A Machine learning-based Framework to Support Monoliths to Microservices Migration: From Identification to Packaging and Deployment

by

Imen TRABELSI

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
Ph.D.

MONTREAL, 18 AUGUST 2025

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mrs. Naouel Moha, Thesis supervisor
Department of Software and IT Engineering, École de technologie supérieure ÉTS

Mr. Yann-Gaël Guéhéneuc, Thesis Co-Supervisor
Departement of Computer Science and Software Engineering, Concordia university

Mr. Lokman Sboui, Chair, Board of Examiners
Department of System Engineering, École de technologie supérieure ÉTS

Mrs. Ghizlane El Boussaidi, Member of the Jury
Department of Software and IT Engineering, École de technologie supérieure ÉTS

Mr. Julien Gascon-Samson, Member of the Jury
Department of Software and IT Engineering, École de technologie supérieure ÉTS

Mr. Mohamed Aymen Saied, External Independent Examiner
Department of Computer Science and Software Engineering, Laval university

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON 18 AUGUST 2025

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

I am deeply grateful to my supervisors, Prof. Naouel Moha and Prof. Yann-Gaël Guéhéneuc, for their invaluable guidance, continuous support, and generous mentorship. Their complementary expertise, constructive feedback, and unwavering belief in my work have profoundly shaped the direction and quality of this research. I feel privileged to have worked under their supervision.

I also warmly thank all my colleagues from the ptiDej team for their collaboration, helpful discussions, and the friendly and inspiring work atmosphere they created. Your support and companionship have made this journey both productive and enjoyable.

I would like to sincerely thank the members of the jury for accepting to evaluate my thesis and for their insightful comments: Prof. Lokman Sboui, Prof. Mohamed Aymen Saied, Prof. Ghizlane El Boussaidi, and Prof. Julien Gascon-Samson.

My sincere thanks go to the Department of Software and IT Engineering at ÉTS and the Department of Computer Science and Software Engineering at Concordia University for providing a rich and supportive research environment.

To Oussama, my partner, my anchor, and my greatest source of strength, thank you. We've been through so much together, and yet you stood by me at every step, with patience, love, and unwavering support. In moments of doubt, you reminded me of my worth; in moments of exhaustion, you gave me courage. This journey would have been unimaginable without you by my side.

To my beloved parents, Adel and Khaoula, thank you for supporting me all along my life, for always believing in me, and for sacrificing so much for us. Your love, strength, and resilience have been my foundation. This achievement is as much yours as it is mine, and I am endlessly grateful for everything you have done to help me reach this point.

To my dear sisters, Tesnim, Sirine, and May, I may be the oldest, but I never felt the pressure of being so, because your love, support, and presence always lifted me up. Thank you for being my

cheerleaders, my safe space, and my constant source of joy. Your strength and kindness have inspired me more than you know, and I am so proud to walk this path with you by my side.

To my grandmother, the peaceful soul I lost during this journey, your absence was deeply felt every step of the way. You will always be missed, and your memory lives on in my heart. To my other beloved grandmother, Mima, I wish you many more years by our side, in good health and joy. Your warmth and prayers have always been a quiet strength in my life.

To my friends and colleagues, thank you for your support, insightful discussions, and all the moments of shared motivation that have made this experience brighter and more meaningful.

This thesis is dedicated to all those who strive to pursue knowledge with passion and perseverance.

Un cadriciel basé sur l'apprentissage automatique pour aider à la migration des systèmes monolithiques vers les microservices : de l'identification à l'emballage et au déploiement

Imen TRABELSI

RÉSUMÉ

La migration des systèmes monolithiques vers les microservices est un sujet d'actualité en raison de la demande croissante de systèmes logiciels évolutifs, faciles à maintenir et agiles. Bien que les microservices offrent des avantages, tels que le déploiement indépendant, l'isolation des défaillances et une meilleure évolutivité, la migration d'un système monolithique existant est un processus complexe, demandeur en ressources et propice aux erreurs. Ce processus comporte plusieurs phases interdépendantes : la pré-migration, l'identification des services, l'empaquetage des services, le déploiement et la surveillance, chacune présentant des défis distincts.

Les approches de migration existantes reposent actuellement sur des techniques manuelles guidées par des experts, rendant le processus coûteux, long et propice aux erreurs. Les avancées récentes en apprentissage automatique et en grands modèles de langage (LLMs) offrent de nouvelles opportunités pour analyser le code source, apprendre des gabarits à partir de systèmes existants et générer du code source ou des fichiers de configurations, ouvrant ainsi des perspectives concrètes pour automatiser et optimiser le processus de migration. Bien que certaines phases, comme l'identification des services, aient été largement étudiées, d'autres, telles que l'empaquetage des microservices et la configuration du déploiement, restent peu explorées et nécessitent encore beaucoup d'intervention manuelle.

Cette thèse débute par une revue systématique de la littérature (SLR) portant sur 81 études primaires publiées entre 2015 et 2024, qui se concentre sur les approches de migration basées sur l'apprentissage automatique. Cette SLR fournit une analyse approfondie des techniques existantes et montre que, si l'identification des services a été bien étudiée, les phases d'empaquetage et de configuration du déploiement demeurent insuffisamment explorées et largement dépendantes d'un travail manuel.

S'appuyant sur ces constats, cette thèse propose un cadriciel fondé sur l'apprentissage automatique pour automatiser trois phases clés du processus de migration : l'identification, l'empaquetage et le déploiement. Ce cadre comprend les approches suivantes : (1) **Identification des services** via *MicroMiner*, qui exploite le clustering et l'analyse sémantique pour décomposer des systèmes monolithiques en microservices tout en considérant les principes de responsabilité unique et de contexte borné ; (2) **Empaquetage des services** via *MicroPacker*, qui utilise une transformation guidée par un LLM pour modulariser le code, générer des API standardisées et intégrer des patrons fondamentaux d'architecture des microservices ; et (3) **Déploiement des services** via *MiDKo*, qui applique une génération augmentée par récupération (RAG) pour produire automatiquement des configurations Kubernetes et Docker précises.

Toutes ces contributions minimisent l'intervention manuelle, améliorent la qualité architecturale et la modularité des microservices résultants et garantissent la génération d'artefacts de

déploiement valides, ce qui permet d'obtenir des microservices cohésifs et déployables, prêts à être exécutés dans des environnements cloud.

Mots-clés: Microservices, Monolithe, Migration, Identification, Empaquetage, Déploiement, Apprentissage automatique

A Machine learning-based Framework to Support Monoliths to Microservices Migration: From Identification to Packaging and Deployment

Imen TRABELSI

ABSTRACT

The migration from monolithic to microservices has gained significant attention due to the growing demand for scalable, maintainable, and agile software systems. While microservices offer benefits, such as independent deployment, fault isolation, and enhanced scalability, migrating an existing monolithic system is a complex, resource-intensive, and error-prone process. This process involves multiple interdependent phases: pre-migration, service identification, service packaging, deployment, and monitoring, each presenting distinct challenges.

Existing migration approaches rely heavily on expert-driven, manual techniques, making the process costly, time-consuming, and error-prone. Recent advancements in Machine Learning and Large Language Models provide new capabilities for analysing source code, learning patterns from existing systems, and generating code or configurations, thus offering concrete opportunities to automate and optimise the migration process. While some phases, such as service identification, have been extensively studied, others, such as microservice packaging and deployment configuration, remain underexplored and require substantial manual effort.

This thesis begins with a Systematic Literature Review (SLR) of 81 primary studies on machine learning-based approaches to monolith-to-microservices migration, published between 2015 and 2024. The SLR provides a comprehensive analysis of existing techniques and reveals that, while service identification has been extensively studied, the phases of microservice packaging and deployment configuration remain underexplored and largely reliant on manual effort.

Building on these findings, this thesis introduces a machine learning-driven framework that automates the three key phases of the migration process, identification, packaging, and deployment, thereby reducing manual effort and enabling the transition toward a modular and cohesive microservices architecture. The framework comprises the following ML-based approaches: (1) **Service Identification** via *MicroMiner*, which leverages clustering and semantic analysis to decompose monolithic systems into microservices aligned with single-responsibility and bounded context principles; (2) **Service Packaging** via *MicroPacker*, which employs LLM-guided transformation to modularise code, generate standardised APIs, and integrate foundational microservice patterns; and (3) **Service Deployment** via *MiDKo*, which applies retrieval-augmented generation to automatically produce accurate Kubernetes and Docker configurations.

Collectively, these contributions minimise manual intervention, enhance the architectural quality and modularity of the resulting microservices, and ensure the generation of valid deployment artifacts, resulting in cohesive and deployable microservices ready for execution in modern cloud environments.

Keywords: Microservices, Monolith, Migration, Identification, Packaging, Deployment, Machine learning

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
API	Application Programming Interface
BM25	Best Matching 25 (ranking function for information retrieval)
CHD	Cohesion at Domain Level
CHM	Cohesion at Message Level
CMQ	Conceptual Modularity Quality
CRUD	Create, Read, Update, Delete
DAO	Data Access Object
DB	Domain Boundary
DLL	Dynamic-Link Library
ERP	Enterprise Resource Planning
ETS	École de Technologie Supérieure
FCM	Fuzzy C-Means
GCN	Graph Convolutional Network
GNN	Graph Neural Network
IFN	Interface Number
K8s	Kubernetes
KDM	Knowledge Discovery Metamodel
KNN	K-Nearest Neighbour

LLM	Large Language Model
ML	Machine Learning
MSA	Microservices Architecture
MVC	Model-View-Controller
NLP	Natural Language Processing
OMG	Object Management Group
PS	Primary Study
RAG	Retrieval-Augmented Generation
REST	Representational State Transfer
RQ	Research Questions
SE	Software Engineering
SLR	Systematic Literature Review
SMQ	Structural Modularity Quality
SOA	Service-Oriented Architecture
SP	Single Responsibility
SVM	Support Vector Machine
URL	Uniform Resource Locator
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

INTRODUCTION

0.1 Context

Monolithic systems, despite their historical prevalence, suffer from several architectural and operational limitations. The tight coupling between their components makes it difficult to isolate faults, scale individual functionalities, or adopt new technologies incrementally. A change in one part of the system often requires redeployment of the entire system, increasing the risk of system-wide failures and lengthening release cycles. Moreover, as the codebase grows, onboarding new developers, maintaining system modularity, and understanding system behavior become increasingly difficult. These challenges hinder agility and pose a barrier to continuous delivery (Fritzsch, Bogner, Wagner & Zimmermann, 2019b).

The microservices architecture (MSA) has emerged as a dominant paradigm in modern software engineering, offering solutions to the architectural and operational limitations of monolithic systems. By decomposing complex systems into independently deployable services, each focused on a specific business capability, MSA mitigates the issue of tight coupling and allows faults to be isolated more effectively. These loosely coupled services communicate via lightweight protocols such as REST APIs or message queues (Nadareishvili, Mitra, McLarty & Amundsen, 2016), enabling incremental adoption of new technologies and reducing the need for system-wide redeployments. This modular structure also supports faster release cycles, facilitates onboarding of new developers, and enhances system maintainability. Overall, microservices foster agility, scalability, and resilience, making them especially suitable for organisations operating at scale.

Industry leaders, such as Amazon, eBay, Netflix, Spotify, and SoundCloud (Varshneya, 2022, 2023; Calçado, 2012), have demonstrated the effectiveness of MSA in achieving rapid release cycles, independent scalability, and fault isolation (Newman, 2015a). This success has led to widespread interest in transitioning existing monolithic systems, where all components are

intertwined within a single source code base toward more flexible and maintainable microservices-based architecture. Despite its benefits, the migration from monolith to microservices is a technically demanding and error-prone process (Fritzsch, Bogner, Wagner & Zimmermann, 2019a). It involves multiple interrelated activities, including understanding the monolithic system, identifying suitable service boundaries, refactoring legacy code, (re)defining dependencies and APIs, and configuring modern continuous delivery systems.

Following prior works on microservices migration, Abgaz *et al.* (2023b), Fritzsch, Bogner, Zimmermann & Wagner (2019c), and Saucedo, Rodríguez, Rocha & dos Santos (2024), we structured the migration process into five phases: Pre-migration, Identification, Packaging, Deployment, and Monitoring. In this thesis, we focus specifically on the three core migration phases: Identification, Packaging, and Deployment, which directly involve transforming monolithic source code into independently deployable microservices, as shown in Figure 0.1.

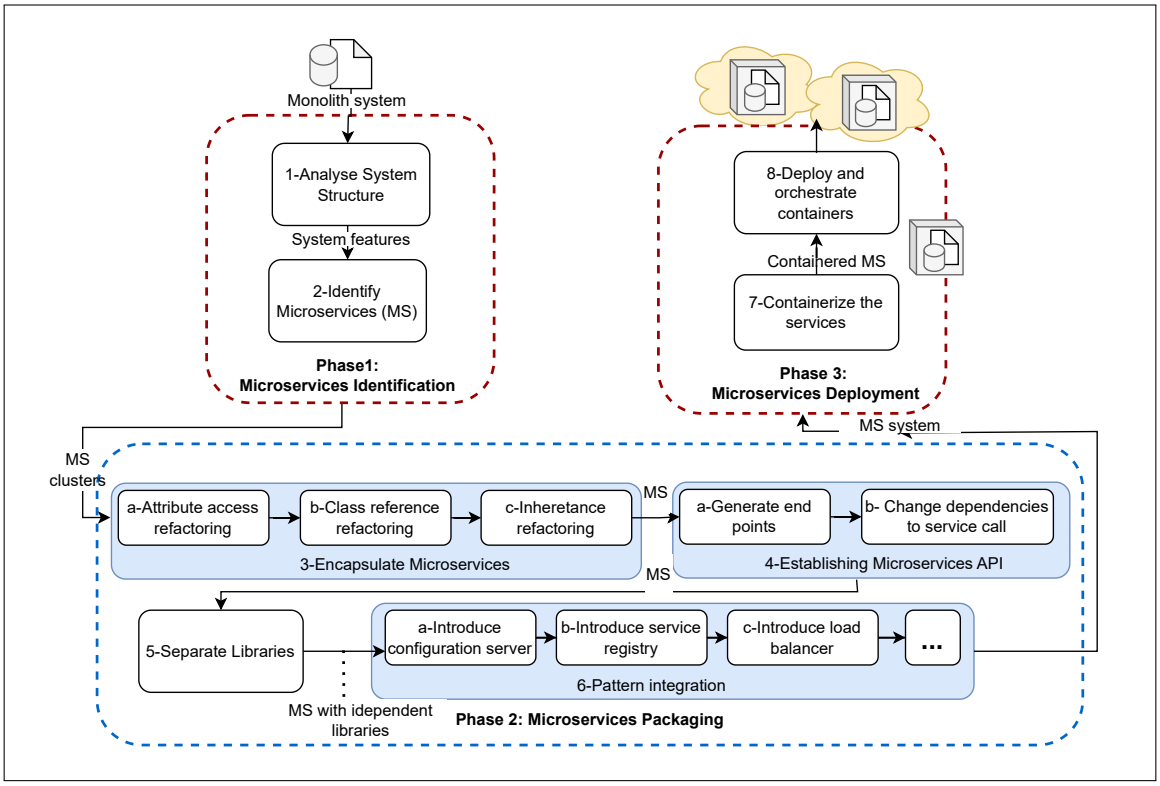

Figure 0.1 Detailed Migration Process

The Identification phase begins with analysing the structure of a monolithic system to understand its components, dependencies, and its internal interactions. Based on this understanding, engineers (or automated tools) decompose the system into clusters that represent candidate microservices. The clustering process is guided by the principles of Single Responsibility and Bounded Context, where each candidate microservice should encapsulate a cohesive business function and own its internal logic and data. This decomposition ensures that each service can evolve independently and that inter-service communication is minimised. The result of this phase is a set of proposed candidate microservices.

The Packaging phase is the next phase. Once candidate microservices are identified, they are transformed into operational microservices. This phase involves encapsulating each microservice by eliminating direct inter-dependencies, such as candidate microservices attribute access, class references, or inheritance. Each service is then equipped with a well-defined API, exposing necessary functionality while hiding internal complexity. Direct code dependencies are replaced with service calls, and shared libraries are either duplicated or externalised to ensure independence of microservices. To support distributed operations, essential architectural patterns are integrated, including centralised configuration servers, service registries for dynamic discovery, and load balancers for traffic distribution and fault tolerance.

The Deployment phase finalises the migration by preparing the microservices for execution in cloud or hybrid environments. Each service is containerised to encapsulate its runtime environment, ensuring portability and reproducibility. Orchestration platforms, such as Kubernetes, are used to manage deployment, service scaling, and failure recovery. This phase includes the configuration of deployment artifacts, such as Dockerfiles and Kubernetes manifests, which define container specifications, service endpoints, and resource allocations (e.g., CPU and memory limits). It also involves configuring the deployment environment and optimising resource usage to ensure cost-efficiency and performance under expected workloads.

Historically, research and practice have addressed the automation of parts of the migration process using heuristic-driven techniques, such as static dependency analysis, software metrics, and rule-based clustering (Abdellatif *et al.*, 2021b). Other approaches have leveraged metaheuristic search methods, like evolutionary algorithms (Sellami, Ouni, Saied, Bouktif & Mkaouer, 2022a), to optimise service decomposition. These techniques have been successfully applied in various contexts, but they often suffer from limited generalisability, require careful parameter tuning, and typically lack support for automation beyond the identification phase.

In recent years, the software engineering (SE) community has increasingly embraced artificial intelligence (AI) to support, optimise, and automate tasks throughout the development lifecycle. From code completion and bug detection to documentation generation and program repair, AI has demonstrated considerable potential to enhance developer productivity (Peng, Kalliamvakou, Cihon & Demirer, 2023), reduce manual workload, and improve software quality (Vaithilingam, Zhang & Glassman, 2022; Allamanis, Barr, Devanbu & Sutton, 2018).

At the forefront of this transformation are Large Language Models (LLMs), which are increasingly being used by both researchers and practitioners to automate various tasks in software engineering. These models have been successfully adapted to capture the syntax, semantics, and structural patterns of source code (Xu, Alon, Neubig & Hellendoorn, 2022; Chen *et al.*, 2021). Models such as CodeBERT (Feng *et al.*, 2020), CodeT5 (Wang, Wang, Joty & Hoi, 2021), and more recent LLMs, including LLaMA (Touvron *et al.*, 2023), DeepSeek (Liu *et al.*, 2024), and GPT (Achiam *et al.*, 2023), have been solving a growing array of software engineering tasks with strong performance and minimal reliance on handcrafted rules. While prompt and context engineering still require manual effort, these models significantly reduce the need for domain-specific heuristics or hardcoded logic.

Given these capabilities, ML and LLMs offer a promising research direction for automating the migration process beyond the identification phase, particularly in packaging and deployment,

which remain underexplored and poorly supported in current approaches. By training on large-scale datasets that include both monolithic and microservice-based systems, as well as accompanying infrastructure configuration, these models can learn contextual patterns to perform transformations, recommend code refactorings, generate service APIs, and synthesise deployment specifications. This learning-based strategy addresses limitations not only of rule- and template-based tools but also of heuristic-driven techniques, which often require fine-tuned parameters and are difficult to generalise. Overall, such models can enable more scalable, adaptable, and generalisable migration processes by supporting end-to-end automation with reduced manual intervention.

0.2 Problem

As we mentioned in the context and summarised in Figure 0.2, the migration process can be structured into three key technical phases: service identification, service packaging, and service deployment.

Insights from our Systematic Literature Review (SLR) reveal that, although the identification phase has been extensively studied in previous works, the packaging and deployment phases remain significantly underexplored. Consequently, comprehensive automation across the full migration framework is still lacking.

Service Identification aims to decompose the monolith into candidate microservices. Existing approaches predominantly rely on static dependency analysis, rule-based clustering, and software metrics, especially coupling and cohesion (Fritzsch, Bogner, Zimmermann & Wagner, 2018; Hassan, Ali & Bahsoon, 2017; Brito, Cunha & Saraiva, 2021). However, these techniques often fall short in real-world scenarios due to their limited support for principles like bounded context and single responsibility, and their dependency on unavailable domain artifacts (e.g., use cases or

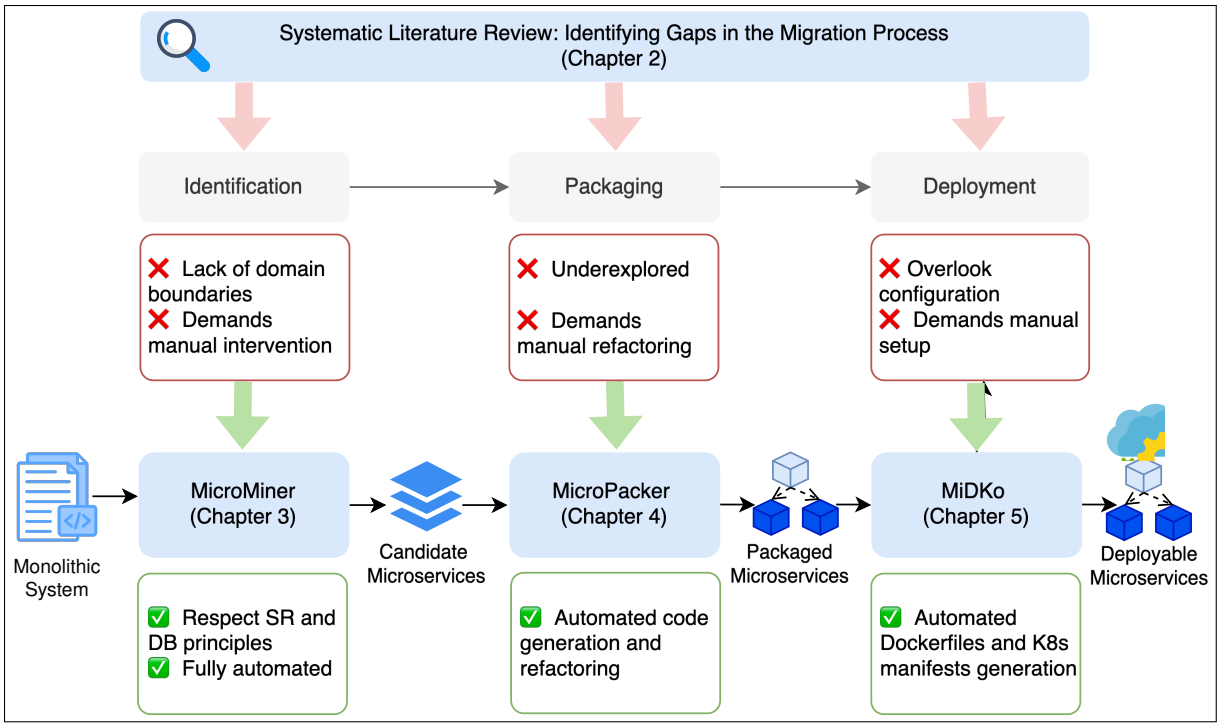

Figure 0.2 Overview of the core challenges in monolith to microservices migration, as identified by our SLR, and the corresponding contributions of this thesis. The figure highlights gaps in automation and tool support across three critical phases: service identification, packaging, and deployment, addressed by the proposed ML-based framework comprising MicroMiner, MicroPacker, and MiDKo

business process models). Manual validation is often required to ensure architectural relevance and domain alignment, reducing scalability and generalisability.

