering and combinatorial optimization. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, pages 47–57, 2023. doi: 10.1109/ICSA56044.2023.00013.

[10] M. Fowler and J. Lewis. Microservices: a definition of this new architectural term, 2014. URL `https://martinfowler.com/articles/microservices.html`. Accessed: 2024-05-21.

[11] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann. Microservices migration in industry: Intentions, strategies, and challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 481–490, Los Alamitos, CA, USA, oct 2019. IEEE Computer Society. doi: 10.1109/ICSME.2019.00081. URL `https://doi.ieeecomputersociety.org/10.1109/ICSME.2019.00081`.

[12] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002. doi: 10.1073/pnas.122653799. URL `https://www.pnas.org/doi/abs/10.1073/pnas.122653799`.

[13] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann. Service cutter: A systematic approach to service decomposition. pages 185–200, 09 2016. ISBN 978-3-319-44481-9. doi: 10.1007/978-3-319-44482-6_12.

[14] W. Jin, T. Liu, Q. Zheng, D. Cui, and Y. Cai. Functionality-oriented microservice extraction based on execution trace clustering. In *2018 IEEE International Conference on Web Services (ICWS)*, pages 211–218, 2018. doi: 10.1109/ICWS.2018.00034.

[15] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng. Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering*, 47(5):987–1007, 2021. doi: 10.1109/TSE.2019.2910531.

[16] M. Kamimura, K. Yano, T. Hatano, and A. Matsuo. Extracting candidates of microservices from monolithic application code. In *2018 25th Asia-Pacific*

Software Engineering Conference (APSEC), pages 571–580, 2018. doi: 10.1109/APSEC.2018.00072.

[17] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970. doi: 10.1002/j.1538-7305.1970.tb01770.x.

[18] C.-Y. Li, S.-P. Ma, and T.-W. Lu. Microservice migration using strangler fig pattern: A case study on the green button system. In *2020 International Computer Symposium (ICS)*, pages 519–524, 2020. doi: 10.1109/ICS51289.2020.00107.

[19] Z. Li, C. Shang, J. Wu, and Y. Li. Microservice extraction based on knowledge graph from monolithic applications. *Inf. Softw. Technol.*, 150(C), oct 2022. ISSN 0950-5849. doi: 10.1016/j.infsof.2022.106992. URL `https://doi.org/10.1016/j.infsof.2022.106992`.

[20] T. Matias, F. F. Correia, J. Fritzsch, J. Bogner, H. S. Ferreira, and A. Restivo. Determining microservice boundaries: A case study using static and dynamic software analysis. 2020.

[21] R. Müller, D. Mahler, M. Hunger, J. Nerche, and M. Harrer. Towards an open source stack to create a unified data source for software analysis and visualization. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 107–111, 2018. doi: 10.1109/VISSOFT.2018.00019.

[22] M. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 69: 026113, 03 2004. doi: 10.1103/PhysRevE.69.026113.

[23] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, June 2006. ISSN 1091-6490. doi: 10.1073/pnas.0601602103. URL `http://dx.doi.org/10.1073/pnas.0601602103`.

[24] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2015.

[25] S. Salii, J. Ajdari, and X. Zenuni. Migrating to a microservice architecture: benefits and challenges. In *2023 46th MIPRO ICT and Electronics Convention (MIPRO)*, pages 1670–1677, 2023. doi: 10.23919/MIPRO57284.2023.10159894.

[26] S. Santos and A. Silva. Microservices identification in monolith systems: Functionality redesign complexity and evaluation of similarity measures. *Journal of Web Engineering*, 08 2022. doi: 10.13052/jwe1540-9589.2158.

[27] A. Selmadji, A.-D. Seriai, H. L. Bouziane, R. Oumarou Mahamane, P. Zaragoza, and C. 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, 2020. doi: 10.1109/ICSA47634.2020.00023.

[28] I. Trabelsi, M. Abdellatif, A. Abubaker, N. Moha, S. Mosser, S. Ebrahimi-Kahou, and Y.-G. Guéhéneuc. From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis. *Journal of Software: Evolution and Process*, 35, 09 2022. doi: 10.1002/smr.2503.

[29] E. Volynsky, M. Mehmed, and S. Krusche. Architect: A framework for the migration to microservices. In *2022 International Conference on Computing,*

Electronics Communications Engineering (iCCECE), pages 71–76, 2022. doi: 10.1109/iCCECE55162.2022.9875096.

[30] P. Zaragoza, A.-D. Seriai, A. Seriai, A. Shatnawi, and M. Derras. Leveraging the layered architecture for microservice recovery. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 135–145, 2022. doi: 10.1109/ICSA53651.2022.00021.