Service Packaging focuses on transforming identified clusters into independently deployable microservices by refactoring the code, creating service interfaces, and integrating architectural patterns, such as service registries or configuration servers. Existing works, e.g., (Zaragoza *et al.*, 2021), in this phase rely heavily on handcrafted rules and transformations, which are brittle and difficult to adapt across systems with varying characteristics and business logic. As a result, packaging remains a labor-intensive task, often requiring developers to manually isolate code, rewrite dependencies, and establish inter-service communication protocols. LLMs have shown the ability to perform context-aware code transformations (Chen *et al.*, 2021), synthesise

missing glue code, and adapt architectural patterns, such as service registries and proxies, to new scenarios (Hou *et al.*, 2023). Unlike rule-based tools, LLMs can infer the intent behind method calls, interface boundaries, or configuration templates, making them well-suited to automate service packaging in a more flexible and scalable manner.

Service Deployment involves generating deployment artifacts, such as Dockerfiles and Kubernetes manifests, that define how microservices are built, configured, and orchestrated. However, most existing research addressing this phase has focused primarily on optimising resource allocation and runtime efficiency (e.g., CPU/memory scheduling, autoscaling policies) (Saucedo, Rodríguez, Rocha & dos Santos, 2025). In contrast, the initial automation of deployment artifacts generation remains underexplored. Industry tools, such as Helm and Kompose, rely on static templates that produce generic configurations. These generic outputs lack the contextual awareness needed to reflect the specific structure and dependencies of each microservice. These solutions often demand substantial manual effort to customise configurations for each microservice, increasing the risk of misconfigurations and introducing considerable operational overhead. The complexity is further compounded by the steep learning curve of orchestration platforms, like Kubernetes, and the shortage of experienced practitioners (Zhu & Gehrmann, 2022; Zhang *et al.*, 2024). Unlike template-based tools, LLMs can generate configuration artifacts based on the system requirements and user inputs.

To address these limitations, this thesis proposes a unified, ML-driven framework that leverages machine learning and large language models to automate key tasks across all three phases. Our contributions include: (1) *MicroMiner*, a type-based service identification approach that uses semantic analysis and classification to produce architecturally relevant microservices; (2) *MicroPacker*, a prompt-driven packaging process that restructures code, extracts APIs, and integrates runtime patterns using LLMs; and (3) *MiDKo*, a deployment automation solution that combines topic modeling and retrieval-augmented generation to produce high-quality,

context-aware Dockerfiles and Kubernetes manifests. These contributions reduce reliance on manual effort, improve the architectural integrity of generated microservices, and ensure that deployment artifacts are both valid and adaptable.

Figure 0.2 presents an overview of the challenges identified by our SLR, along with the contributions of this thesis that aim to address them using machine learning and large language models.

Thesis Statement

This thesis investigates whether monolith-to-microservices migration can be fully automated, while ensuring accuracy and minimising manual effort, by leveraging machine learning and large language models across the phases of service identification, packaging, and deployment.

0.3 Contributions

This section outlines the main contributions of the thesis. In addition to the primary contributions that directly support the proposed framework for automating monolith-to-microservices migration, we also present related and collaborative contributions that address complementary challenges in microservices and software engineering more broadly.

0.3.1 Primary Contributions

This thesis addresses the automation of monolith-to-microservices migration across three key phases: *service identification*, *service packaging*, and *service deployment*. As shown in Figure 0.2, each contribution targets one of these phases where existing research lacks automation and tool support. We describe the core contributions below and are supported by peer-reviewed and–or submitted publications.

Systematic Literature Review on ML-based Migration Approaches.

We conducted a comprehensive Systematic Literature Review (SLR) of 81 primary studies (2015–2024) investigating the use of machine learning (ML) in monolith-to-microservices migration. The SLR analyses five key dimensions: (1) targeted migration phases, (2) input types and granularity, (3) ML algorithms employed, (4) evaluation practices, and (5) persistent challenges. It reveals a clear imbalance in research efforts, with the majority focusing on service identification while leaving packaging and deployment significantly underexplored. This SLR serves as the empirical foundation of this thesis, motivating the development of our three-phase framework.

- *A Systematic Literature Review of Machine Learning Approaches for Migrating Monolithic Systems to Microservices. Accepted to IEEE Transactions on Software Engineering, 2025.*

MicroMiner: An ML-based Approach for Service Identification.

To address the limitations of heuristic and rule-based decomposition methods, we propose *MicroMiner*, a type-based service identification approach that combines machine learning classification with semantic clustering. *MicroMiner* identifies service types (entity, application, and utility) using a supervised ML classifier and clusters them into microservices based on both structural and semantic relationships. The resulting services adhere to microservices design principles such as bounded context and single responsibility. We validate *MicroMiner* on four real-world monolithic systems, showing superior performance over existing tools in terms of precision, modularity, and architectural coherence.

Although during this thesis we had other contributions to the identification of microservices, as reflected in the publications listed below, we focus exclusively on *MicroMiner* as the representative and most mature contribution of this part of our research. The other studies

explore complementary techniques (e.g., method-level analysis and graph-based models), but are beyond the scope of the framework proposed in this thesis.

- *Magnet: Method-based Approach Using Graph Neural Network for Microservices Identification. IEEE 21st International Conference on Software Architecture (ICSA), 2024.*
- *MicroMatic: Fully Automated Microservices Identification Approach from Monolithic Systems. ACM/IEEE 6th International Workshop on Software Engineering Research & Practices for the Internet of Things, 2024.*
- *From Legacy to Microservices: A Type-based Approach for Microservices Identification Using ML and Semantic Analysis. Journal of Software: Evolution and Process, 2023.*

***MicroPacker*: An LLM-guided Framework for Microservice Packaging.**

We introduce *MicroPacker*, a systematic approach that leverages Large Language Models to automate code refactoring, API generation, and pattern integration during the packaging phase. *MicroPacker* decouples dependencies among services, builds REST/gRPC interfaces, and integrates runtime patterns, such as service registries and configuration servers. The approach relies on structured prompt generation followed by validation-driven refinement. We apply *MicroPacker* to two monolithic systems and report a significant reduction in manual effort, with the LLMs generating mostly correct and complete artifacts.

- *MicroPacker: A Framework for Microservices Packaging using LLMs. Under Review, 2025.*
- *Exploring the Systematic Use of LLMs for Microservices Generation. AI-PA, ICSOC 2024.*

MiDKo: A RAG-based Framework for Deployment Automation.

To address the limitations of template-based deployment tools, we propose *MiDKo*, an approach that automates the generation of Dockerfiles and Kubernetes manifests using Retrieval-Augmented Generation (RAG). *MiDKo* extracts topic-based representations from a large corpus of existing deployment artifacts. Retrieved artifacts guide LLMs in generating configurations tailored

to the specific requirements of each microservice. We validate the generated artifacts for syntax, deployability, and expert feedback, demonstrating MiDKo adaptability and reliability in generating Dockerfiles and Kubernetes manifests.

- *MiDKo: Automated Dockerfile and Kubernetes Manifests Generation for Microservices Deployment. Under Review, 2025.*

0.3.2 Related Contributions

These publications address microservices-related problems.

- *BOAM: A Business-oriented Identification Approach of Microservices within Legacy Systems. 22nd International Conference on Service-Oriented Computing (ICSOC), 2024.*
- *On the Maintenance Support for Microservice-based Systems Through the Specification and Detection of Microservice Antipatterns. Journal of Systems and Software, 2023.*

0.3.3 Other Contributions

These publications result from collaborations outside the scope of this thesis.

- *TGenAI: LLM-based Approach for Functional Test Case Generation for IoT Systems. Under Revision in Journal of Systems and Software, 2025.*

0.4 Thesis Outline

To ensure clarity and avoid ambiguity, we define the structural terms used throughout this thesis as follows:

- **Migration process**: the overall process from monolithic system to MSA.
- **Phase**: a high-level division of the migration process, such as *identification*, *packaging*, or *deployment*.

- **Task and Subtask**: specific operational activities carried out within each phase, representing progressively finer levels of granularity.
- **Framework**: the overarching AI-driven solution that includes all approaches for automating monolith-to-microservices migration.
- **Approach**: the concrete solution developed to address the challenges of a given phase.
- **Stage and Step**: the sequential elements that compose each approach, detailing its logical flow and execution steps.

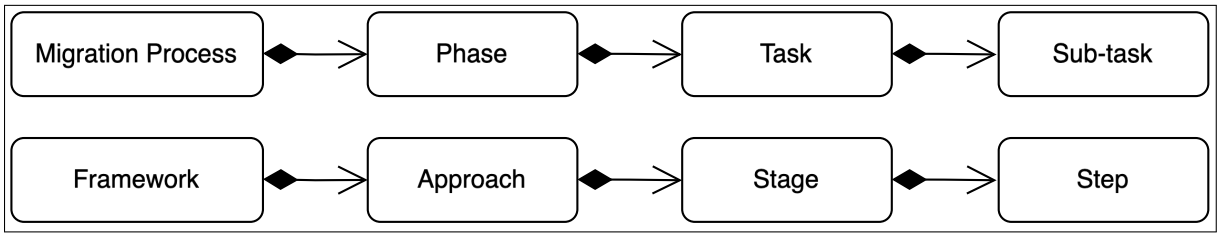

Figure 0.3 Hierarchical organisation of key terms used in this thesis

Figure 0.3 illustrates the hierarchical relationships among these terms.

This thesis is structured into five chapters. Chapter 1 presents a Systematic Literature Review on the use of machine learning techniques for monolith-to-microservices migration. It analyses 81 primary studies to identify key trends, evaluate existing approaches, and highlight automation gaps across the service identification, packaging, and deployment phases. These findings motivate the development of the unified, automation-oriented migration framework proposed in this thesis. Chapter 2 presents the **first approach** of the proposed migration framework, targeting the **identification phase**. It introduces *MicroMiner*, a machine learning-based technique that leverages classification and semantic clustering to extract architecturally coherent, loosely coupled candidate microservices from monolithic systems. Chapter 3 presents the **second approach** of the framework, focusing on the **packaging phase**. It introduces *MicroPacker*, a systematic method that uses Large Language Models to automate source code restructuring, API generation, and the integration of essential microservices design patterns such as service

registries and configuration servers. Chapter 4 presents the **third approach** of the framework, addressing the **deployment phase**. It details *MiDKo*, a deployment automation framework that uses retrieval-augmented generation to produce context-aware Dockerfiles and Kubernetes manifests. Finally, Chapter 5 concludes the thesis by summarising the core contributions, discussing their implications for automated software migration, and outlining directions for future research.

CHAPTER 1

LITERATURE REVIEW

1.1 Introduction

Many approaches have been proposed to address the various phases of migrating monolithic systems to microservices, such as heuristic-based approaches and domain-driven approaches. these approaches often require manual intervention and lack adaptability across different systems (Abdellatif *et al.*, 2021b). However, recent advancements in machine learning offer a promising opportunity to tackle these challenges, including identifying service boundaries, analysing interdependencies, and predicting failure during monitoring. ML techniques excel at processing large and complex datasets, uncovering patterns, and supporting decision-making processes that are otherwise difficult to achieve with traditional approaches. For example, ML can assist in automating repetitive and error-prone tasks, such as clustering related components or optimising deployment strategies, potentially reducing migration time and improving accuracy and consistency (Toumi, Bagaa & Ksentini, 2023).

Building on the motivation and challenges outlined in the introduction, this chapter presents a comprehensive review of existing research on monolith-to-microservices migration Ml-based approaches. Through a systematic literature review, we analyse the state-of-the-art in machine learning techniques in the migration process. We study the role of ML in this domain by addressing key research questions, such as the phases of migration it automates, the characteristics of the input data it leverages, the types of ML approaches applied, how these approaches are evaluated, and the challenges faced by researchers. By answering these questions, this study seeks to provide a comprehensive understanding of the current state of ML-based migration techniques and identify opportunities for future research.

While several studies have explored specific aspects of microservice migration and the associated challenges, to the best of our knowledge, no prior work investigated systematically existing ML

approaches for this migration to understand the automated phases, used inputs, applied ML techniques, followed evaluation processes, and encountered challenges.

To address this objective, we define our primary research question as: *"How are machine learning techniques used in the migration of monolithic systems to microservices?"* From this, we derive the following specific research questions:

❶ **RQ1**: Which migration phases are automated by ML?
❷ **RQ2**: How are inputs used by ML migration approaches?
❸ **RQ3**: What ML approaches do researchers apply?
❹ **RQ4**: How do researchers evaluate ML approaches?
❺ **RQ5**: What challenges arise in using ML approaches?

To answer these research questions, we followed the updated Preferred Reporting Items for Systematic Review and Meta-Analysis (PRISMA) statement for reporting systematic reviews (Page *et al.*, 2021a). We screened 2,301 potentially relevant studies from eight digital libraries, considering publications between 2015 and 2024. Using inclusion and exclusion criteria along with snowball, we assessed the quality of the primary studies based on their design, methodology, analysis, and conclusions. We retained a total of 81 Primary studies (PSs) for analysis.

The primary contribution of this chapter is a comprehensive understanding of how ML can be used to support the migration of monolithic systems to microservices. This is achieved through the following key RQs:

1. A systematic analysis of the migration phases automated using ML techniques, identifying gaps in automation across the migration lifecycle (RQ1).
2. An organised synthesis of the types of inputs used in ML-driven migration, including their granularity, sources and how it is preprocessing, forming a basis for understanding the input usage in migration (RQ2).
3. A detailed analysis of machine learning techniques employed for migration, highlighting commonly used models, emerging techniques, and their features (RQ3).

4. An exploration of evaluation practices for ML-based migration approaches, identifying common metrics, benchmarking, and success criteria (RQ4).
5. A discussion of the challenges encountered when applying ML for migration, including scalability, data availability, and the interpretability of models, along with directions for future research (RQ5).

We organise the remainder of this chapter as follows. Section 1.2 presents other SLRs on monolithic to microservices migration. Section 1.3 describes the SLR methodology used and provides an overview of the analysis process. Section 1.4 focuses on phases of migration that are currently automated using ML. Section 1.5 examines the types of data collected and processed as inputs for ML during migration. Section 1.6 identifies the ML techniques applied in this context. Section 1.7 discusses how ML-based approaches are evaluated for migration. Section 1.8 investigates the challenges and limitations of applying ML to migration. Section 1.9 discusses the observations. Section 1.10 outlines the potential threats to the validity of our study. Finally, Section 1.11 summarises the findings.

1.2 Other SLRs

The migration of monolithic applications to microservices has gained significant attention in recent years due to its potential to improve scalability, flexibility, and maintainability in software systems. Several studies have explored various aspects of this migration, ranging from decomposition frameworks to the challenges faced during the migration. This section presents an overview of the most relevant works, focusing on decomposition frameworks, migration strategies, and the challenges associated with microservice adoption.

Abgaz *et al.* (2023a) systematically reviewed the decomposition of monolithic applications into microservices, introducing the Monolith to Microservices Decomposition Framework (M2MDF). This framework identifies the major phases and key elements involved in the decomposition process. Furthermore, the study analyses existing methods, tools, and metrics used for evaluating the decomposition process and proposes future research directions, particularly in refining

decomposition techniques. In contrast, our study focuses on the use of ML for the migration from monoliths to microservices, whereas Abgaz *et al.* (2023a) focused on decomposition only and did not consider the use of ML for migration or its automation. Fritzsch *et al.* (2019c) also classified and compared refactoring approaches used to decompose monoliths, highlighting the strategies used and the challenges faced during migration. However, their study did not explore the use of machine learning (ML) for migration, nor did it address migration phases, inputs, ML approaches used, or evaluations of ML approaches.

Several studies have explored the challenges and limitations of the migration process.
Velepucha & Flores (2021) addressed quality attributes (QAs) commonly considered during migration, identifying improvement tactics from 72 reviewed studies. In contrast, our study emphasises leveraging ML to automate migration, focusing on the phases, inputs, ML techniques, and evaluation strategies. Ponce, Márquez & Astudillo (2019) categorised migration techniques based on their level of automation and validation focus, analysing the challenges encountered across different system types. Despite their insights, they did not account for automation or ML-driven migration. The findings of Velepucha & Flores (2021) and Ponce *et al.* (2019) align with those of Razzaq & Ghayyur (2023), who examined factors contributing to successful industry migration, highlighting key organisational challenges. However, their study did not explore ML-based migration strategies or automation. Bushong *et al.* (2021) analysed microservices from an evolutionary perspective, classifying methods and techniques used for microservice analysis. They identified overlaps between traditional software analysis and microservice-specific approaches. Nevertheless, their study did not investigate ML-driven migration strategies or address evaluation metrics and input requirements. Similarly, Saucedo *et al.* (2024) proposed a migration process based on real-world cases, providing a catalogue of tools, techniques, and influencing factors. Their work offers valuable insights into migration challenges and benefits.

Other studies have provided insights into migration strategies and lessons learned.
Silva Filho & Figueiredo Carneiro (2019) reviewed strategies for migrating legacy systems to microservices and discussed the challenges encountered. However, they did not examine ML-driven automation. Capuano & Muccini (2022) emphasised the role of quality attributes in

guiding migration decisions but did not explore aspects covered in our study. Similarly, Kalske, Mäkitalo & Mikkonen (2018) identified key motivations and challenges faced by companies transitioning to microservices, yet their study did not investigate the role of ML in migration automation. Both Capuano & Muccini (2022) and Kalske *et al.* (2018) emphasised the importance of strategic planning and phased migration approaches. Mparmpoutis & Kakarontzas (2022) explored data-driven artifacts, such as database schemas, for identifying microservices during migration, demonstrating how existing data can inform the re-architecting process. Nonetheless, their study did not address ML-driven migration or automation. Kazanavičius & Mažeika (2019) examined the technical challenges and benefits of migrating legacy systems, assessing various migration methods and their drawbacks. However, they did not consider inputs, ML techniques, or evaluation criteria.

Di Francesco, Lago & Malavolta (2018) conducted an empirical study on migration practices, identifying the activities performed and challenges faced by practitioners throughout the migration process but did not consider the use of ML techniques or its automation. Furthermore, this study did not consider ML-based approaches, nor did it address key factors such as migration inputs and evaluation metrics. Fritzsch *et al.* (2019b) also explored the intentions and strategies behind microservice migration, investigating the technical and organisational challenges encountered during this transition. However, their study did not examine the use of machine learning (ML) techniques or the automation of the migration process, nor did it address evaluation metrics or inputs required for such approaches.

1.3 Research Method

We followed the updated PRISMA guidelines (Page *et al.*, 2021b,c) and Kitchenham, Madeyski & Budgen (2022) guidelines to review and report our findings. We used three main phases: planning, conducting, and reporting the review. During the planning phase, we defined the objective of SLR and reviewed the protocol. The objective of this SLR is defined in Section 1.1. This section defines the review protocol for conducting the SLR. It consists of six steps:

① defining the research questions,

② formulating the search query,
③ selecting the studies,
④ snowballing,
⑤ assessing the quality of the studies, and
⑥ extracting and analysing the data.

In the following subsections, we explain each step of our review protocol.

1.3.1 Research Questions

This study answers the following RQs:

RQ1: Which migration phases are automated by ML?

Rationale: By analysing the phases and the tasks supported by ML, we want to understand the role and impact of ML in the migration process. This investigation provides insights into how ML is integrated into current migration approaches.

RQ2: How are inputs used by ML migration approaches?

Rationale: By categorising inputs based on their types, granularities, sources, and preprocessing methods, we want to create a structured classification of the inputs and understand how inputs are gathered, prepared, and utilised in machine learning models for microservices migration.

RQ3: What ML approaches do researchers apply?

Rationale: By investigating the ML approaches in the migration, we want to (1) identify and classify the ML models used, (2) examine their learning paradigms, and (3) analyse the features leveraged to enhance model performance. This RQ provides a comprehensive understanding of how ML contributes to different phases of the migration.

RQ4: How do researchers evaluate ML approaches?

Rationale: By investigating how ML-based approaches for microservices migration are evaluated, we want to (1) identify the evaluation metrics used to assess the quality and performance of these approaches, (2) explore the benchmarks employed for comparative analyses, (3) understand the success criteria that define their effectiveness, and (4) analyse the tools available for implementing and evaluating these approaches.

RQ5: What challenges arise in using ML approaches?

Rationale: By examining the limitations, we aim to identify and analyse the key challenges associated with applying machine learning techniques in the migration from monolithic architectures to microservices.

1.3.2 Search Query

We formulated our search query by applying the PICO (Population, Intervention, Comparison, Outcome) framework (Cooke, Smith & Booth, 2012). We followed the following steps:

❶ Obtaining the main terms from our main research question as stated in the introduction.
❷ Identifying the possible synonyms of the main terms.
❸ Applying the Boolean OR to combine possible synonyms of the main terms.
❹ Applying the Boolean AND to combine expressions in the previous step.

As a result of PICO framework, we formulated the following search query:

> (Monolith* OR Exist* OR Legac*) AND (Microservice* OR Micro-service* OR MSA) AND (Migrat* OR Identif* OR Decompos* OR Extract* OR Transform* OR Refactor* OR Transit* OR Creat* OR Genera*) AND (Machine learning OR ML OR Neural Network* OR *coder OR *supervised OR Reinforcement Learning OR Model*)

To obtain more comprehensive results, we used the asterisk (*) in search queries as a wildcard to match any sequence of characters.

1.3.3 Studies Selection

We applied the PRISMA steps to select the PSs. The main steps include database identification, removal of duplicates, screening, eligibility assessment, multiple rounds of both backward and forward snowballing, and quality assessment of each study.

1.3.3.1 Databases Identification

We selected seven online digital libraries: ACM Digital Library, Compendex, IEEE Xplore, ScienceDirect, SpringerLink, Scopus, Web of Science, and Wiley. These libraries are widely used for literature reviews in software engineering, as recommended by Dyba, Dingsoyr & Hanssen (2007).

We applied our search query to each of these digital libraries. However, some libraries impose restrictions when performing queries. For instance, ScienceDirect limits queries to a maximum of eight connectors, while the ACM Digital Library does not allow wildcards. We adjusted the search query to meet the specific requirements of each library. Our search was confined to English-language, peer-reviewed scholarly articles published in journals, conferences, and workshops between 2015 and 2024. This time frame was chosen due to the increase in microservices-related literature starting in 2015 (Vural, Koyuncu & Guney, 2017). We initially retrieved a total of 2301 studies from seven libraries.

1.3.3.2 Duplicates Removal

We merged the results from the digital libraries and manually removed duplicate studies. This reduced the total from 2301 studies to 1219,.

1.3.3.3 Studies Screening

We defined inclusion and exclusion criteria and applied them to select relevant primary studies while excluding irrelevant ones.

Inclusion Criteria: We considered the following inclusion criteria for PSs selection:

- **IC1:**The study is written in English.
- **IC2:**The study is published between 2015 and 2024.
- **IC3:**The study is published in journals, conferences, or workshops.
- **IC4:**The study has at least 4 pages.
- **IC5:**The study focuses on the migration of monolith applications into microservices.
- **IC6:**The study uses a machine/deep learning algorithm.
- **IC7:**The study provides enough information to answer at least three research questions (RQs).
- **IC8:**The study has its full text available online.
- **IC9:**The study provides sufficient migration details.
- **IC10:**The study uses an automated or semi-automated migration approach.

Exclusion Criteria: We considered the following exclusion criteria:

- **EC1:**The study is not primary.
- **EC2:**The study has not been peer-reviewed.
- **EC3:**The study is a graduate thesis or project report.
- **EC4:**The study's full text is not available online.
- **EC5:**The study does not provide enough details.
- **EC6:**The study does not provide an automated or semi-automated migration approach.

We applied our inclusion and exclusion criteria in two steps, first using the titles and abstracts, then the full texts. Criteria IC1-IC6 and EC1-EC3 were applied to the titles and abstracts, while IC7-IC10 and EC4-EC6 were applied to the full texts. During the initial screening, we assessed the titles, abstracts, and page counts, and determined if the studies qualified as primary. Two authors independently conducted the screening using predefined inclusion and exclusion criteria. To ensure consistency, we compared the screening results for 50 randomly selected studies using Cohen's Kappa (Cohen, 1968)

We calculated Cohen's Kappa, achieving near-perfect agreement (k=0.85), highlighting the consistency of our screening. For the remainder of the process, authors met regularly to review results and resolved disagreements through discussion and consensus. After completing the initial screening, we selected 68 studies for full-text review as potential PSs.

1.3.3.4 Eligibility Assessment

In the second round of screening, two authors independently applied IC7-IC10 to 68 studies by thoroughly reviewing them. To ensure a shared understanding of the inclusion and exclusion criteria, we compared the results of 15 randomly selected studies using Cohen's Kappa. We calculated a near-perfect agreement (k=0.93), demonstrating strong consistency between the two authors. This allowed us to confidently proceed with the eligibility assessment for the remaining studies. Ultimately, we identified 54 PSs that met the eligibility criteria.

1.3.4 Snowballing

We conducted four rounds of snowballing to identify additional PSs. In Round 1, we reviewed references and citations of 54 initial PSs, identifying 1,347 potential studies and adding 19 PSs, bringing the total to 73. In Round 2, we repeated the process for the 19 new PSs, identifying 602 potential studies and adding 11 PSs, increasing the total to 84. In Round 3, we examined the 11 new PSs, identifying 190 potential studies and adding 3 PSs, raising the total to 87. Finally, in Round 4, we reviewed the 3 new PSs, identifying 43 potential studies and adding 2 PSs, resulting in a final total of 89 PSs.

1.3.5 Quality Assessment

Following the guidelines proposed by Li *et al.* (2021c) and Dybå & Dingsøyr (2008), we developed a quality assessment checklist comprising eight questions (or quality criteria) to evaluate the quality of our primary studies (PSs). Table 1.1 shows the quality criteria we used in this study.

Table 1.1 Quality Criteria

No	Quality Criteria
Q1	Is there a clear statement of the aims of the research? Consider: Is there a rationale for why the study was undertaken?
Q2	Was the research design appropriate to address the aims of the research? Consider: Did the researcher justify the design of the research?
Q3	Was the research method implemented in a way that addressed the research issue? Consider: Has the researcher discussed the process or the details of the methods that were chosen/proposed?
Q4	Is there a clear statement of findings? Consider: Has an adequate discussion/evaluation of the evidence identified or method proposed, both for and against the researchers' arguments, been demonstrated?
Q5	Has the limitation or future work been considered adequately? Consider: Did the researcher examine the limitations and future work?
Q6	Is there an adequate description of the context in which the research was carried out? Consider: Did the researcher explain how the context influenced the study, and was this context relevant to the aim of the research?
Q7	Was the data collected in a way that addressed the research issue? Consider: Did the researcher collect data from relevant sources, and was the data collection process clearly described and ethically sound?
Q8	Was the data analysis sufficiently rigorous? Consider: Did the researcher provide a thorough and unbiased examination of the data, and were the limitations of the analysis acknowledged?

Responses to each question were: "Yes" (1 point), "Partially" (0.5 points), or "No" (0 points). Three authors independently applied this checklist to each study, discussed any discrepancies, and achieved consensus on the scores. We calculated the quality score for each study by summing the individual scores and converting this total into a percentage. Studies achieving an 80% score or higher were retained. Using this criterion, we excluded eight studies falling below the threshold and included 81 PSs in our review.

1.3.6 Data Extraction and Analysis

We analysed and extracted the necessary data from each primary study (PS) to address our research questions (RQ1-RQ5). Table 1.2 presents the key data items extracted, including

the item name, a brief description, and the corresponding research question it addresses. For simplicity, only three data items are displayed. The full details of the extracted data are available in our publicly accessible replication package.

Table 1.2 Data Extraction Template

Data Item	Short Description	RQs
Code	A unique identifier assigned to the paper	
Title	The title of the research paper	
Year	The year in which the paper was published	
Venue	The conference, journal, or workshop where the study was published	
Authors	The individuals who contributed to conducting the study	
Focus	The main topic of the study	
Class	The category of the study	
Migration Phase	The specific phase of system migration from monolith to microservices	RQ1
Automated Task	Exact task automated during the migration phase	RQ1
Input type	Define the types of input used, such as source code, logs, UML diagrams, etc.	RQ2
The granularity of Data	The level of detail or precision in the data such as files, classes, lines of code, or logs.	RQ2
Source of Data	Indicates the data source: open source, industrial projects, user-generated, or collected.	RQ2
Data preprocessing	Steps to prepare data, including collection, cleaning, normalisation, and transformation.	RQ2
ML Technique	Indicates the machine learning techniques used.	RQ3
Learning Approach	Indicates the learning approach: supervised, semi-supervised, unsupervised, etc.	RQ3
Feature Selection	Specifies the features used for model training.	RQ3
Evaluation Metrics	Specifies the metrics used to evaluate the output.	RQ4
Benchmark	Indicates benchmarks or comparisons with other systems.	RQ4
Success Criteria	Specifies success criteria, such as improved recall, quality, or performance.	RQ4
Tool Availability	Indicates tool availability: open-source, commercially available, or a proof of concept (PoC).	RQ4
Challenges	Identifies the challenges discussed in the paper.	RQ5

1.4 Migration Phases Automated by ML (RQ1)

The migration of legacy systems to microservices involves a sequence of complex phases, many of which are now supported or automated by machine learning. In this section, we analyse the phases of the migration process and identify the specific tasks that ML automates in this process. The migration process from legacy systems to microservices can be divided into five phases: Pre-migration, Identification, Packaging, Deployment, and Monitoring. The identification phase is the most frequently covered in our primary studies (PSs), with 48% (n=39), followed by the monitoring phase at 35% (n=28). The deployment phase is also notable, covered by 15% of PSs (n=12), while the pre-migration phase is the least discussed, with only 2.24% (n=2).

1.4.1 Pre-migration

The pre-migration phase focuses on evaluating the current approaches, methods, and tools used in the legacy system. This phase also involves planning the migration by identifying suitable strategies, defining objectives, and outlining the steps needed to ensure a smooth transition. Researchers have leveraged ML to automate tasks such as designing microservices, which assists in planning microservice architecture by analysing legacy systems (Nakazawa, Ueda, Enoki & Horii, 2018), and predicting success rates of migration, which uses predictive models to evaluate potential migration outcomes (Alshammari, Almadhor, Qasem, Alkhateeb & Amjad, 2023).

1.4.2 Identification

The identification phase defines the boundaries of prospective microservices. This involves detecting functional modules, mapping dependencies, and clustering components with business requirements. ML techniques have been applied to automate tasks such as boundary identification, which detects functional boundaries within monolithic systems
(Gysel, Kölbener, Giersche & Zimmermann, 2016b); clustering, which groups related components or services using ML-based clustering algorithms (Kamimura, Yano, Hatano & Matsuo,

2018); and microservices identification, which automates the process of determining candidate microservices (Zhong, Teng, Ma, Chen & Yu, 2023; Daoud *et al.*, 2021; Saidi, Tissaoui, Benslimane & Faiz, 2022; Daoud *et al.*, 2020b; Faria & Silva, 2023; Sun, Boranbaev, Han, Wang & Yu, 2022; Trabelsi *et al.*, 2023; Trabelsi, Moha, Gueheneuc & Geffard, 2024a; Trabelsi, Popa, Pereyrol, Beaulieu & Moha, 2024b).

1.4.3 Packaging

The packaging phase encapsulates the identified components into functional microservices. This phase includes defining service interfaces, managing dependencies, and generating missing components. However, no studies in our review proposed the use of ML for this phase, highlighting a significant gap in the research landscape.

1.4.4 Deployment

The deployment phase focuses on deploying the created microservices into a target environment. This involves setting up orchestration tools, configuring infrastructure, and integrating with existing systems. ML techniques are increasingly used to automate tasks such as automated deployment, which automates the deployment of microservices into production environments (Trabelsi *et al.*, 2024b; Lv *et al.*, 2022); resource management, which ensures efficient use of infrastructure resources during deployment (Khan *et al.*, 2023); (Joseph, Martin, Chandrasekaran & Kandasamy, 2019; Luan & Shen, 2024; Gan *et al.*, 2019); resource allocation, which dynamically allocates resources to microservices based on workload demands (Khan *et al.*, 2023; Shafi, Abdullah, Iqbal, Erradi & Bukhari, 2024; Song *et al.*, 2023; Gan, Liang, Dev, Lo & Delimitrou, 2022; Luan & Shen, 2024; Yang, Nguyen, Jin & Nahrstedt, 2019; Zeng, Wang, Li, Wu & Zhang, 2023); microservice autoscaling, which adjusts the scale of microservices to match performance requirements (Tong, Meng, Song, Pan & Yu, 2023); microservice orchestration, which manages dependencies and interactions between microservices (Li, Tan, Wang, Li & Luo, 2022); microservice placement, which decides optimal placements for microservices within the infrastructure (Ray, Banerjee & Narendra, 2023; Joseph *et al.*,

2019); and resource estimation, which estimates the resource needs of microservices to optimize deployment strategies (Chow, Deshpande, Seshadri & Liu, 2022).

1.4.5 Monitoring

The monitoring phase ensures the reliability and efficiency of the microservices-based system post-migration. This phase involves continuous performance tracking, anomaly detection, and dynamic resource management to meet changing workload demands. ML models enhance the monitoring process by enabling proactive detection of issues and optimising resource usage. Tasks automated in this phase include anomaly detection, which identifies abnormal behaviors or performance issues in microservices ((Shi, Li, Liu, Chang & Li, 2022; Zhang *et al.*, 2022a; Chen *et al.*, 2023c))performance analysis, which evaluates the performance metrics of deployed microservices (Song *et al.*, 2024); detection of failure types, which classifies failure types to facilitate troubleshooting (Xu *et al.*, 2023; Sun *et al.*, 2024); fault diagnosis, which diagnoses root causes of failures to enable faster resolution (Song *et al.*, 2024); root cause analysis, which pinpoints the underlying issues causing anomalies or failures (Gan *et al.*, 2022; Gan, Liang, Dev, Lo & Delimitrou, 2021); privacy risks detection, which identifies potential privacy concerns in microservice interactions (Chou, Al-Masri, Kanzhelev & Fattah, 2021); and sanity checks, which verifies the correctness of system states and responses (Chow *et al.*, 2022).

1.5 Inputs used by ML migration approaches(RQ2)

Understanding the types of inputs collected and processed in the migration process is essential for identifying how machine learning techniques are applied. We categorised inputs by types, granularity, sources, and preprocessing methods.

1.5.1 Input Types

In this section, we classify the input types identified in the primary studies (PSs) into five distinct groups, which illustrate their use in the migration phases. Our analysis reveals that runtime

artifacts are the most commonly used inputs, appearing in 60% of the PSs (n = 49), emphasising their critical role in understanding system performance during migration. Executable software models, such as source code and configuration files, are the second most frequently used, accounting for 37% of PSs (n = 30). In contrast, domain artifacts and model artifacts, each identified in 8% of PSs (n = 7), provide valuable high-level system descriptions, while technical artifacts are the least common, found in only 2% of PSs (n = 2).

1.5.1.1 Domain Artifacts

Domain artifacts capture high-level information about the system's business and functional requirements, aiding in the understanding and identification of microservices. These include API documentation, which represents details about the available APIs and their interactions. For example, Sun *et al.* (2022) highlighted the use of the OpenAPI specification of the legacy system's RESTful APIs to identify microservices. Quality of Service (QoS) constraints represent parameters that must be maintained during migration. The authors of (Chou *et al.*, 2021) used service communication traces to meet predefined QoS requirements, such as availability and reliability. Architecture recommendations, as provided by Selmadji *et al.* (2020), offer expert suggestions for migration strategies and component separation. Functional descriptions, such as those provided by Morais, Bork & Adda (2021), represent high-level descriptions of system functionalities, capturing technological attributes and relationships between services.

1.5.1.2 Runtime Artifacts

Runtime artifacts are derived from the system's runtime behavior and are crucial for understanding performance and operational characteristics. These include resource metrics, which focus on information about resource consumption, such as memory and CPU usage. The study by Chow *et al.* (2022) explored the use of API traffic logs to analyse the runtime behavior of systems. Similarly, Xu *et al.* (2023) leveraged system metric data, such as CPU, memory, and disk usage, along with failure propagation information, to identify resource bottlenecks. Du, Xie & He (2018) used memory utilisation, network latency, and packet loss to detect

anomalies in containerised microservices, while Shafi *et al.* (2024) used CPU usage metrics to analyse performance across different stages of system operation. Monitoring metrics are another key runtime artifact. Kong, Li, Ge, Zhang & Li (2024) introduced SpanGraph, a tool that leverages monitoring metrics, trace logs, and configuration files to construct directed graphs representing microservice interactions. Performance metrics, such as response time, throughput, and latency, are also critical. Santos, Sampaio, Rosa & Cavalcanti (2024) used time series data of performance metrics to predict system performance. Trace logs provide detailed insights into the execution flow of the system, as demonstrated by Chen *et al.* (2023a), who used trace logs and performance metrics for anomaly detection. Workloads, which describe system usage scenarios and patterns, are used by Abdullah, Iqbal, Erradi & Bukhari (2019b) to predict the response time of microservices.

1.5.1.3 Model Artifacts

Model artifacts describe the system from an architectural or design perspective, often providing an abstract representation. These include business processes, which show workflows and processes that the system supports. For example, Daoud *et al.* (2020a) leveraged business process models (BPMN) to identify and define microservices within the monolithic system, aligning system components with organisational workflows. UML diagrams, which represent details of the system design, are also commonly used. For instance, Zhang *et al.* (2022a) combined code, logs, and UML diagrams to provide unique insights for anomaly analysis, leveraging the visual representation of system design to identify issues. User stories are another key model artifacts, used by Vera-Rivera, Puerto, Astudillo & Gaona (2021) to guide the organisation of microservice development tasks.

1.5.1.4 Executable Software Models

Executable software models pertain to the actual software and its configuration, offering insights into the system's implementation. For example, Trabelsi *et al.* (2024b) used source code to identify microservices, while Nakazawa *et al.* (2018) analysed source code files from a monolithic

application to generate a calling-context tree for better system understanding and transformation. Configuration files are used by Kong *et al.* (2024) alongside trace logs and monitoring metrics to construct SpanGraph, a directed graph representing microservice interactions for fault localisation. Data files are used by Song *et al.* (2023) to support latency-aware provisioning.

1.5.1.5 Technical Artifacts

They provide supporting information necessary for system operation. These include server information, which details the infrastructure supporting the system. For example, Ray *et al.* (2023) used service radius and capacity, as well as runtime traces to predict microservice placement.

1.5.2 Input Granularity

The granularity of input data, ranging from high-level abstractions like system architecture diagrams to fine-grained elements such as lines of code or individual function calls, plays a critical role in determining the precision and scope of machine learning tasks. System-level granularity is the most frequently used (51%, n=41), as seen in Lv *et al.* (2022) and Lv *et al.* (2024), which monitor and analyse system components and interactions. Code-level granularity, focusing on classes and methods, is the second most common (32%, n=26), used for tasks like refactoring and microservice identification, as in Rathod, Joseph & Martin (2023) and Trabelsi *et al.* (2024a). Process-level granularity (9%, n=7) examines workflows, with Al-Debagy & Martinek (2019) using API operation-level data for decomposition and Daoud *et al.* (2020b) using activity-level granularity for workflow modeling. Data-level granularity, the least explored (2%, n=2), focuses on stored information, such as table-level data for microservice clustering (Romani, Tibermacine & Tibermacine (2022)) and entity-level representations for service boundaries (Gysel *et al.* (2016b)). Application-level granularity (7%, n=6) examines higher-level abstractions, with Bajaj, Bharti, Gupta, Gupta & Yadav (2024) using use case-level data for workflow modeling and Zhang *et al.* (2022a) analysing application files for

anomaly detection. URI and API-level granularity, as in Abdullah, Iqbal & Erradi (2019a) and Al-Debagy & Martinek (2019), support system decomposition and service identification.

1.5.3 Input Sources

Our analysis shows that open-source datasets are the most widely used, representing 62% of PSs (n=50), showcasing the reliance on publicly available repositories for experimentation and validation. For example, Cao & Zhang (2022) validated their approach on on open-source applications like JPetStore. Real-world data, which provides insights from operational systems, is used in 35% of PSs (n=28). For instance, Li, Du, Chang, Mukherjee & Eide (2021b) used the AIOps Challenge dataset to detect anomalies and faults in microservice systems. Meanwhile, synthetic data, generated to simulate real-world conditions, is utilised in 14% of PSs (n=11). For example, Gan *et al.* (2021) combined synthetic data from Apache Thrift with open-source datasets like DeathStarBench to debug performance in microservice environments, while Alshammari *et al.* (2023) collected data using a survey and expert interviews to optimise migration strategies.

1.5.4 Preprocessing Tasks

Preprocessing ensures the quality and usability of input data for machine learning models. Code analysis is the most frequently employed technique (58%, $n = 47$), underscoring the importance of understanding system structure and semantics. Data extraction, used in 48% of PSs ($n = 39$), captures essential operational and dependency information, as in Faria & Silva (2023), which used JavaParser to extract methods and class dependencies, and Chen, Guang, Wang & Yan (2023b), which parsed execution traces to build graph representations. Data cleaning, employed in 30% of PSs ($n = 24$), improves data quality through noise reduction and normalisation. For example, Chen *et al.* (2023a) applied density-based clustering to filter noisy data points, while Chow *et al.* (2022) normalised trace data for compatibility with further analysis. Dependency modeling, found in 26% of PSs ($n = 21$), structures relationships for better analysis. For instance, Liu, Lu, Zhang, Zhang & Wang (2020a) used vectorisation and embedding for clustering, and

Sooksatra, Maharjan & Cerny (2022) built dependency graphs using normalised matrices for analysis with Variational Autoencoders (VAE). Modeling and task structuring, as in Lv *et al.* (2024), represented microservice dependencies using adjacency matrices, while Luan & Shen (2024) captured features like CPU usage and request counts for workload analysis.

1.6 ML approaches applied by researchers (RQ3)

In this section, we discuss the key ML models used in microservices migration, their learning paradigms, and the features that enhance model performance.

1.6.1 Models

Different models have been applied to address various phases of the migration process. We categorised these models into classical machine learning, deep learning, graph-based models, and reinforcement learning. Classical ML models were the most frequently used (37.04%, n = 30), followed by graph-based models (28.40%, n = 23), reinforcement learning (23.46%, n = 19), and deep learning (15 PSs).

1.6.1.1 Classical Machine Learning

Classical machine learning techniques leverage structured data and mathematical models to identify patterns, make predictions, or optimise solutions. These techniques are widely applied to automate tasks such as service identification, dependency analysis, and performance prediction. Classification techniques, such as Support Vector Machines Trabelsi *et al.* (2023), Random Forest Daoud *et al.* (2021), and Naive Bayes Al-Debagy & Martinek (2021a), are used to categorise components and analyse dependencies. Regression techniques, including Linear Regression Al-Debagy & Martinek (2021a) and Support Vector Regression Zhong *et al.* (2023), predict performance metrics and resource usage. Clustering methods, such as K-means Al-Debagy & Martinek (2021a) and Density-Based Spatial Clustering of Applications with Noise

Sellami, Saied & Ouni (2022b), group components for service decomposition. Search-based techniques, like Genetic Algorithms Alshammari *et al.* (2023), optimise migration strategies.

1.6.1.2 Deep Learning

Deep learning techniques leverage neural networks to analyse complex data structures. Autoencoders and Variational Autoencoders (Sooksatra *et al.*, 2022) are used for dimensionality reduction and feature learning. Recurrent networks, such as Long Short-Term Memory (Li *et al.*, 2021b) and Gated Recurrent Units (Zhang *et al.*, 2022a), process sequential data for anomaly detection. Transformers, including Bidirectional Encoder Representations from Transformers (Chen *et al.*, 2023c) and CodeBERT (Faria & Silva, 2023), excel in natural language and code analysis tasks.

1.6.1.3 Graph-Based Models

Graph-based methods model component relationships as graphs, enabling service interaction and dependency analysis. Graph Convolutional Networks (Mathai, Bandyopadhyay, Desai & Tamilselvam, 2022) and Graph Attention Networks (Trabelsi *et al.*, 2024a) aggregate features from neighbouring nodes to improve graph representations. Variational Graph Autoencoders (Sooksatra *et al.*, 2022) combine graph convolutional networks with variational inference for tasks like link prediction.

1.6.1.4 Reinforcement Learning

Reinforcement learning learns optimal strategies through interactions with the environment. Fuzzy Q-Learning (Joseph *et al.*, 2019) used to handles uncertainty in resource allocation, while Deep Q-Learning (Abdullah *et al.*, 2019b) and Deep Deterministic Policy Gradient (Lv *et al.*, 2024) are used to optimise autoscaling and resource usage. Multi-Agent Deep Deterministic Policy Gradient (Ray *et al.*, 2023) improves service placement in edge computing environments.

1.6.2 Learning paradigm

The learning paradigms applied in the studies include unsupervised learning, which is the most popular (67.9%, $n = 55$), used for tasks like microservice extraction, clustering, and anomaly detection (Trabelsi *et al.*, 2023; Shahini & Momeni, 2024); reinforcement learning (23.46%, $n = 19$), primarily for resource provisioning and deployment optimisation (Song *et al.*, 2023; Lv *et al.*, 2024); supervised learning (20.99%, $n = 17$), applied in anomaly detection and resource optimisation (Tan & Li, 2024; Abdullah *et al.*, 2019b); semi-supervised learning (4.94%, $n = 4$), which combines labelled and unlabeled data to refine service boundaries (Al-Debagy & Martinek, 2021a); and self-supervised learning (2.47%, $n = 2$), which generates pseudo-labels for tasks like failure localisation and resource provisioning (Sun *et al.*, 2024; Zeng *et al.*, 2023).

1.6.3 Selected Features

Machine learning models in these PSs rely on diverse feature types to analyse, predict, and optimise different aspects of the migration process. Structural features are the most popular (62.96%, $n = 51$), including class dependencies (Nitin, Asthana, Ray & Krishna, 2022), method calls (Kamimura *et al.*, 2018), data dependencies (Romani *et al.*, 2022), transactional dependencies (Daoud *et al.*, 2021), and call graph dependencies (Qian *et al.*, 2023). Behavioral features (43.21%, $n = 35$) capture runtime interactions, such as invocation paths (Kamimura *et al.*, 2018), log events (Shahini & Momeni, 2024), contextual log entries (Shahini & Momeni, 2024), and temporal patterns (Shahini & Momeni, 2024). Semantic features (37.04%, $n = 30$) focus on the meaning of system components, leveraging semantic embeddings (Trabelsi *et al.*, 2023), function names (Saidi, Tissaoui & Faiz, 2023), and API descriptions (Kamimura *et al.*, 2018). Performance features (29.63%, $n = 24$) measure system efficiency, including CPU usage (Bajaj *et al.*, 2024), memory usage (Sooksatra *et al.*, 2022), network traffic (Liu & Zhang, 2024), response times (Bajaj *et al.*, 2024), and availability (Nunes, Santos & Rito Silva, 2019). These features collectively enable tasks such as microservice identification, dependency analysis, anomaly detection, and performance optimisation.

1.7 Evaluation of ML approaches (RQ4)

In this section, we begin by discussing the various evaluation metrics commonly employed to assess the performance of machine learning models in migration scenarios. Next, we explore the conducted comparative analyses. Following this, we outline the success criteria adopted in the literature. Finally, we will discuss the tools availability of those works .

1.7.1 Evaluation Metrics

Evaluation metrics are categorised into five key areas: classification and prediction metrics (55.56%, n=45), software design metrics (28.40%, n=23), system behavior metrics (22.22%, n=18), clustering metrics (14.81%, n=12), and developer-centric metrics (3.70%, n=3). Each category addresses specific aspects of microservices migration.

1.7.1.1 Classification and Prediction Metrics

These metrics assess model performance in labeling and prediction tasks, primarily during the identification phase. Precision measures the accuracy of positive predictions, with variations like Precision@K (PR@K) and Mean Average Precision (MAP) used for ranked predictions or multiple queries. It is commonly applied in monitoring for anomaly detection, as seen in (Du *et al.*, 2018; Shahini & Momeni, 2024; Chen *et al.*, 2023c; Shi *et al.*, 2022). Recall evaluates the model's ability to identify all relevant instances, particularly in anomaly detection and fault identification tasks, as demonstrated in (Du *et al.*, 2018; Al-Debagy & Martinek, 2021a). The F1-Score balances precision and recall, making it critical for monitoring and service identification tasks, as shown in (Chen *et al.*, 2023b; Du *et al.*, 2018; Bajaj, Bharti, Goel & Gupta, 2020; Shahini & Momeni, 2024; Daoud *et al.*, 2021). Accuracy represents the overall correctness of predictions, with variations like Top-K Accuracy used in multi-class classification tasks, as seen in (Zhong *et al.*, 2023; Tan & Li, 2024; Nakazawa *et al.*, 2018; Tong *et al.*, 2023). The Area Under the ROC Curve (AUC) evaluates binary classification models, particularly in anomaly detection, as highlighted in (Daoud *et al.*, 2020a; Kalia *et al.*,

2020). Finally, the Matthews Correlation Coefficient (MCC) provides a balanced measure of classification quality, especially in imbalanced datasets, as used in (Shahini & Momeni, 2024).

1.7.1.2 Clustering Metrics

These metrics evaluate the quality of service clustering during the identification phase. The Dunn Index measures the separation between clusters, as demonstrated in (Zhong *et al.*, 2023; Sellami *et al.*, 2022b; Bajaj *et al.*, 2020; Chen *et al.*, 2023b). The Silhouette Score assesses the similarity of objects within their cluster compared to others, as seen in (Sellami *et al.*, 2022b; Tong *et al.*, 2023; Daoud *et al.*, 2020a; Chou *et al.*, 2021). Newman-Girvan Modularity evaluates the strength of network division into clusters, as highlighted in (Saidi *et al.*, 2023; Zhong *et al.*, 2023; Chen *et al.*, 2023b; Tan & Li, 2024). Non-Extreme Distribution ensures balanced cluster sizes (Mathai *et al.*, 2022; Qian *et al.*, 2023; Rathod *et al.*, 2023; Sooksatra *et al.*, 2022). Maximum Cluster Size limits the largest cluster size to avoid granularity issues (Nunes *et al.*, 2019). The Number of Singleton Clusters counts clusters with a single element to prevent excessive fragmentation (Nunes *et al.*, 2019).

1.7.1.3 System Behavior Metrics

These metrics assess operational performance, resource utilisation, and scalability during and after migration. Response Time measures the time taken for the system to respond to a request, as highlighted in (Al-Debagy & Martinek, 2021a; Tan & Li, 2024; Bajaj *et al.*, 2020; Du *et al.*, 2018). Resource Utilisation evaluates the effectiveness of resource use, as seen in (Sellami *et al.*, 2022b; Tong *et al.*, 2023; Saidi *et al.*, 2023; Chou *et al.*, 2021). Energy Consumption assesses the total energy required for operations, as demonstrated in (Daoud *et al.*, 2021; Tan & Li, 2024; Bajaj *et al.*, 2020; Nakazawa *et al.*, 2018). The SLA Violation Rate measures the percentage of time a system fails to meet its Service Level Agreement obligations, as shown in (Saidi *et al.*, 2023; Sellami *et al.*, 2022b; Daoud *et al.*, 2020a; Tong *et al.*, 2023). Scalability evaluates the system's capacity to maintain or improve performance under increased workload, as in (Daoud *et al.*, 2021; Zhong *et al.*, 2023; Chen *et al.*, 2023b; Chou *et al.*, 2021).

1.7.1.4 Software Design Metrics

These metrics evaluate software quality during the evaluation, focusing on maintainability and modularity. Cohesion measures the relatedness of functionalities within a service, as demonstrated in (Daoud *et al.*, 2021; Sellami *et al.*, 2022b; Bajaj *et al.*, 2020; Chou *et al.*, 2021). Coupling assesses dependencies between services, as seen in (Al-Debagy & Martinek, 2021a; Zhong *et al.*, 2023; Saidi *et al.*, 2023; Tan & Li, 2024). The Granularity Metric evaluates service size and scope, as highlighted in (Sellami *et al.*, 2022b; Tong *et al.*, 2023; Chen *et al.*, 2023b; Nakazawa *et al.*, 2018). Structural Modularity measures the degree of system decomposition into independent components, as shown in (Zhong *et al.*, 2023; Daoud *et al.*, 2020a; Bajaj *et al.*, 2020; Du *et al.*, 2018). Cognitive Complexity assesses code understandability for developers, as demonstrated in (Saidi *et al.*, 2023; Sellami *et al.*, 2022b; Nakazawa *et al.*, 2018; Tan & Li, 2024).

1.7.1.5 Developer-Centric Metrics

These metrics evaluate alignment with business requirements and developer feedback during the validation. Developer Validation assesses qualitative insights from developers, as seen in (Daoud *et al.*, 2021; Sellami *et al.*, 2022b; Bajaj *et al.*, 2020; Nakazawa *et al.*, 2018). Closeness to Manual Expert Analysis compares automated classifications to expert evaluations, as highlighted in (Sellami *et al.*, 2022b; Daoud *et al.*, 2020a; Chen *et al.*, 2023b; Tong *et al.*, 2023).

1.7.2 Benchmarking

The reviewed PSs demonstrate a strong emphasis on benchmarking to evaluate and compare the effectiveness of their approaches. Direct method comparisons are the most commonly conducted, appearing in 51.85% of the PSs (n=42), while evolutionary method comparisons are less frequent, accounting for only 7.41% of the PSs (n=6).

1.7.2.1 Direct Comparisons

This category includes methodologies that assess the performance of ML models against existing approaches and techniques. Most PSs utilised comparisons with the State of the Art to benchmark current techniques against the latest advancements in the field, as well as against traditional rule-based and heuristic methods. For example, (Abdullah *et al.*, 2019b; Liu & Zhang, 2024; Al-Debagy & Martinek, 2021a; Bajaj *et al.*, 2024; Chen *et al.*, 2023b; Trabelsi *et al.*, 2023, 2024a) highlight comparisons across ML-based approaches and traditional techniques. Additionally, some PSs focused on the effectiveness of widely used algorithms and techniques, such as clustering algorithms like K-Means, DBSCAN, and their variations. These were explored in studies like (Cai, Han, Su & Wang, 2021; Chen *et al.*, 2022; Dehghani, Kolahdouz-Rahimi, Tisi & Tamzalit, 2022).

1.7.2.2 Evolutionary Comparisons

This category emphasises the development of new methodologies built upon previous research. A few PSs aimed at enhancing existing Research by improving established techniques through the integration of novel features or adaptations for specific microservices migration scenarios. For instance, (Daoud *et al.*, 2020b; Mathai *et al.*, 2022) demonstrated enhancements to clustering and service identification methodologies. Other PSs explored combinatory approaches, combining different methodologies to leverage their strengths and improve migration outcomes. Examples include hybrid methods that integrate clustering with deep learning models for better service decomposition, as seen in (Chen *et al.*, 2023b; Chow *et al.*, 2022).

1.7.3 Success Criteria

The success of ML-based migration approaches is determined by various criteria. Technical performance is the most emphasised, appearing in 55.56% of the PSs (n=45), with key indicators including improvements in precision, recall, and resource optimisation. Structural quality is considered in 35.80% of the PSs (n=29), focusing on modularity and cohesion. Outcome

alignment with standards and baselines is assessed in 29.63% of the PSs (n=24). Functional and domain-specific success is addressed in 25.93% of the PSs (n=21), while system effectiveness and adaptability are evaluated in only 17.28% of the PSs (n=14).

1.7.3.1 Technical Performance Metrics

These metrics focus on the operational effectiveness of the ML models and their impact on system performance. Key indicators include increased precision, recall, and F1-Score, which demonstrate the effectiveness of the proposed ML-based approach in performing specific tasks, as shown in (Al-Debagy & Martinek, 2019; Bajaj *et al.*, 2024; Song *et al.*, 2024; Shi *et al.*, 2022). Reduced response times and latency indicate improved system performance, leading to better user experiences and satisfaction, as highlighted in (Li *et al.*, 2022; Lv *et al.*, 2022; Nitin *et al.*, 2022). Enhanced efficiency, reflecting the ability of the migration process to operate with limited resources while achieving desired outcomes, is another critical indicator, as seen in (Cao & Zhang, 2022; Joseph *et al.*, 2019). Increased throughput, which measures the system's capacity to handle requests, reflects the effectiveness of the ML approach in optimising service delivery, as demonstrated in (Zhong *et al.*, 2023; Li *et al.*, 2021b).

1.7.3.2 Outcome Alignment with Standards and Baselines

This criterion evaluates how well the outcomes of the ML-driven migration align with established standards and benchmarks. Achieving a defined baseline of performance metrics is crucial for evaluating the success of the migration, as improvements over this baseline indicate the effective use of ML methodologies. For example, (Sun *et al.*, 2022; Kamimura *et al.*, 2018; Gysel *et al.*, 2016b) demonstrated surpassing baseline metrics as a success criterion. Additionally, developer architectural alignment ensures that the architecture of the migrated services aligns with developer expectations, with positive feedback from developers serving as a key indicator of success, as seen in (Nunes *et al.*, 2019; Morais *et al.*, 2021).

1.7.3.3 Structural Quality

This category focuses on the integrity and quality of the microservices architecture. Increased modularity ensures that microservices are independently deployable and maintainable, allowing for easier updates and scaling, as noted by (Daoud *et al.*, 2021; Desai, Bandyopadhyay & Tamilselvam, 2021). Increased cohesion assesses the relatedness of functionalities within a service, as highlighted in (Mathai *et al.*, 2022; Vera-Rivera *et al.*, 2021). Decreased coupling measures the reduction in dependencies between services (Cao & Zhang, 2022; Trabelsi *et al.*, 2024a).

1.7.3.4 System Effectiveness and Adaptability

This criterion evaluates how well the system performs and its ability to adapt to changing requirements. Increased scalability measures the system's ability to scale resources up or down as needed to accommodate varying workloads, as emphasised in (Lv *et al.*, 2024; Santos *et al.*, 2024). Increased resource utilisation aims to maximise the effectiveness of resource use during migration, as shown in (Zhang *et al.*, 2022a; Tan & Li, 2024). Increased automation evaluates the extent to which processes are automated, reducing manual intervention and enhancing efficiency, as highlighted in (Trabelsi *et al.*, 2024b).

1.7.3.5 Functional and Domain-Specific Success

This criterion evaluates how well ML models meet domain-specific requirements and objectives. Increased anomaly detection quality enhances the reliability of the migration process and operational stability, as demonstrated in (Shahini & Momeni, 2024; Chen *et al.*, 2023c). Reduced QoS violations indicate that the migrated services are performing within acceptable parameters, as seen in (Gan *et al.*, 2022; Abdullah *et al.*, 2019b). Increased cost efficiency reflects the ability to achieve desired outcomes with minimal financial expenditure, as highlighted in (Shafi *et al.*, 2024; Abdullah *et al.*, 2019a).

1.7.4 Tool Types and Availability

Research prototypes are the most commonly reported tool type, appearing in 27.18% of the PSs (n=22), while only 2.47% (n=2) describe production-grade applications. Regarding availability, 69.14% of the PSs (n=56) did not provide information on tool availability, whereas 30% (n=24) described open-source tools. Proprietary tools are mentioned in one PS.

1.7.4.1 Tool Types

Research prototypes, developed within research projects for validation purposes, are often partial implementations. Examples include tools proposed in (Al-Debagy & Martinek, 2021b; Bajaj *et al.*, 2020; Desai *et al.*, 2021; Gan *et al.*, 2019; Nitin *et al.*, 2022). In contrast, production-grade applications are fully developed, robust, and scalable tools ready for real-world deployment, as highlighted in (Gysel *et al.*, 2016b).

1.7.4.2 Tool Availability

Open-source tools, made publicly available through platforms like GitHub, enable reuse and collaboration, as seen in (Cao & Zhang, 2022; Desai *et al.*, 2021; Nitin *et al.*, 2022; Trabelsi *et al.*, 2024a). Proprietary tools, such as those discussed in (Chow *et al.*, 2022), have restricted access and are not publicly available. However, the majority of PSs (69.14%) provide no information on tool availability, limiting reproducibility and reuse.

1.8 Challenges in using ML approaches (RQ5)

Machine learning has shown potential in supporting various phases of microservices migration process. However, several challenges limit its effectiveness and broader adoption. These challenges, derived from the analysis of primary studies (PSs), range from technical complexities to data quality issues, scalability concerns, integration difficulties, and specialised skill requirements. This section discusses these challenges and explores possible solutions.

1.8.1 Technical Complexity

One of the primary challenges in applying ML to software migration is handling the variability and complexity of legacy systems. Identifying service boundaries within monolithic architectures remains a difficult task, often leading to suboptimal service decomposition that affects system performance, scalability, and maintainability (Nunes *et al.*, 2019; Faria & Silva, 2023; Rathod *et al.*, 2023). Nunes *et al.* (2019) emphasise the need to focus on transactional contexts rather than structural dependencies when decomposing monolithic systems. Their approach aims to preserve business logic while enabling effective service extraction. Similarly, Rathod *et al.* (2023) propose using Relational Topic Modeling (RTM) to combine structural dependencies with semantic analysis, ensuring that refactoring operations improve design quality.

Potential Solution: A hybrid approach combining ML techniques with domain-driven design and expert knowledge can refine results and ensure practical applicability. Increasing model explainability can also help software architects validate and refine service decomposition outcomes.

1.8.2 Data Quality and Availability

ML models rely on diverse datasets, including source code repositories, execution traces, logs, and performance metrics. However, data unavailability, inconsistency, incompleteness, and noise significantly impact the accuracy and reliability of ML-based approaches (Zhang *et al.*, 2022a; Daoud *et al.*, 2020b; Du *et al.*, 2018; Kalia *et al.*, 2020; Nunes *et al.*, 2019; Saidi *et al.*, 2022; Bajaj *et al.*, 2024; Sooksatra *et al.*, 2022; Liu *et al.*, 2020a; Liang, Li & Peng, 2024; Shahini & Momeni, 2024; Zhang, Zhang, Peng & Sha, 2022b; Li, Tang, Wen *et al.*, 2021a; Gan *et al.*, 2019; Mathai *et al.*, 2022; Song *et al.*, 2024; Chen *et al.*, 2023a; Al-Debagy & Martinek, 2019; Chen *et al.*, 2023b; Xu *et al.*, 2023; Rathod *et al.*, 2023; Yang *et al.*, 2019; Vera-Rivera *et al.*, 2021; Trabelsi *et al.*, 2024b; Shi *et al.*, 2022; Qian *et al.*, 2023; Wang, Li, Qi, Lu & Wu, 2024; Trabelsi *et al.*, 2024a; Romani *et al.*, 2022; Trabelsi *et al.*, 2023). Mathai *et al.* (2022) highlight the dependency of decomposition techniques on external artifacts, such as runtime traces and

commit histories, which are often incomplete or unavailable. Similarly, Bajaj *et al.* (2024) stress that inconsistent SDLC artifacts, including use case models and functional requirements, pose challenges in greenfield developments.

Potential Solution: Robust data preprocessing pipelines, data augmentation techniques, and validation mechanisms can mitigate inconsistencies and improve reliability. Establishing standardised datasets for benchmarking ML models can also facilitate the development of more robust approaches.

1.8.3 Scalability Concerns

Scalability becomes a critical issue when ML models are applied to large and complex monolithic systems. These models often experience increased computational overhead, leading to performance bottlenecks and delays in the migration process (Kamimura *et al.*, 2018; Zhang *et al.*, 2022a; Chen *et al.*, 2022; Sooksatra *et al.*, 2022; Al-Debagy & Martinek, 2021b; Liu *et al.*, 2020a; Sun *et al.*, 2024; Li *et al.*, 2021a; Gan *et al.*, 2019; Cao & Zhang, 2022; Luan & Shen, 2024; Rathod *et al.*, 2023; Song *et al.*, 2023; Vera-Rivera *et al.*, 2021; Lv *et al.*, 2024; Bajaj *et al.*, 2020; Santos *et al.*, 2024; Shafi *et al.*, 2024; Daoud *et al.*, 2021; Chou *et al.*, 2021; Ray *et al.*, 2023; Tong *et al.*, 2023; Li *et al.*, 2022; Zeng *et al.*, 2023; Song *et al.*, 2024). (Tong *et al.*, 2023) discuss the inefficiencies of centralised ML strategies in edge-cloud environments, where distributed architectures complicate data collection and synchronisation. These challenges highlight the need for efficient processing strategies to handle large-scale migration tasks.

Potential Solution: Scalable ML frameworks, distributed computing, and incremental learning approaches can enhance efficiency. Optimised data vectorisation and clustering techniques can also reduce computational costs.

1.8.4 Integration Complexities

Integrating ML into migration workflows is complex due to the diversity of tools, platforms, and architectures involved (Nunes *et al.*, 2019; Yang *et al.*, 2019; Lv *et al.*, 2024; Ray *et al.*, 2023). (Lv

et al., 2024) introduce a Graph Convolutional Network (GCN) combined with deep reinforcement learning (DRL) to optimise microservice deployment. While this approach enhances decision-making, its integration into existing migration frameworks remains challenging. (Ray *et al.*, 2023) explore ML-driven microservice placement strategies in edge computing, emphasising the need for adaptive, automated decision-making processes.

Potential Solution: Developing modular ML components with standardised APIs can improve interoperability. Using containerization and microservices for ML components simplifies deployment and integration with existing architectures.

1.8.5 Specialised Skills and Resource Requirements

ML-based migration requires expertise in both machine learning and software engineering, which many development teams lack (Cai *et al.* (2021); Abdullah *et al.* (2019b); Zhang *et al.* (2022b); Li *et al.* (2021a); Gan *et al.* (2019); Li *et al.* (2021b); Gan *et al.* (2022); Khan *et al.* (2023); Mathai *et al.* (2022); Song *et al.* (2023); Yang *et al.* (2019); Lv *et al.* (2024); Dehghani *et al.* (2022); Trabelsi *et al.* (2024b); Santos *et al.* (2024). Santos *et al.* (2024))highlights the complexity of implementing ensemble learning models for performance forecasting, while (Dehghani *et al.*, 2022) emphasises the difficulty of applying AI-based migration techniques due to their reliance on reinforcement learning and neural networks.

Potential Solution: Investing in ML training programs and fostering collaboration between software engineers and ML experts can bridge skill gaps. Cross-disciplinary teams can facilitate more effective ML adoption in migration projects.

1.9 Discussion

Figure 1.1 presents a high-level classification derived from our answers to the research questions, summarising the key findings discussed in the previous sections. Building on this classification, we further analyse the four core dimensions addressed by our study, including Phases, Input Types, ML Approaches, and Evaluation Methods. In addition to examining each dimension

independently, we conduct a cross-dimension analysis to reveal how these aspects co-occur across the selected primary studies. We also identify several research gaps, both from what is missing in the literature and from patterns of underexplored combinations across dimensions. These gaps are discussed in detail later in this section and serve as the foundation for the contributions proposed in this thesis.

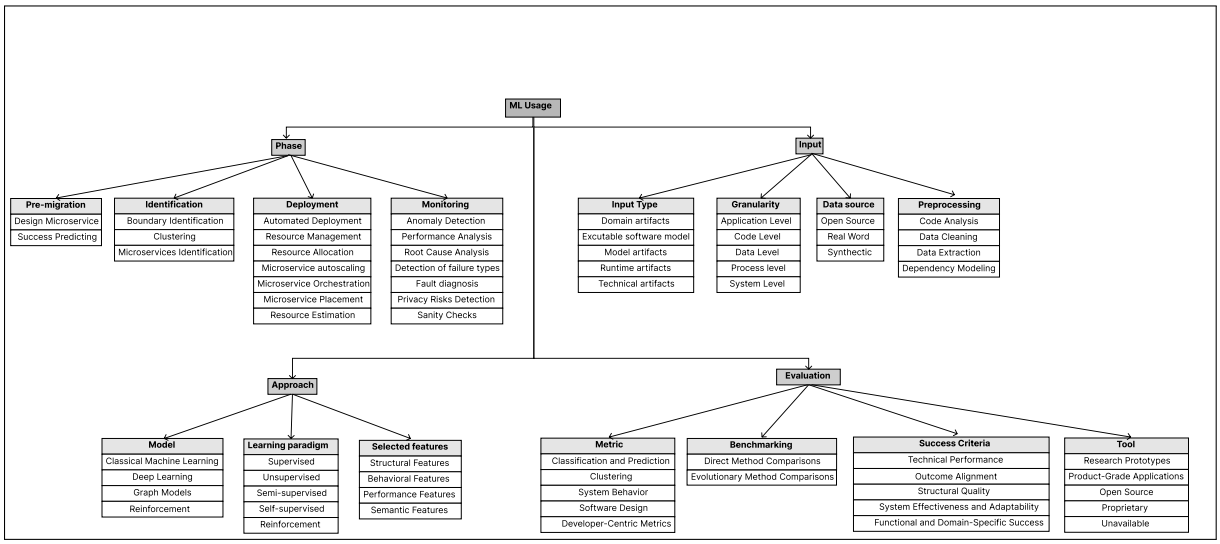

Figure 1.1 ML Usage in Migration: A Classification Overview

1.9.1 Phases

Our findings highlight that Identification is the phase in which ML approaches have been the most applied by researchers, with 39 PSs addressing automating this phase. It is followed by Monitoring (28 PSs) and Deployment (12 PSs), reflecting the research community's emphasis on defining microservices, monitoring their functionality, and ensuring their deployment into production environments. The pre-migration is perceived as straightforward or reliant on domain-specific knowledge, which limits its appeal to academic exploration. Also, the variability in business logic and technical landscapes across domains makes it challenging to generalise findings, further discouraging research. Additionally, we found no PSs proposing an ML-based approach to assist the Packaging phase. This phase remains largely overlooked, despite relying on extensive code generation and refactoring—both of which are still predominantly manual

and time-consuming. Earlier machine learning approaches struggled to automate this phase effectively. However, the advent of Large Language Models (LLMs) presents new opportunities to address these challenges. LLMs have the potential to automate substantial aspects of the code generation process, such as API generation, offering solutions that are not only automated but also more consistent and accurate (Hou *et al.*, 2023). This shift paves the way for future research exploring the integration of LLMs into the Packaging phase, enabling groundbreaking approaches that could significantly enhance automation in this crucial phase.

1.9.2 Input

The effectiveness of any machine learning-based approach, including those for microservices migration, depends on the quality, diversity, and preparation of its input data. The classification categorises inputs into five main types: domain artefacts (e.g., API documentation), Executable software model (e.g., source code), model artifacts (e.g., use cases), runtime artefacts (e.g., resource metrics, logs), and technical artifacts (e.g., Servers information). Among these, runtime artifacts are the most frequently utilised, appearing in 49 PSs, because they are crucial for capturing dynamic system behaviours, making them particularly valuable in monitoring and deployment phases. In contrast, domain and model artifacts are predominantly used during the identification and pre-migration phases.

Data sources vary, with most PSs relying on open-source data (50 PSs), due to their accessibility. Real-world data sources (28 PSs), and synthetic data sources (11 PSs) provide additional data, though their use is limited by accessibility and confidentiality challenges. The limited use of real-world and industrial systems justifies the gap in using academic findings on practical, large-scale applications.

Preprocessing steps, including data extraction (39 PSs), data cleaning (24 PSs), code analysis (47 PSs), and dependency modelling (21 PSs), are used for preparing data for machine learning models. These steps ensure that raw data is transformed into a structured and meaningful format, enabling models to effectively analyse system components, dependencies, and relationships.

While these preprocessing techniques have been explored in various studies, challenges such as handling incomplete or noisy data, and automating complex code analysis remain critical areas for further research.

1.9.3 Approaches

The PSs employ a diverse range of ML models that we categorised into classical machine learning, deep learning, graph-based methods, and reinforcement learning. Among these, classical machine learning is the most prominent, appearing in 30 PSs, with clustering, classification, and regression techniques commonly used in the PSs. Deep learning techniques, including autoencoders, transformers, and recurrent neural networks, are utilised in 15 PSs to model complex relationships and patterns in data. These models are applied in the monitoring phase to predict failure root causes and in deployment to optimise resource allocation. Graph-based methods, used in 23 PSs, focus on capturing structural dependencies among code components or log traces, while reinforcement learning is applied in 19 PSs, primarily for the deployment and monitoring phases. Supervised learning is specially used in the monitoring phase since labelled data is available. Unsupervised learning techniques are mostly used in 55 PSs where labelled data is mostly unavailable. In the identification phase, clustering algorithms, such as DBSCAN and K-means are employed in 6 PSs to identify microservices in monolithic systems, evaluated based on metrics such as cohesion and coupling. Reinforcement learning is used in deployment and monitoring phases for adaptive resource allocation and anomaly prediction.

ML models in these PSs leverage a variety of feature types to enhance different phases of the migration process. Performance features (24 PSs), such as CPU usage, memory consumption, and response time, are commonly used for performance prediction, anomaly detection, and deployment optimisation. Structural Features (51 PSs) enable ML models to learn structural and relational patterns, facilitating service decomposition, impact analysis, and dependency resolution. Semantic features (30 PSs), such as function names, API descriptions, and business logic embeddings, support domain-aware clustering and automated service identification by capturing functional similarities. Behavioural features (35 PSs), including invocation paths,

transactional dependencies, and execution traces, provide insights into runtime behaviour, helping ML models understand component interactions and operational patterns. While these feature types enhance model effectiveness, challenges remain in automating feature extraction and handling noisy dependencies.

1.9.4 Evaluation

The evaluation of microservices migration approaches relies on a combination of metrics, benchmarking, and success criteria. The analysis of evaluation metrics in the reviewed PSs highlights a strong emphasis on classification and prediction metrics (45 PSs), which are predominantly used to assess the accuracy and effectiveness of ML models in microservices identification and anomaly detection. Software design metrics (23 PSs) follow, focusing on cohesion, coupling, and modularity to ensure maintainability and architectural quality, exclusively used in the identification phase. System behaviour metrics (18 PSs), such as response time and resource utilisation, are widely used to evaluate the performance and scalability of migrated microservices. Clustering metrics (12 PSs) provide insights into the quality of service decomposition, ensuring well-formed and meaningful microservices groupings. In contrast, developer-centric metrics (3 PSs) remain underexplored, despite their importance in validating the practicality and usability of automated migration approaches.

Benchmarking is a key validation method, with direct method comparisons being the most common (42 PSs), where approaches are evaluated against state-of-the-art techniques. Evolutionary method comparisons, which assess how approaches improve with new features, are significantly less frequent (6 PSs). Expanding evolutionary benchmarking could offer deeper insights into model robustness, scalability, and long-term adaptability.

The success of ML-based migration approaches is primarily measured by improving technical performance (45 PSs), like precision, recall, and resource optimisation. Structural quality (29 PSs), including modularity and cohesion, is also a key consideration, while alignment with standards (24 PSs) remains underexplored. Functional success (21 PSs) and system adaptability

(14 PSs) receive the least attention, highlighting gaps in assessing scalability, automation, and long-term viability. Tool availability is a major issue, with 56 PSs lacking accessible implementations, raising concerns about reproducibility and benchmarking. While open-source tools (24 PSs) promote transparency, the lack of standardised evaluation frameworks hinders industry adoption. Future research should emphasise long-term adaptability, compliance with industry standards, and practical tool accessibility to bridge the gap between research and real-world application.

1.9.5 Cross-Dimensions Analysis

While each research question focused on a specific dimension, we extended our analysis with a cross-dimensions examination to explore how migration phases, input types, ML models, and evaluation metrics co-occur across the selected primary studies. To support this analysis, we constructed a *Sankey Diagram* (Figure 1.2) that visualises complete migration paths derived from the selected primary studies. Each stream in the diagram corresponds to a unique combination of four key dimensions: Migration Phase → Input Type → ML Model → Evaluation Metric. The width of each stream is proportional to the number of PSs following that path, while the color indicates the originating migration phase. An interactive version of this diagram is available online at this link[1], where hover tooltips display the full path along with the corresponding number of supporting PSs.

To structure our cross-dimensions analysis, we first identified the most frequent migration paths across all reviewed studies. These represent combinations of migration phase, input type, ML technique, and evaluation metric that occur most often. Table 1.3 presents the prominent paths reported in five or more primary studies.

This analysis highlights some dominant paths in how machine learning is applied across microservices migration phases. The most frequent migration paths emerge in the monitoring

[1] An interactive version of this figure is available at: https://imen-trabelsi.github.io/SLR-MS-Migration/Cross-RQs-Interactive-Diagram.html. Hovering over a stream reveals the complete path and the associated number of PSs.

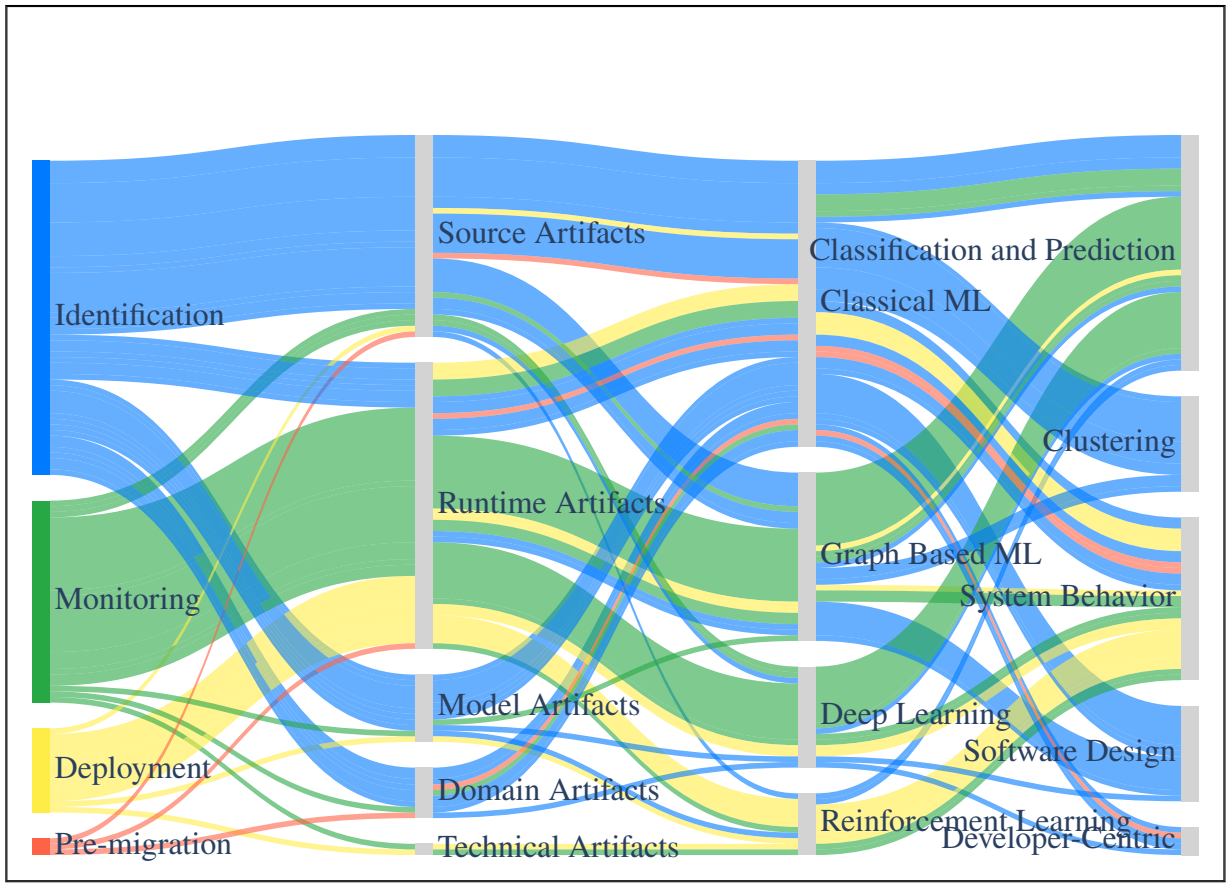

Figure 1.2 Cross-RQ analysis of ML-based migration approaches. Each stream represents a path linking Migration Phase, Input Type, ML Technique, and Evaluation Metric.

Table 1.3 Most Frequent Migration Paths across Phase, Input, ML Technique, and Evaluation

Phase	**Input Type**	**ML Technique**	**Evaluation Metric**	**#PSs**
Monitoring	Runtime Artifacts	Graph Based ML	Classification and Prediction	13
Monitoring	Runtime Artifacts	Deep Learning	Classification and Prediction	10
Identification	Source Artifacts	Classical ML	Software Design	7
Identification	Source Artifacts	Classical ML	Clustering	7
Identification	Source Artifacts	Graph Based ML	Software Design	6
Deployment	Runtime Artifacts	Reinforcement Learning	System Behavior	5

phase. These studies typically rely on runtime artifacts, such as logs, execution traces, and system-level metrics, and apply either graph-based ML or deep learning. Graph-based ML is used in 13 primary studies (Xu *et al.* (2023); Chen *et al.* (2023c, 2022); Kong *et al.* (2024); Shi

et al. (2022); Gan *et al.* (2022); Huang, Yang, Yu, Li & Zheng (2023); Liang *et al.* (2024); Chen *et al.* (2023a); Gan *et al.* (2021); Sun *et al.* (2024); Zhang *et al.* (2022b,a)). Deep learning is used in 10 PSs (Ding *et al.* (2024); Luan & Shen (2024); Gan *et al.* (2019); Li *et al.* (2021b); Shahini & Momeni (2024); Song *et al.* (2024); Wang *et al.* (2024); Tan & Li (2024); Liu *et al.* (2020b); Cai *et al.* (2021)). Both configurations are primarily evaluated using classification and prediction metrics, such as precision, recall, F1-score, and AUC. These studies address anomaly detection, fault prediction, or behavioral monitoring in production environments.

In the identification phase, the dominant input remains source artifacts, including source code, call graphs, and code embeddings. Classical machine learning techniques such as K-Means, DBSCAN, and Agglomerative Clustering are applied in conjunction with either software design metrics 7 PSs (Trabelsi *et al.* (2023); Nitin *et al.* (2022); Al-Debagy & Martinek (2021a); Saidi *et al.* (2023); Sellami *et al.* (2022b); Cao & Zhang (2022); Faria & Silva (2023)) apply classification models to source code, or clustering to model-level inputs such as use-case diagrams and class diagrams.

In the deployment phase, most approaches rely on runtime artifacts and leverage reinforcement learning for dynamic optimization, resource allocation, or autoscaling. These appear in 5 studies (Lv *et al.* (2024); Yang *et al.* (2019); Song *et al.* (2023); Ray *et al.* (2023); Lv *et al.* (2022)), where the evaluation relies on system behavior metrics such as response time, memory consumption, or CPU utilization. A few additional studies (e.g., Joseph *et al.* (2019)) use classical ML or deep learning, but they are less common in this phase.

Several underexplored paths also emerged. These include configurations where domain-level artifacts (e.g., API documentation) are combined with classical ML and evaluated with system behavious metrics (e.g., Gysel *et al.* (2016b)) or cases where runtime input is introduced in the identification phase to guide service boundary prediction through behavioral insight (e.g., Morais *et al.* (2021), Abdullah *et al.* (2019a)).

Beyond full migration paths, we examined pairwise relationships between individual dimensions to surface finer-grained patterns. For instance, input types vary significantly across phases.

Source artifacts dominate the identification phase in 26 PSs, where they support architectural decomposition and service extraction (e.g., Trabelsi *et al.* (2023); Desai *et al.* (2021); Zhong *et al.* (2023); Nitin *et al.* (2022); Rathod *et al.* (2023)). Runtime artifacts are prevalent in both monitoring and deployment, used to inform operational decisions or detect abnormal behaviors (Chou *et al.*, 2021; Ding *et al.*, 2024; Li *et al.*, 2021a). Domain and model artifacts are primarily used in pre-migration or identification, reflecting their importance in early planning (Alshammari *et al.*, 2023; Daoud *et al.*, 2020a). Configuration artifacts appear almost exclusively in deployment studies that target environmental tuning or resource scheduling (Ray *et al.*, 2023; Yang *et al.*, 2019).

Similarly, ML technique choices differ markedly across phases. Classical ML is most prominent in the identification phase, used for clustering and classification (e.g., Trabelsi *et al.* (2023); Bajaj *et al.* (2020); Morais *et al.* (2021); Abdullah *et al.* (2019a)). Graph-based models also used for the identification phase, particularly when structural relations are central to the decomposition logic (Trabelsi *et al.*, 2024a; Nunes *et al.*, 2019). Approaches for monitoring, on the other hand, favor deep learning techniques that can model temporal or sequential patterns (e.g.,Chen *et al.* (2023a); Gan *et al.* (2021); Zhang *et al.* (2022b); Tan & Li (2024)). Reinforcement learning appears predominantly in the deployment phase, applied in contexts requiring adaptive policy learning (e.g.,Lv *et al.* (2024); Song *et al.* (2023); Lv *et al.* (2022); Yang *et al.* (2019)).

Evaluation practices reflect these distinctions. Studies in the identification phase rely mostly on software design metrics, such as modularity, cohesion, and coupling, which evaluate the structural quality of identified microservices candidates (Qian *et al.*, 2023; Desai *et al.*, 2021; Zhong *et al.*, 2023; Rathod *et al.*, 2023; Al-Debagy & Martinek, 2021a; Chen *et al.*, 2023b; Liu & Zhang, 2024). Most monitoring studies (17 out of 18 PSs) rely on classification and prediction metrics such as precision, recall, F1-score, and AUC. However Tong *et al.* (2023) validated their approach using only system behavior metrics, including Average Waiting Time (AWT), SLA violation rate, latency, and the standard deviation of autoscaling performance. Other studies (Gan *et al.*, 2022, 2021; Luan & Shen, 2024) combined classification metrics with system behavior metrics. All deployment studies apply system behavior metrics such as

latency or throughput to assess performance improvements and adaptability. Developer-centric or human-in-the-loop metrics are rare, particularly in pre-migration (Alshammari *et al.*, 2023), and during the identification phase (Stojanovic & Lazarević, 2023; Morais *et al.*, 2021; Gysel *et al.*, 2016b).

We also examined input–technique pairings. Source artifacts are mostly combined with classical ML and, to a lesser extent, graph-based ML (Trabelsi *et al.*, 2024a; Eski & Buzluca, 2018; Kong *et al.*, 2024). Runtime artifacts drive the use of deep learning and reinforcement learning, given their rich behavioral data and need for temporal modeling (Lv *et al.*, 2024; Song *et al.*, 2023; Khan *et al.*, 2023; Lv *et al.*, 2022; Ding *et al.*, 2024; Chen *et al.*, 2022). Domain artifacts are used with supervised techniques aimed at classification or prediction tasks during specially during the indentification phase (Sun *et al.*, 2022; Al-Debagy & Martinek, 2019).

1.9.6 Key Research Gaps Identified

In addition to the challenges explicitly reported by the primary studies, our systematic review and cross-dimensional analysis reveal several overarching research gaps in the application of machine learning for monolith-to-microservices migration:

- **Insufficient dataset availability and diversity.** The majority of studies rely on synthetic or small-scale open-source systems. Industrial datasets are rarely used, and even fewer are made publicly available, limiting the reproducibility, comparability, and external validity of existing approaches.
- **Absence of standard evaluation frameworks.** There is no unified benchmark or evaluation protocol for assessing ML-based migration techniques, making it difficult to compare results across studies or measure progress consistently.
- **Limited tool accessibility and maturity.** Only 30% of the reviewed studies provide open-source implementations. The lack of accessible and production-ready tools hinders both reproducibility and practical adoption in industrial contexts.

- **Over-reliance on low-level structural metrics for service identification.** Most techniques focus on coupling, cohesion, or static dependencies, while overlooking architectural criteria such as bounded contexts, single responsibility, or alignment with domain concepts.
- **Lack of automation in the packaging phase.** Despite its central role in migration, the packaging phase—responsible for refactoring code, defining APIs, and encapsulating services—remains entirely unaddressed by existing ML-based approaches.
- **Overlook deployment artifact generation.** While some ML techniques target resource allocation and autoscaling, the automated generation of deployment configuration files (e.g., Dockerfiles, Kubernetes manifests) is almost entirely overlooked.

1.10 Threats to Validity

In this section, we identify potential threats associated with construct validity, internal validity, external validity, and conclusion validity.

1.10.1 Construct Validity

This aspect focuses on the sources used and the approach adopted for data collection. It includes the selection process of primary studies and the methodology employed to extract data in relation to the research questions. ***Exclusion of relevant studies*:** The possibility of overlooking relevant studies poses a threat to the study. Since our search was based on titles and keywords, there is a chance that some relevant studies were unintentionally excluded. The effectiveness of our search depends largely on how well digital libraries index research papers. To address this issue, we used seven widely recognised and comprehensive databases for literature reviews (Dyba *et al.*, 2007). Additionally, we conducted four rounds of snowballing to identify further potentially relevant studies.

***Selection bias in PSs*:** The process of selecting PSs may have led to the exclusion of relevant studies. As the selection was conducted manually, subjective judgment could have influenced the final set of chosen studies. To mitigate this risk, we clearly defined the study's objectives

and research questions beforehand, adhering to the PRISMA guidelines. We also established well-defined inclusion and exclusion criteria.

***Bias in data extraction*:** Since data extraction was performed manually, there is a risk of personal bias affecting the results. To reduce this risk, we designed a structured data collection form. Two researchers independently carried out the extraction process following the predefined form. We ensured consistency through interrater agreement and held multiple discussions involving all authors to reach a consensus. Furthermore, inconsistencies in terminology across PSs posed another challenge. We addressed this by systematically discussing and agreeing upon a standardised vocabulary.

1.10.2 Internal Validity

Internal validity pertains to the methods employed in this study and the accuracy of the conclusions derived from them.

***Review completeness*:** This study primarily investigates the use of ML techniques in the migration process: phases, input, approach, evaluation, and challenges. Some PSs may not provide exhaustive details on all these aspects. To minimise this limitation, we included only those studies that can answer at least three RQs.

***Research methodology*:** The methodology used in this study may introduce certain biases, potentially impacting the reliability and validity of the findings. To mitigate this risk, we followed the updated PRISMA guidelines and carefully curated our study selection process to ensure relevance. The inclusion and exclusion criteria were rigorously defined and reviewed by all authors to enhance objectivity.

1.10.3 External Validity

This concerns the extent to which our findings can be generalised across all migration approaches that use ML. Our review focuses exclusively on academic literature, meaning that industry practices may not be fully represented if they were not documented in research

papers. Additionally, we only considered studies published within a specific timeframe, which could limit the generalizability of our conclusions. Nonetheless, we plan to conduct an industry-focused study to supplement these findings.

1.10.4 Conclusion Validity

This aspect pertains to the soundness of the conclusions drawn from the extracted data. To ensure validity, we conducted a thorough analysis of the extracted data and engaged in multiple discussion sessions to cross-verify our conclusions. We strictly based our findings on the data derived from the selected PSs, ensuring that our conclusions remain well-supported and justified.

1.11 Conclusion

This systematic literature review examined how machine learning techniques have been applied to support the migration from monolithic systems to microservices. Following PRISMA guidelines, we analysed 81 primary studies to assess which migration phases are most commonly automated, what types of inputs are used, which ML techniques are applied, how these approaches are evaluated, and what challenges persist across the field.

Our findings indicate that the majority of ML-based migration approaches focus on three main phases: service identification, deployment optimisation, and post-migration monitoring. However, the review reveals several critical research gaps that remain insufficiently addressed. These include: (1) an over-reliance on low-level software metrics for service identification, with limited consideration of architectural principles such as bounded contexts and single responsibility, (2) a complete lack of automation in the microservice packaging phase, which is essential for transforming identified components into deployable services, and (3) minimal attention to the automated generation of deployment configuration artifacts, such as Dockerfiles and Kubernetes manifests. In addition to these core gaps, we observed broader limitations related to tool availability, dataset standardisation, and the absence of consistent evaluation frameworks.

To address these limitations, the remainder of this thesis proposes an ML-based framework that tackles three underexplored but essential phases of the migration pipeline: service identification, microservice packaging, and deployment configuration. Each of the next chapters introduces a dedicated approach to these phases, directly building upon the gaps identified in this review:

- Chapter 2 introduces *MicroMiner*, a machine learning–based approach that uses static analysis and semantic clustering to identify architecturally relevant microservices from monolithic source code, improving boundary accuracy and automation level.
- Chapter 3 presents *MicroPacker*, an approach that leverages large language models to automate code restructuring, API generation, and pattern integration—responding directly to the lack of automation in the packaging phase.
- Chapter 4 introduces *MiDKo*, which uses retrieval-augmented generation to automate the production of deployment artifacts such as Dockerfiles and Kubernetes manifests, addressing the manual nature of template-based tools.

In this thesis, we aim to push the boundaries of automation in microservices migration by delivering an automated tool-augmented framework that addresses practical challenges identified in this review.

CHAPTER 2

MICROSERVICES IDENTIFICATION PHASE

2.1 Introduction

Microservices identification is widely recognised as the most challenging phase of the migration process (Khadka, Saeidi, Jansen & Hage, 2013; Abdellatif *et al.*, 2018; Lewis, Morris, O'Brien, Smith & Wrage, 2005). This difficulty stems from the need to ensure that identified microservices meet stringent criteria related to fitness for purpose, modularity, quality of service, and operational independence.

Our SLR (presented in Chapter 1) further revealed that while numerous approaches have been proposed to support microservices identification (Fritzsch *et al.*, 2018; Ahmadvand & Ibrahim, 2016; Gysel, Kölbener, Giersche & Zimmermann, 2016a; Hassan *et al.*, 2017; Klock, Van Der Werf, Guelen & Jansen, 2017), most of them rely heavily on structural metrics such as coupling and cohesion. However, these metrics alone are insufficient for generating architecturally sound microservices. They often fail to account for domain-driven concerns such as the single responsibility principle and bounded contexts, which are foundational to microservice architecture (Dragoni *et al.*, 2017). Moreover, several approaches assume access to high-level artifacts such as business process models, use cases, or activity diagrams, which are rarely available in practice, thereby limiting their applicability and accuracy.

To overcome these limitations, we propose *MicroMiner*, a type-based microservices identification approach that leverages machine learning and semantic analysis to guide the decomposition of monolithic software systems. Our approach is driven by the identification of service types, predicted using a trained classification model. These types inform the clustering of components into microservices that adhere to two key principles: single responsibility and loose coupling. Unlike purely structural approaches, *MicroMiner* incorporates both static dependencies and semantic relationships to ensure that each resulting microservice aligns with a bounded context and encapsulates a cohesive business functionality.

In this context, we adopt the service type taxonomy proposed by Abdellatif *et al.* (2020b), which distinguishes between:

- **Application services**: services that implement business functionalities within a single application.
- **Entity services**: services that provide access to persistent data and support CRUD operations.
- **Utility services**: platform-specific services that offer cross-cutting functionalities such as logging or configuration.

While the broader taxonomy also includes Enterprise and Business services, our approach focuses on identifying Application, Entity, and Utility services. This is because the distinction between Application and Enterprise services depends on the reuse scope, which cannot be inferred from source code alone.

While our work builds on the idea of distinguishing service types, it differs fundamentally from earlier type-sensitive approaches such as ServiceMiner (Abdellatif *et al.*, 2020b), which first applies hierarchical clustering on static dependencies to obtain candidate services, and then classifies them into Entity, Utility, and Application types using deterministic, rule-based heuristics. Such approaches are limited in generalisability because their rules must be explicitly tuned for each system. Also, this approach stops at typed services rather than producing complete microservices. By contrast, *MicroMiner* predicts class types upfront using a machine learning model and leverages them to guide clustering, enabling the direct identification of architecturally-relevant microservices that encapsulate cohesive business functionalities.

We validate our approach on four monolithic software systems, build independent ground-truths, and show that *MicroMiner* identifies architecturally-relevant microservices with a precision of 68.15% and a recall of 77%. We further measure and validate the quality of *MicroMiner* using five quality metrics (Jin *et al.*, 2019). In addition, we compare our results with two static-based microservices identification approaches: ServiceCutter (Jain, Zhao & Chinta, 2004), which is a heuristic-based microservices identification approach, and another state-of-the-art microservices

identification tool proposed by (Brito *et al.*, 2021). The results show that microservice candidates produced by *MicroMiner* are by far the best in terms of functional independence and modularity.

The remainder of this chapter is organised as follows. Section 2.2 presents the *MicroMiner* approach in detail, including its three main stages: class typing, typed-service identification, and microservice mapping. Section 2.3 describes the experimental setup. Section 2.4 presents the quantitative and qualitative results of the *MicroMiner* approach. Section 2.5 discusses the threats to validity and provides practical recommendations for applying the approach. Finally, Section 2.6 concludes the chapter.

2.2 Identification Approach: MicroMiner

We want to decompose monolithic software systems into microservices that are based only on analyses of the source code of object-oriented monolithic software systems.

Figure 2.1 shows our proposed approach for identifying microservices in object-oriented monolithic software systems. It is divided into three main stages:

- The first stage is *Class Typing*, which decomposes a system horizontally into three layers by classifying each class in the system. We propose a method based on ML to predict/assign a label to each class. Labels are *Application* for classes belonging to the Business layer, *Entity* for the persistence layer, and *Utility* for the Utility layer. The layers and their classes are input to the next stage of our approach. We details this first stage in Section 2.2.1.
- The second stage is *Typed-Service Identification*, which identifies *Application*, *Entity*, and *Utility* services from the classes in each layer. We rely on a graph clustering technique that takes into account the static relationships among classes. We describe this stage in Section 2.2.2.
- The third stage is *Services to Microservices Mapping*, with generates the microservices.It groups the typed services using soft clustering, which is a type of clustering in which each element can belong to more than one cluster. In our context, an element is a service while a cluster corresponds to a microservice. We identify then the microservices according to the

analyses of (1) the relationships between the services and (2) the application domain. Each microservice is composed by one or many *Application*, *Entity*, and *Utility* services. The details of this stage are in Section 2.2.3.

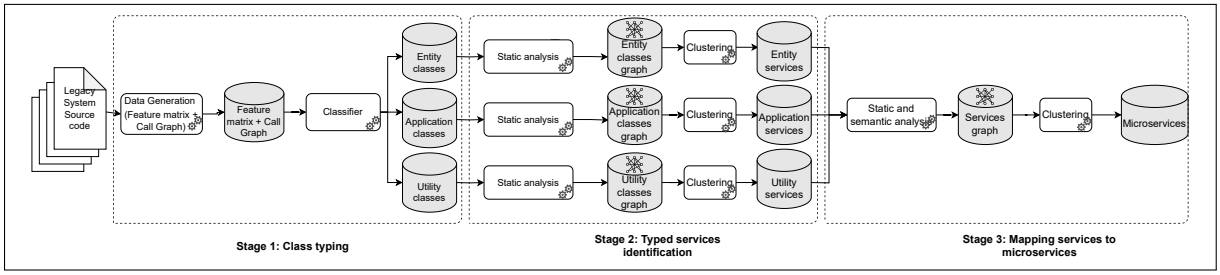

Figure 2.1 Overview of *MicroMiner*

2.2.1 Stage 1: Class Typing

This section describes the first stage of our microservices identification approach, which relies on ML to classify each class in a system. At the end of this stage, the model will classify each class of a monolithic software system as *Application*, *Entity*, or *Utility*.

Mathematically, we looked for the probability distribution $\mathcal{P}(\mathcal{Y}|\mathcal{I}, \mathcal{D})$ of missing labels of classes, where $\mathcal{Y}$ is a class label, $\mathcal{I}$ is the input of the classifier, and $\mathcal{D}$ is the training dataset. We represent our dataset as a direct graph structure $\mathcal{G} = (\mathcal{V}, \mathcal{X}, \mathcal{E})$ where all the classes of the system form the set of the nodes indexes $\mathcal{V}$, the node feature matrix $\mathcal{X}$, and the calls among classes form the set of edges $\mathcal{E}$.

2.2.1.1 Call Graph Generation

As a data generation pre-processing step on the source code, we construct our graph $\mathcal{G} = (\mathcal{V}, \mathcal{X}, \mathcal{E})$ by generating the call graph $\mathcal{E}$. We parse the source code of the system and build its model using the OMG *Knowledge Discovery Metamodel* (KDM) (El Boussaidi, Belle, Vaucher & Mili, 2012), which was defined to represent (legacy) systems at different levels of abstraction and regardless of the used languages and technologies. We use MoDISCO (Bruneliere, Cabot, Dupé & Madiot, 2014), an open-source Eclipse plugin that provides an extensible framework,

(1) to obtain KDM models from source code in different languages and (2) to visit the KDM models and generate the call graphs. For each system, MoDisco generates the corresponding KDM file in an XML Metadata Interchange (XMI) format[1], an OMG standard for exchanging metadata information via eXtensible Markup Language (XML). It contains a representation of the software code elements (e.g., packages, classes, methods, attributes, etc.) and their associations (e.g., inheritance, aggregation, association, the relationship of containment between packages, etc.).

Running example.

In the following, we use a running example to illustrate our work. We use a simplified version of the *FXML-POS* monolithic system. *FXML-POS* is an open-source[2] inventory management system in Java. It includes 56 classes. It follows the model-view-controller architecture and provides several features related to ERP, such as purchase management, sales management, supplier management, etc. We exclude some classes and consider only the 13 classes related to sales, purchases, and products to simplify this running example. This excerpt contains four classes and four relationships. *ProdcutModel* implements *ProductDao*. *ProductController* instantiates *ProdcutModel* and invokes three of its methods. Finally, *ProdcutModel* instantiates *Product*.

2.2.1.2 Feature Matrix Generation

We generate the feature matrix $\mathcal{X}$ by one of the following methods:

- **Metrics and relations:** Our first initiative was to use the same class-level metrics and method-level metrics $\mathcal{M}$, which were used in a rule-based service-type detection approach (Abdellatif *et al.*, 2020a), to obtain the feature matrix $\mathcal{X}_{|\mathcal{V}|\times|\mathcal{M}|}$. The method-level metrics were used to compute some class-level metrics. For example, we considered the number of incoming/outcoming calls to/from each method in the system to compute the fan-in (the

[1] https://www.omg.org/spec/XMI/2.1.1

[2] https://github.com/sadatrafsanjani/JavaFX-Point-of-Sales

number of incoming calls) and fan-out (the number of outcoming calls) related to each class. The class-level metrics also include the Mccabe cyclomatic complexity, the number of "try-catch" in each class and the number of database queries. However, later experimental results, in Table 2.5, showed that we could not achieve good accuracy for the classification. To improve the accuracy of the results, we added extra features related to the relationships between the classes, such as aggregation, inheritance, and implementation. Therefore, we define the matrix $\mathcal{X}_{|\mathcal{V}|\times|\mathcal{V}|}$ as:

$$\mathcal{X}(i,j) = \begin{cases} 1, & \text{if classes } i \text{ and } j \text{ are related by an inheritance,} \\ & \text{aggregation, or implementation relationship} \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

which improved the results. We further increased accuracy by concatenating the two feature matrices and obtaining $\mathcal{X}_{|\mathcal{V}|\times|\mathcal{M}+\mathcal{V}|}$, whose results we detail in Section 2.4.1.1.

- **CodeBERT:** Our second initiative was to feed the source code into the pre-trained model for programming languages, CodeBERT (Feng *et al.*, 2020) to generate the node features of the graph. We can think of CodeBERT as a function that maps source-code classes into a feature embedding within a n-dimensional space, where each class has its corresponding n-dimensional vector representation that can be easily fed into the ML classifier.

  We rely on CodeBERT to generate the representation of the source code because (1) it has been shown to achieve superior performance in many NLP tasks, (2) it supports several programming languages, such as Go, Java, Python, Ruby, etc., (3) the pre-trained model of CodeBERT is open source[3] and easily integrated into our approach. CodeBERT generates the node feature matrix $\mathcal{X}$ of size $|\mathcal{V}| \times n$, where each row i represents the node feature of the source-code class i with dimension n.

[3] https://github.com/microsoft/CodeBERT

2.2.1.3 System Classes Classification

Given the labelled node feature matrix $\mathcal{X}$ and the call graph $\mathcal{E}$, we train supervised ML models to classify the source-code classes as belonging to *Application*, *Entity,* or *Utility* services.

The supported classifiers of our approach are:

- **Support Vector Machine (SVM)** is a supervised classical ML algorithm for classification or regression tasks (Amari & Wu, 1999). It is a widely used and relatively simple. The classifier separates data points using hyperplanes with the largest margin. It finds optimal hyperplanes in multidimensional space to separate different classes and classify new data points (an SVM classifier is also called a discriminant classifier).
- **Graph Convolutional Network (GCN)** is one of the most well-known deep Graph Neural Networks (GNNs) (Kipf & Welling, 2016). It provides a powerful neural network architecture for learning features from graphs by inspecting neighbouring nodes. In our approach, we use GCN as a classification model that uses the generated embedding node features $\mathcal{X}$ and the call graph $\mathcal{E}$ to learn how to classify the source-code class to *Application*, *Entity*, or *Utility* service.

In addition to SVM, *MicroMiner* supports different classical ML classifiers such as ***Decision Tree*** (Swain & Hauska, 1977), ***K-Nearest Neighbour (KNN)*** (Peterson, 2009), ***Logistic Regression*** and ***Naive Bayes*** (Rish *et al.*, 2001). However, SVM gives the best accuracy among all these algorithms (see Table 2.6).

2.2.2 Stage 2: Typed-Service Identification

We now identify the services in the monolithic software system by analysing the static relationships between the classes of the same layer, i.e., of the same type. Then, we apply the Louvain community detection algorithm (Blondel, Guillaume, Lambiotte & Lefebvre, 2008) on each class layer to group and create Application, Entity, and Utility services.

2.2.2.1 Static Relationship Computation

We analyse the static relationships among the constituents of each class in each layer (methods, fields, etc.). A relationship could be a *generalization*, an *aggregation*, or an *association* between classes, for example. We assign a weight to each of them according to their relative importance. The class relationship weights are assigned as follows. We first rank the relationships based on their strength or relative importance. The strength of a class relationship is based on how dependent the classes involved in the relationship are. More precisely, two classes that are strongly dependent on one another are considered tightly coupled and should be in the same cluster. Consequently, if we change one class, this will most likely affect the other class. We classify the class relationships into three types according to their importance and assign a weight between 5 and 100 according to the strength of the relationship. The values of the assigned weights are depicted in Table 2.1. First, we assign the weight of 100 for highly dependent (strong) class relationships such as Generalization. Second, we assign the weight of 25 to relationships with medium dependencies such as Association. Finally, we assign the weight of 5 to class relationships with low dependencies, such as method invocation between two classes. The total weight between a pair of related constituents is:

$$\text{Weight}(C_i, C_j) = \sum_{t=1}^{T} W_t \times NR_t \tag{2.2}$$

where C_i and C_j are the constituents, T is the number of relationships, W_t the weight of a relationship of type t, and NR_t the number of such relationship between C_i and C_j. We thus obtain three graphs representing each layer.

Table 2.1 Assigned weights of the static relationships

Relationship	Generalization	Aggregation	Implementation	Association	Instantiation	Method invocation
Weight	100	100	100	25	25	5

Generalization and Implementation are identified through inheritance and interface clauses, which create strong and stable dependencies. Aggregation occurs when a class contains another class as a field, representing a structural "has-a" relationship, whereas Association denotes weaker usage dependencies such as parameters, return types, or local variables. Instantiation is detected when objects are explicitly created using the `new` keyword, reflecting a runtime dependency. Finally, Method Invocation is captured when a class calls a method from another, which represents the weakest coupling. This hierarchy of relationships explains the assigned weights, with inheritance and structural ties ranked highest, and simple invocations lowest.

Running example.

After analyzing the relationships among classes in the running example, we compute their weights. For example, class *ProductController* instantiates class *ProductModel* and invokes three of its methods. Thus, using the weights in Table 2.1, we obtain

$$\text{Weight}(C_{\text{ProductController}}, C_{\text{ProductModel}}) = 25 \times 1 + 5 \times 3 = 40 \tag{2.3}$$

2.2.2.2 Type-specific Services Identification

In each graph of each layer, we group classes into candidate services. The grouping algorithm considers the weights *w(e)* on the edges when searching for the communities. A large weight

$w(e_k)$ on an edge e_k between two nodes `A` and `B` implies that the classes `A` and `B` belong to the same candidate service.

We rely on the Louvain community detection algorithm (Blondel *et al.*, 2008) to derive communities from the graphs. It is an unsupervised algorithm divided into two steps: modularity optimisation and community aggregation. For modularity optimisation, it initialises the communities randomly. Then, it relocates each node into a different community until there is no significant increase in modularity. Nodes of each community are collapsed into one node, and the same process is repeated.

To improve the clustering results achieved by the Louvain community detection algorithm, we aggregate modules/communities that are only accessible from some other modules in a same cluster. For example, if a class `A` is only accessible from classes `B` and `C`, with `B` and `C` in a same cluster *CL*, then we put `A` in the cluster *CL* and obtain $CL = [A, B, C]$.

Running Example.

Table 2.2 shows the results of the second step of our approach on the running example. After applying the clustering algorithm and the refinement process, we obtained five typed services.

Table 2.2 Running example: Typed-Services

Service	Utility Service 1 (US1)	Entity Service 1 (ES1)	Entity Service 2 (ES2)	Application Service 1 (AS1)	Application Service 2 (AS2)
Classes	HibernateUtil	Product ProductDao ProductModel	Sale SalesModel SaleDao	ProductController ProductInterface productEditController productAddController	SaleInterface SalesController

2.2.3 Stage 3: Services to Microservices Mapping

We generate microservices based on the typed services obtained in the previous stage by analysing the static and semantic relationships among identified services. We choose each Application service as the core business component of each microservice and merge related Entity and Utility services to form the final microservices, as follows.

2.2.3.1 Static Relationship Computation

We consider the typed services that have been identified in the previous stage as our unit of work, which are no longer the source-code classes. We are now interested in the relationships between services, which we compute in two steps.

First, we compute the weight of the direct relationships between two services. We consider all calls between the services by computing the sum of the weights of the incoming and outcoming calls between the services using the previously generated call graphs. We create an undirected, edge-weighted graph $\mathcal{G} = (\mathcal{E}, \mathcal{V})$, in which each node $vi \in \mathcal{V}$ corresponds to a service $si \in \mathcal{S}$ and each edge $e_i \in \mathcal{E}$ represents the relationships between two services. Each edge e_i has a weight that shows how strong the link between two services is. Second, we apply Floyd–Warshall algorithm (Floyd, 1962) to create the adjacency matrix between all services of the graph $\mathcal{G}$ by finding the shortest path between each node of the graph. Thus, we obtain the static distance between two services.

2.2.3.2 Semantic Relationship Computation

Each microservices must have a single responsibility. Consequently, we perform a domain-related service decomposition to generate isolated functional microservices that belong to a specific bounded context (Newman, 2015b). We analyse the semantic relationships between the identified services to group them according to their domain. Figure 2.2 shows that the semantic analysis of the source code is used to extract the semantic coupling between the services in four steps.

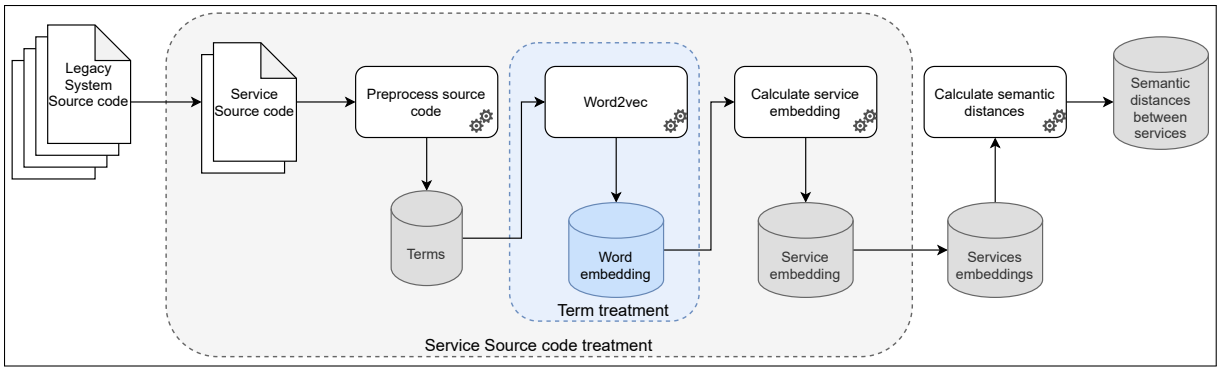

Figure 2.2 Semantic analysis pipeline

First, we perform a pre-processing step, which involves: (1) tokenizing the source code in which the text is separated at each blank space, (2) removing any terms related to the programming language (e.g., `public`, `private`, etc. in Java), (3) separating the composed terms using naming conventions like camel case and underscore (Enslen, Hill, Pollock & Vijay-Shanker, 2009), and (4) lemmatizing terms into their base forms.

Second, we encode each term in a numerical representation to facilitate their manipulation. We use word embedding, which is one of the most popular representations of words. It transforms terms into a numerical vector and captures the context of a word in a document, the semantic and syntactic similarity, the relationship with other words, etc. We use the pre-trained, Google-news-based Word2Vec that contains three million terms [4].

Third, we determine the vector representation of the services, i.e., their embeddings. We use a simple yet efficient technique to obtain the mean of all term vectors related to a service. Indeed, a study showed that, for the sentiment text analysis task, using the average of term embeddings to obtain a document embedding yields similar results compared to more complex techniques (Iyyer, Manjunatha, Boyd-Graber & Daumé III, 2015).

Finally, we compute the cosine distance between the service embeddings to obtain the semantic distances between each pair of services.

[4] https://github.com/mmihaltz/word2vec-GoogleNews-vectors

2.2.3.3 Microservices Generation

To assure both the contextual consistency and high cohesion of the microservices, we rely on both static and semantic weights. To find the final weight to qualify how strong the relationship between the services/clusters is, we combine both weights and perform a unit-based normalization on the static and semantic weights to adjust their values to $[0, 1]$. Then, we compute the final weight w_{ij} by balancing the other two weights $w_{Static_{ij}}$ and $w_{Semantic_{ij}}$ using two parameters α and β:

$$w_{ij} = \alpha \times w_{\text{Static}_{ij}} + \beta \times w_{\text{Semantic}_{ij}} \tag{2.4}$$

An expert must specify these parameters as they depend on the systems, e.g., in the case of a system in which components are poorly named, β could be reduced in favour of α to reduce the dependence on the semantic analysis in favour of the static relationships/analysis.

After calculating the final weights between the typed services, we cluster these services to compose the microservices. In this step, we consider each application service as the central element of each microservice c_i, because they "own" the functionality. We use the fuzzy C-means clustering (FCM) algorithm (Bezdek, Ehrlich & Full, 1984) because it is one of the most widely used fuzzy clustering algorithms. The FCM algorithm returns membership scores representing the degrees of membership of data points x_i to each cluster c. This membership score ms_{ij} is calculated by:

$$ms_{ij} = \frac{1}{\sum_{k=1}^{c} \left(\frac{\|x_i - c_j\|}{\|x_i - c_k\|} \right)^{\frac{2}{m-1}}} \tag{2.5}$$

where m is the fuzziness parameter and k is the number of clusters.

We apply the FCM algorithm with one iteration because we do not want to change the cluster centres. Then, we must specify a threshold at which we consider whether a service belongs or not to a microservice. Again, an expert should specify this threshold, especially because it may vary from one system to another.

Our approach respects the loose coupling principle of microservices. If two microservices dependent on a same service, we duplicate this service into each microservice to prevent any dependency between them. For example, in the case of a retail management software, the Entity service *Product*, which manages product data, is used by the microservices *Sales* and *Product Restocking*. We include this Entity service in both microservices.

Running example.

Finally, we obtain two microservices for our running example. *MS1={US1,ES1,AS1}* and *MS2={US1,ES1,ES2,AS2}*. The Utility service *US1* is duplicated in both microservices. The *Product* Entity service ES1 is also duplicated in both the *Sales* and *Product* microservices, due to its strong coupling with the two services.

2.3 Study design

The overall quality of the generated results of our approach has been first evaluated by the fifth author, who is an expert in microservice-based systems. He thoroughly analysed the systems and the generated microservices decomposition by our approach to making sure that the identified microservices embed cohesive classes and belong to bounded contexts. Despite the positive feedback of our expert regarding the generated results, we further validate our approach by (1) comparing our results with two state-of-the-art microservices identification approaches, and (2) relying on qualitative metrics to further evaluate the quality of the identified microservices.

This section presents the experiments that we conducted on four case studies to validate our approach both quantitatively against a ground-truth and qualitatively using evaluation metrics.

We provide further details on the setups and outputs of the three stages of our approach: the Class Typing, the Typed-service Identification, and the Microservices Mapping.

2.3.1 Case Studies

To evaluate the approach, we sought legacy systems that (1) are open source and available online, (2) whose initial architecture is monolithic so we discarded service-oriented systems, and (3) are object-oriented because we rely on classes as the basic entities in our approach. However, we faced a lack of availability of systems that meet these three criteria. Therefore, we conducted our studies on only four systems of different sizes that we could find in the literature and that were used to validate similar state-of-the-art approaches. Thus, we considered Compiere, a large ERP system that contains 1,042 classes; JForum, a medium-sized system that contains 271 classes; and PetClinic and POS which contain fewer than 100 classes.

2.3.1.1 Compiere

Compiere is one of the few large, Java, open-source *legacy* ERP systems. It was first introduced by *Aptean* in 2003[5]. It provides businesses, government agencies, and non-profit organisations with flexible and low-cost ERP features[6], such as business partners management, warehouse management, purchasing and sales order management (quotes, book orders, etc.). We use *Compiere* v3.3 because (1) it is the first stable release of the system, (2) it was released more than 15 years ago, (3) it is not based on microservices.

2.3.1.2 FXML-POS

This system was presented as our running example in Section 2.2.1.1 as well as used as a case study to validate our approach. We use *FXML-POS* because it offers several features and it is not microservice-oriented.

[5] http://www.aptean.com

[6] http://www.compiere.com/products/capabilities/

2.3.1.3 PetClinic

PetClinic is an open-source[7] Java-based veterinary clinic management system that allows veterinarians to manage information about pets and their owners. It is based on the model-view-controller architecture and has 52 classes. This system provides several features such as pet management, owner management and visits management. We chose to use PetClinic because there is also a new version of the system built using the microservices architecture that serves as the basis for creating our ground-truth.

2.3.1.4 JForum 3

JForum is an open-source discussion board system implemented in Java. We worked with the last version that includes nearly 300 classes. It is based on the model-view-controller architecture and provides several features, such as user management, message system, topic management, etc. We use JForum because (1) it is not microservice-oriented, and (2) it was previously used to validate several state-of-the-art monolith-to-service migration approaches (Saidani, Ouni, Mkaouer & Saied, 2019; Lapuz, Clarke & Abgaz, 2021; Jin *et al.*, 2019).

2.3.2 Ground-truths

To assess the reliability of our approach, we need two kinds of ground-truths for each system. The first type of ground-truth is related to the services and their types. This type of ground-truth will be used to validate the two first stages of our approach (class typing and typed-service identification). Also, we need a second type of ground-truth which is related to the microservices in the monolithic systems. It will be finally used to evaluate the microservice mapping. We asked two independent PhD students to identify services and microservices in *Compiere* and *FXML-POS* systems. They relied on several artifacts to build the ground-truth architectures manually by (1) analyzing and understanding the systems and (2) extracting the reusable parts

[7] https://github.com/spring-projects/spring-petclinic

that could become services / microservices. To recover their designs and to visualize class dependencies, they used Understand [8] integrated development environment. Additionally, they generated views of their call graphs that we make available online [9]. They also reviewed extensively the system documentation as well as their source code to have the best possible understanding and accurately identify both services and microservices that can be integrated into the targeted SOA-based system and in a microservice architecture. Finally, they annotated the services manually according to their types. Table 2.3 shows the statistics of the ground-truth decomposition.

Table 2.3 Overview of the ground-truths

Legacy system	# System-Classes	# Entity Services	# Application Services	# Utility Services	# Microservices
Compiere	1,042	358	30	85	92
FXML-POS	55	9	10	3	9
PetClinic	52	7	7	4	7
JForum 3	271	31	73	19	61

In most systems, the number of application services is close to the number of microservices because application services usually encapsulate domain-specific functionalities that can be mapped almost directly into independently deployable units. For instance, in *JForum 3*, *FXML-POS* and *PetClinic*, each application service corresponds nearly one-to-one to a microservice. However, this correlation does not hold for *Compiere*. Due to its highly modular architecture and the strong interactions between application, entity, and utility services, the decomposition process often required splitting a single application service across multiple microservices. In other words, one application service in *Compiere* may be distributed into different microservices to preserve consistency, reduce coupling, and manage the complexity of its business processes.

[8] https://www.scitools.com/

[9] http://si-serviceminer.com

2.3.3 Evaluation Metrics

To evaluate the quality of the identified microservices, we follow the work of (Jin *et al.*, 2019) and consider the following metrics that are related to functionality independence and modularity aspects.

2.3.3.1 Independence of Functionality

The functionality independence refers to the external independence, i.e., the independence and consistency of the functionality that the microservice provides to its external users, as defined by the single responsibility principle (SRP). These metrics are calculated using the interfaces of a microservice. Typically, an interface is a class that exposes functionality as an endpoint. The methods for each interface are considered as operations.

IFN (*Interface Number*) *ifn* measures the number of published interfaces of a microservice. The smaller the *ifn*, the more likely the microservice has a single responsibility. The *IFN* is the average of all *ifn*.

CHM (*Cohesion at Message level*) *chm* measures the cohesion of interfaces published by a microservice at the message level. This is achieved by calculating the similarity between two message-level operations based on parameters and return values. A higher *chm* represents a higher cohesiveness of the microservice. The *CHM* is the average all *chm*.

CHD (*Cohesion at Domain level*) *chd* measures the cohesion of interfaces published by a microservice at the domain level. This metric measured very similarly to *chm*, but instead of using only the message terms, all domain terms contained in the operation signature are taken into account. A higher *chd* represents a higher cohesiveness of the microservice. The *CHD* is the average all *chd*.

2.3.3.2 Modularity

Modularity evaluates the cohesion of microservices in their internal interactions and the looseness of interactions between microservices. To evaluate the modularity of service candidates, we use the following metrics: Quality of structural modularity (*SMQ*) and Quality of conceptual modularity (*CMQ*).

SMQ (*Structural Modularity Quality*) measures the quality of modularity from a structural point of view. The higher the *SMQ*, the more modular the service is. The SMQ calculation is composed of two terms: The first measures the structural cohesion of a service (intra-connectivity) using the number of edges within a service, and the second measures the coupling between services (inter-connectivity) based on the number of edges between services.

CMQ (*Conceptual Modularity Quality*) measures the quality of modularity from a conceptual perspective, like the *SMQ*. *CMQ* is composed of two terms; the only difference is that in the *CMQ* definition, an edge between two entities exists if the intersection between the set of textual terms of the entities is not empty. The higher the *CMQ*, the better.

2.3.4 Experimental Setup

In the following, we present the experimental settings at each stage of the *MicroMiner* architecture.

2.3.4.1 Class Typing

For the datasets, as we mentioned in Section 2.2.1, we consider the graph representation of the four legacy systems as our dataset form. For all the classification models, we use 80% of the graph nodes for training ($\mathcal{V}_{train}$). The rest 20% of the nodes are used for testing ($\mathcal{V}_{test}$), except for GCN model is splitted into 10%, for validation ($\mathcal{V}_{valid}$) and 10% for testing. Now,

we present the setup of the experiment for our classification models categories:

Deep GNNs. We considered a two-layer graph convolutional network (GCN) for the semi-supervised classification of the nodes of a graph. At the training, the GCN classifier takes as input: **1)** The adjacency matrix $\mathcal{A}_{train}$ of the training graph $\mathcal{G}_{train}$ of the legacy system to migrate. The nodes of the call graph $v_i \in \mathcal{V}_{train}$ correspond to classes $c_i \in C$ from the system to analyse. **2)** The feature matrix $\mathcal{X}_{train}$ that embed information about the system-classes. We experimented with several representations for the feature matrix (e.g., CodeBERT embeddings, Method-level and class-level metrics, etc.) as described earlier in Section 2.2.1.2. We validate the GCN classifer using the validation components ($\mathcal{V}_{valid}$, $\mathcal{E}_{valid}$, $\mathcal{X}_{valid}$), and we test with the test components ($\mathcal{V}_{test}$, $\mathcal{E}_{test}$, $\mathcal{X}_{test}$). Table 2.5 summarises the results of GCN experiments with different feature representations on all systems. Given GCN shows the best performance with CodeBERT embedding on *Compiere* and *FXML-POS*, we state the performance results of GCN on PetClinic and JForum 3 when using CodeBERT embeddings only.

Classical ML. As we showed in Section 2.2.1.3, we used several classical ML classification models. To classify the graph nodes (i.e., system-classes) based on their features. All these classical models rely on the code embeddings that CodeBERT generates. With *SVM*, we use the linear kernel function. For *KNN classifier*, we set $k = 5$ for the number of nearest neighbours to be considered in the voting process. We also experimented with *Decision Tree* and set the tree's maximum depth to 2. For *Logistic Regression*, we used *lbfgs* as the optimisation algorithm. The final algorithm we applied was Gaussian Naive Bayes. In Table 2.6, we present the performance measures of all classifiers in our four use cases. We report the accuracy that gives the overall correctness of the models, i.e., the fraction of total samples that were correctly classified by the classifier and the F-measure for each class (Utility, Entity, and Application).

2.3.4.2 Typed-service Identification

After predicting the type of each class on our different case studies, we constructed three graphs per system: the Utility graph, the Entity graph, and the Application graph, as explained in Section 2.2.2.1. Then, we applied the Louvain algorithm for community detection to cluster and identify Utility, Entity, and Application services. For the Louvain algorithm, the only parameter to specify was the *resolution* that controls the community sizes. We have set it to its default value of 1 for all experiments.

2.3.4.3 Microservices Mapping

As mentioned in Section 2.2.3.3, we calculate the final weight to qualify the strength of the relationship between the services/clusters by combining the static and semantic weights based on two parameters α and β. Table 2.4 shows the considered values of these parameters for the different systems. For *Compiere*, the code components, variables and method names are poorly named, so we reduced the value of β in favour of α to reduce the dependence of the semantic analysis on the static relation. For the other three systems, we gave equal weights to semantic and static relationships as the naming employed in the code was expressive and meaningful.

Table 2.4 Total weight parameters

System	*Compiere*	*FXML-POS*	*PetClinic*	*Jforum*
α	0.8	0.5	0.5	0.5
β	0.2	0.5	0.5	0.5

In order to generate microservices, we cluster the typed services using Fuzzy C-means clustering algorithm. To calculate the membership score, we need to specify the number of clusters and the fuzziness m parameter. In our case, the number of clusters is equal to the number of *application services*. We used $m = 2$ for all systems because it is the most used value (Gao, PEI & XIE,

2000). Moreover, by adopting this value, we avoided getting large microservices and thus increased the cohesion of microservices while maintaining a low coupling value.

2.4 Results

In the following, we present the quantitative and qualitative results of each stage of the *MicroMiner*.

2.4.1 Quantitative Results

In the following, we present the quantitative results of each stage of the *MicroMiner* approach.

2.4.1.1 Class Typing Results

We present in Table 2.5 the GCN accuracy with respect to different feature matrix formalisms. The results show that the most accurate classification result is achieved by using the code embeddings generated through CodeBERT with an average accuracy of **74%**.

As shown in Table 2.6, the SVM classifier achieved the highest accuracy across the four systems with **86%** for *Compiere*, **92%** for *FXML-POS*, **87%** for *PetClinic* and **82%** for *JForum*. However, we observe that the F-measure of some classes is slightly higher when using other classifiers, albeit with a small difference. Based on these results, we selected SVM in the classification stage of *MicroMiner* to predict the class types of the system because it outperformed GCN and other classifiers and showed overall better results.

Table 2.5 GCN experiments results for all systems with respect to different generated feature matrix

Feature matrix	**Feature matrix dimension**	***Compiere* Accuracy**	***FXML-POS* Accuracy**	***PetClinic* Accuracy**	***JForum* Accuracy**
Method-level and class-level metrics	$\|\mathcal{V}\| \times 6$	51%	59%	N/A	N/A
Relations	$\|\mathcal{V}\| \times \|\mathcal{V}\|$	65%	62%	N/A	N/A
Method-level and class-level metrics+relations	$\|\mathcal{V}\| \times (\|\mathcal{V}\| + 6)$	58%	62%	N/A	N/A
CodeBERT embeddings	$\|\mathcal{V}\| \times 768$	**72%**	**75%**	**76%**	**73%**

Table 2.6 Source-code class classification results with the classical ML models using CodeBERT embedding of the four legacy systems

Legacy system	Quality metrics	Decision Tree	SVM	KNN (k=5)	Logistic Regression	Naive Bayes
Compiere	Accuracy	73%	**86%**	81%	83%	52%
	Class Application F1-score	54%	68%	**69%**	61%	47%
	Class Entity F1-score	85%	**93%**	91%	92%	64%
	Class Utility F1-score	13%	62%	37%	**64%**	35%
FXML-POS	Accuracy	84%	**92%**	66%	84%	84%
	Class Application F1-score	86%	**93%**	75%	86%	86%
	Class Entity F1-score	87%	**97%**	58%	90%	87%
	Class Utility F1-score	**50%**	**50%**	0%	0%	**50%**
PetClinic	Accuracy	79%	**87%**	74%	77%	69%
	Class Application F1-score	67%	**81**%	58%	**81**%	77%
	Class Entity F1-score	82%	**89%**	71%	73%	77%
	Class Utility F1-score	**50**%	**50%**	0%	0%	0%
JForum	Accuracy	79%	**82%**	69%	76%	62%
	Class Application F1-score	69%	**83%**	75%	66%	53%
	Class Entity F1-score	81%	**87%**	58%	80%	69%
	Class Utility F1-score	10%	**30%**	18%	22%	18%

2.4.1.2 Typed-service Identification Results

We used our ground-truth architecture for each system to quantitatively validate the typed services. We measured precision, recall, and F-measure for the identification of each service type and reported the results in Table 2.7. We obtained architecturally significant typed services with a total precision of 68.3%, a recall of 83.9%, and an F-measure of 75.2% for *Compiere*. We also achieved a precision of 86.3%, a recall of 86.3%, and an F-measure of 86.3% for *FXML-POS*. For PetClinic, we got a precision of 70%, a recall of 66%, and an F-measure of 68%. For our fourth system, JForum, we obtained a precision of 83.1%, a recall of 60.1%, and an F-measure of 69.7%.

The performance of this stage depends on the results of the previous classification stage because we consider the classes of each type independently. For example, we had in *Compiere* a precision of 49.2% for the identification of Application services. We missed some Application services because, in the previous stage of class typing, we had a classification accuracy of 68%. We missed some classes of type Applications to cluster into Application services. Also, in *FXML-POS*, we could not identify some Utility services for the same reasons.

To obtain better results for identifying typed-service, developers could refine the classification when applied to their systems.

Table 2.7 Overview of Typed-service identification results

Legacy system	Service type	Precision	Recall	F-measure
Compiere	Application	49.2%	81.3%	61.4%
	Entity	74.3%	90.7%	81.7%
	Utility	68.2%	83.9%	75.2%
	Total	68.2%	83.9%	75.2%
FXML-POS	Application	83.3%	100%	90.9%
	Entity	88.8%	88.8%	88.8%
	Utility	100%	33.3%	50%
	Total	86.3%	86.3%	86.3%
PetClinic	Application	75%	85%	80%
	Entity	83.3%	71.4%	76.9%
	Utility	50%	75%	60%
	Total	70%	66%	68%
Jforum	Application	89.8%	72.6%	80.2%
	Entity	77.2%	54.8%	64.1%
	Utility	50%	21%	29.5%
	Total	83.1%	60.1%	69.7%

2.4.1.3 Microservices Mapping Results

In this section, we analyse and validate the generated microservices. We used our ground-truth architectures to calculate the precision, recall, and F-measure of our approach.

We applied *MicroMiner* on *Compiere*, *FXML-POS*, *PetClinic* and *Jforum* to show its practical accuracy in identifying microservices in existing systems. We also applied these two state-of-the-art microservices identification approaches, ServiceCutter and the *Topic modelling-based* approach. We measured the precision, recall, and F-measure for each system and reported the results in Table 2.8. We found that *MicroMiner* identified architecturally-relevant microservices with 68.15% precision, 77% recall, and 72.1% F-measure.

Table 2.8 Comparison results of microservices identification approaches

Legacy system	**Approach**	**Precision**	**Recall**	**F-measure**
Compiere	*ServiceCutter*	5%	21.7%	8.1%
	Topic modelling-based	22.5%	29.3%	25.4%
	MicroMiner	75.2%	72.8%	73.9%
FXML-POS	*ServiceCutter*	7.8%	33.3%	12.6%
	Topic modelling-based	29.4%	55.5%	38.4%
	MicroMiner	72%	88%	80%
PetClinic	*ServiceCutter*	8.5%	42%	14.2%
	Topic modelling-based	36.3%	57.1%	44.3%
	MicroMiner	71.4%	71.4%	71.4%
Jforum	*ServiceCutter*	4.7%	11.9%	6.7%
	Topic modelling-based	44.1%	45.2%	44.6%
	MicroMiner	54%	76.1%	63.1%

Also, while we identified 92 microservices in the ground-truth microservice-based architecture of *Compiere*, *MicroMiner* identified 89 microservices, among which 67 were correctly composed.

We also obtained for the same system a precision of 75.2%, a recall of 72.8%, and an F-measure of 73.9%. For *FXML-POS*, while we identified nine microservices in the ground-truth, *MicroMiner* identified 11 microservices, with nine of them correctly identified. We obtained a precision of 72%, a recall of 88%, and an F-measure of 80%. Table 2.9 shows the number of duplicated services. We noticed that this number is not large because we selected a relatively small fuzziness parameter.

In general, the correctly-identified microservices are built around accurately classified Application services because we choose to consider Application services as the central component of each microservice. The correctly identified microservices in *Compiere*, for example, are related to bank-statement management, account management, partners, warehouses, orders, invoice management, etc.

The incorrectly identified microservices were mainly coarse-grained: they contained a large number of classes. When we analysed these microservices, we found that the related Application services contained tightly coupled classes but covered more than one domain/business functionality. Thus, poor identification of Application services inside a system led to poor identification of the corresponding microservices. For example, one of the identified Application services from *Compiere* aggregated classes related to both projects and payments, which led to the generation of a large microservice that did not respect the single responsibility principle.

Furthermore, when services were not correctly typed, the quality of the identified microservices suffered. For example, in *Compiere*, some classes related to *data migration* should have been labelled as Utility-related classes. However, *MicroMiner* classified these classes as Application-related, which led to the identification of incorrect Application services and, therefore, to incorrect/poor microservices.

Furthermore, our approach outperformed *ServiceCutter* and *Topic modelling-based* approach in terms of identification precision, recall and F-measure. Indeed, *ServiceCutter* generated poor results on all systems that we have considered in our experiment. We observed that this approach produced a large number of services of unbalanced size. More precisely, we observed

that *ServiceCutter* generates in general, a few large services and many small ones containing one or two classes. This considerably affects the microservices identification performance because the generated services are not architecturally relevant. For the *Topic modelling-based* approach, the generated services were partially acceptable. However, many classes that are not directly relevant to the service context, e.g., generic classes like "*EntityUtils*" or "*BaseEntity*" are always incorrectly mapped because they do not have a clear domain connection. For example, when applying the *Topic modeling-based* approach on *PetClinic*, generic classes such as "*EntityUtils*" and "*BaseEntity*" are always incorrectly mapped to their corresponding microservices, as such classes provide cross-cutting functionalities that could take part in several possible service domains. Moreover, such approach highly relies on identifying microservices according to the semantic relationships between the system components. In contrast, our approach considers both static and semantic relationships between the system's classes and relies on a more structured and two-staged clustering approach that (1) groups classes according to the types of their provided functionalities, and then (2) groups the generated clusters according to their domain.

Table 2.9 Duplicated services per system

System	*Compiere*	*FXML-POS*	*PetClinic*	*Jforum*
# duplicated services	22	5	4	12

2.4.2 Qualitative Results

To evaluate the quality of the identified microservices, we conducted a metrics-based and a content-based evaluation.

2.4.2.1 Metric-based Evaluation

Table 2.10 shows the values of the microservice quality metrics obtained using *ServiceCutter*, *Topic modelling-based approach* and *MicroMiner* across the four systems.

IFN quantifies the interfaces that are exposed by a given microservice, i.e., any class that exposes functionality as an endpoint. Hence, a smaller IFN represents a higher likelihood of a service having a single responsibility. The median is almost three in our case. Although our IFN values are relatively low, they are not optimal ($IFN = 1$) because the legacy systems are improperly designed. For example, in *FXML-POS*, we found *ProductController*, *ProductAddController*, and *ProductEditController* that should have been merged into a single class. The *Topic modelling-based* approach gives better IFN values for *Compiere* and *Jforum*.

CHM represents message-level cohesiveness. For *MicroMiner*, it has a mean around 0.7, which is a positive statement regarding the independence of microservices. Our approach outperforms the two others in most cases.

CHD quantifies the cohesion at the domain level. For *MicroMiner*, it has a mean around 0.7, which indicates that the generated microservices respect the bounded-context principle, thanks to the semantic analysis in our process. For *Compiere*, the value is low because the elements of the system are poorly named. The *Topic modelling-based* approach gives better CHD values for *Compiere* and *Jforum*.

SMQ and **CMQ** represent the modularity of the microservices. We mainly calculate the differences between cohesion and coupling and the values are bound between -1 and 1. *microminer* exhibits distinctively positive differences from the cohesion to coupling across all systems and medians values are roughly around 0.07 and 0.05 respectively. We therefore conclude that the generated microservices have a good modularity: they are loosely coupled and strongly cohesive. Our approach outperforms the two other approaches.

Table 2.10 Overview of the generated microservices quality

Legacy system	Approach	IFN	CHM	CHD	SMQ	CMQ
Compiere	*ServiceCutter*	4.1	0.23	0.21	-0.17	-0.21
	Topic modelling-based	**2.1**	0.46	**0.54**	-0.02	0.01
	MicroMiner	3.4	**0.73**	0.53	**0.11**	**0.09**
FXML-POS	*ServiceCutter*	2.9	0.54	0.33	-0.12	0.01
	Topic modelling-based	2.5	0.47	**0.79**	-0.03	0.03
	MicroMiner	**2.3**	**0.69**	0.72	**0.09**	**0.05**
PetClinic	*ServiceCutter*	2.3	0.42	0.53	0.02	-0.03
	Topic modelling-based	2.1	0.43	**0.72**	0.03	0.02
	MicroMiner	**1.9**	**0.75**	0.67	**0.05**	**0.03**
Jforum	*ServiceCutter*	3.1	0.34	0.24	-0.01	-0.03
	Topic modelling-based	**2.3**	0.43	**0.62**	0.03	0.05
	MicroMiner	2.8	**0.67**	0.56	**0.04**	**0.O6**

2.4.2.2 Content-based Evaluation

For the content-based evaluation, we detail the results of applying *MicroMiner* on *FXML-POS* to identify relevant microservices in the system. We take the example of sales and supplier management in *FXML-POS*. We qualitatively study the microservices related to these business functionalities while we detail how *MicroMiner* helps practitioners to identify such microservices.

The initial classification step of our approach predicts the layer to which each class in the system belongs (i.e., Utility, Entity, or Application). Based on this classification, we apply the Louvain community detection algorithm on each layer to identify the clusters that correspond to the typed services. Finally, based on the Application services, we perform a vertical clustering over the

layers to merge Utility, Entity, and Application services that belong to the same domain to form microservices.

First, we divide the system into three layers (Utility, Entity, and Application service layers). In the Utility layer, we have some cross-cutting functionalities, like printing, logging, and Hibernate-related functionalities. In the Entity layer, we find the DAO classes that support CRUD actions on the data and the models that represent the data: classes *SaleDAO*, *SupplierDAO*, *SaleModel*, and *SupplierModel*. In the Application layer, we find the classes that provide functionalities related to supplier and sales management. Second, we perform a clustering to create the services in each layer and to get the types of these services. Third, we choose the Sales and Vendors Application services as central components of two microservices and perform a fuzzy clustering to create the microservices. Finally, we obtain two microservices: the first is related to sales management, which comprises the Serialisation Utility service, Sales entity service, and Sales application service. The second microservice is composed of Hibernating Utility service, Vendor entity service, and Vendor application service.

Thus, we could identify with *MicroMiner* architecturally-relevant microservices in the different systems. We believe that our approach can assist practitioners in identifying candidate microservices because it relies only on the static analysis of the system source code to migrate and automate the microservices identification process with acceptable precision and recall.

2.5 Discussions

We will describe in this section the recommendations that we derived based on our observations and the threats to the validity of our approach.

2.5.1 Discussion and Recommendations

In the first stage of our approach, the use of the ML classifier may not be necessary when the different types of classes in the system are well packaged into their respective source files. However, our target in this work is legacy monolithic systems to be migrated to microservices.

These legacy systems generally embed several poor design problems and packaging issues that may hinder their maintenance. For instance, different types of classes are not necessarily packaged into their respective source files. This is the case of *Compiere*, for example, where we found different classes pertaining to different types in some packages. Also, when the system is too large and complex, it is not easy to manually check if each class in the system is correctly packaged according to its type. To deal with all these challenges and to make the approach more generalisable, we rely on our ML classifier to detect the types of classes that will be then mapped to their corresponding typed services. In our case, we use entity, application, and utility services, which are considered fine-grained services because they are specialised services that cover a specific domain context and one role (data management, business logic, etc.) at a time (Xiao, Wijegunaratne & Qiang, 2016). However, MSAs are composed of loosely coupled specialised components that often operate independently of each other. Therefore, a microservice should encapsulate all elements to perform a specific business functionality. In our case, a microservice is composed of different service types (i.e., entity, application and utility) to perform a specific business functionality.

Our approach works best on systems with a relatively good architecture and design. Although the quality of the architecture is not very impactful (as our approach does not consider packages, etc.), the quality of the system design indeed may impact our approach. In fact, if a system is poorly designed and has code smells that reduce the separation of concerns within/between classes, our approach would fail to identify (relevant) microservices like any typical static-based service identification approach.

Our approach depends on several metrics and thresholds. For the fuzziness parameter m, we recommend using values in the range [1.5, 2.5] to allow some classes to be duplicated in several microservices. By limiting the fuzziness parameter m to 2.5, we avoid obtaining large numbers of duplicated classes. Thus, we increase the cohesion of the microservices while maintaining a low coupling value. For parameters that balance static and semantic weights, their selection depends on the quality of the initial system source code, e.g., in the case of a system in which components are misnamed, the semantic weight values could be reduced in favour of the static

weight values. For the weight of each type of static relationship, we recommend using the values mentioned in this work, because we relied on previous work to assign these values. For the last step of our approach, an expert should specify the threshold for the desired size of microservices since this parameter depends on the system to migrate and the expert's expectations.

The availability of the source code is essential to migrate an existing system to microservices using our *MicroMiner* because we derive a bottom-up microservices identification approach. From a practical standpoint, the availability of the source code of monolithic or legacy systems to be migrated is common in many industrial contexts (Abdellatif *et al.*, 2021a). The source code is often the most up-to-date available source of information about existing software systems (Abdellatif *et al.*, 2021a). Furthermore, our approach is applicable to both large and small systems. For example, we validated *MicroMiner* on a relatively small system (POS). The results were promising despite the small size of such a system. Furthermore, we obtained similar good results when applying *MicroMiner* on Compiere, which is considered a relatively large system.

A common question in microservice identification is whether there exists a unique best decomposition. In practice, multiple valid decompositions may coexist, each reflecting different trade-offs such as granularity, modularity, or domain alignment. For example, one decomposition may privilege fine-grained services to maximise independence, while another may prefer coarser-grained services to reduce communication overhead. This consideration also influenced how we constructed the ground truths used in our evaluation: instead of assuming a single perfect decomposition, we designed the ground truths as one plausible and architecturally consistent decomposition, built upon expert judgment and established microservices principles. Therefore, the goal of automated approaches like *MicroMiner* is not to prescribe the one best decomposition, but to systematically generate architecturally sound candidates that satisfy microservice principles.

We believe that our approach is useful for both researchers and practitioners involved in migrating legacy systems to a microservices architecture because (1) we automate one of the most labor-

intensive steps in migrating such systems, which is the identification of microservices, (2) our approach yields architecturally meaningful candidate microservices that satisfy two main principles of MSA: loose coupling and single responsibility per microservice, (3) our approach offers the possibility to balance static and semantic weight according to their importance and the code naming quality of the legacy system, and (4) our approach can be applied to systems of different programming languages, thanks to the use of language-independent techniques in the different steps, such as CodeBERT which supports various languages.

Finally, we recommend converting utility services into libraries (jar for java, DLL for .Net languages, etc.). This could reduce the size of the source code files, minimise class redundancy and allow them to be accessible to external systems. Furthermore, as we have opted for fuzzy clustering, meaning that certain classes can occur in several microservices, we recommend performing a code slicing step at the end, to refine the microservices and get rid of dead code that is never used/reached in the microservice.

2.5.2 Threats to validity

2.5.2.1 Internal validity

Our microservices identification approach and its validation depend on several metrics and thresholds that threaten the internal validity of our results. To mitigate these threats, we used different algorithms and threshold values.

The results of our approach were qualitatively evaluated by the fifth author because of his expertise in developing microservices. However, we must accept a threat to the validity of this validation because this author participated in some meetings discussing the work in general and the approach in particular. Therefore, this author is not entirely independent and could have been biased. Yet, we accept this threat because (1) he is an expert at developing and studying microservices, and (2) finding an expert who is willing to validate our approach on different systems is difficult. Besides, we provide all results and validation for others to verify

the validation or perform it again independently. Finally, we mitigate this validation bias by (1) comparing our results with two state-of-the-art microservices identification approaches and (2) relying on qualitative metrics to further evaluate the quality of the identified microservices.

2.5.2.2 Construct validity

The quality of the legacy source code may have an impact on the results of the microservices identification. Legacy monolithic systems may embed some poor design practices that may limit the accuracy of static-based microservices identification approaches. To mitigate this threat, we rely on the analysis of both static and semantic relationships between the system's elements on different levels (in the class level and service level).

We relied on the generation of ground truths to validate quantitatively the microservices identified by our approach. Two independently PhD students extensively analysed the systems to obtain two sets of ground-truths: the first set was related to the services and their types, while the second set was for the microservices. The generated ground-truths by both students were very similar. They reconciled the differences between the generated ground-truths through discussions and end-up with common ground-truths used to validate our microservices identification approach. We are aware that there is no single "correct" microservice-based version of the analysed legacy systems. To reduce the bias, we may ask the projects' owners or software developers to provide the ground-truths. However, getting in touch with such experts is challenging. Furthermore, producing an accurate ground-truth microservice based architecture for a monolithic system is an arduous and time-consuming task. We do not expect that the projects' owners or software developers will accept to take considerable time away from their daily obligations to build the ground-truths on our behalf. However, we do not exclude such tasks and will try to involve software developers in our experiments as future work.

2.5.2.3 External validity

In order for the classification model to classify accurately, there must be a similar distribution of data on which the model makes predictions as the data on which the model was trained. The classification model in the class typing stage was trained using only four legacy systems; hence our results could not be generalised. A small portion of the labelled classes is requested to adapt the already trained model to mitigate this threat. Finally, the generalisation of our training could be enhanced if we label and analyse a large set of monolithic systems. However, building such a dataset is challenging as we have to (1) select hundreds of systems pertaining to different domains and relying on different architectures, (2) analyse and review the source code of these systems, and (3) manually classify each of the classes inside the selected systems. Considering all these challenges, we aim as future work to train the ML classifier using a large set of monolithic systems and then use the trained ML model to predict the classes of new monolithic systems.

2.6 Conclusion

In this chapter, we presented *MicroMiner*, an ML–based approach for identifying microservices within monolithic software systems. *MicroMiner* is guided by a taxonomy of service types, which are predicted using classification models trained on features extracted from source code. The approach combines static analysis of component relationships with semantic similarity analysis to ensure that each extracted microservice aligns with a single bounded context.

We evaluated *MicroMiner* on four legacy systems and compared the identified services against independently constructed ground-truths. The results demonstrate that *MicroMiner* can effectively identify modular and functionally independent microservices, achieving an average precision of 68.15%, recall of 77%, and F-measure of 72.1%. These results confirm the potential of *MicroMiner* to automate the identification stage of microservices migration and provide accurate and modular service decomposition.

The candidate microservices identified in this stage serve as the foundation for the next steps of the migration process that include microservice packaging and deployment. In the following chapters, we address the challenge of transforming these candidates into independently deployable microservices by automating the following phases of the migration process that include microservice packaging and deployment. Chapter 3 introduces *MicroPacker*, an LLM–based approach that automates the packaging phase by code refactoring, API generation, and the integration of microservices design patterns.

The candidate microservices identified in this phase provide the foundation for the subsequent phases of the migration process, including packaging and deployment. In the following chapters, we automate these phases to transform the candidates into independently deployable microservices. Chapter 3 introduces *MicroPacker*, an LLM-based approach that automates the packaging phase through code refactoring, API generation, and the integration of microservice design patterns.

CHAPTER 3

MICROSERVICES PACKAGING PHASE

3.1 Introduction

As highlighted in our Systematic Literature Review (Chapter 1, Section 1.9), the packaging phase of microservices migration remains significantly underexplored. Most existing efforts concentrate on identifying service candidates but fall short of addressing the subsequent transformation into deployable, self-contained microservices.

This chapter addresses that gap by building on the microservice candidates produced in Chapter 2 through automated identification. These candidates serve as the foundation for restructuring the monolithic codebase into a set of independently deployable services.

To that end, we introduce *MicroPacker*, an LLM-based approach that automates the packaging phase. It encompasses code encapsulation, API generation, library separation, and the integration of essential runtime patterns such as configuration servers, service registries, and load balancers. The goal of *MicroPacker* is to reduce the manual effort required to restructure legacy code while ensuring that the resulting microservices are deployable and consistent with architectural best practices.

We posit that LLMs can effectively support this transformation-heavy and labour-intensive phase of migration. Specifically, we explore their capacity to reduce reliance on expert developers, streamline the migration process, and generate microservices that are both structurally sound and operationally functional. Our approach is grounded in Kent Beck's principle: *"make it work, make it right, make it fast."* In this work, we focus on making it work by ensuring functional completeness and making it right by aligning with microservices design principles. Performance optimisation (making it fast) is left for future exploration.

To evaluate the effectiveness of our approach, we assess it along three key dimensions:

- Completeness: Does *MicroPacker* generate all the expected artifacts for each packaging task, regardless of correctness?
- Correctness: How accurately does *MicroPacker* produce code that meets the intended objectives of each task?
- Operationality: How functional is the generated code in practice, and what level of manual adjustment is required to achieve full deployability?

Our evaluation across two real-world systems—PetClinic and FXML-POS—shows that *MicroPacker* achieves high performance across all three dimensions. By the third iteration, artifact-level completeness reached 100% in nearly all packaging tasks. Correctness, measured via precision, recall, and F1-score, consistently exceeded 0.85, with several core artifacts achieving perfect scores. Moreover, over 80% of the generated codebase was operational with minimal manual intervention. These results demonstrate that *MicroPacker* can effectively automate the packaging phase, producing structurally correct and deployable microservices with limited human effort.

The remainder of this chapter is structured as follows. Section 3.2 presents the prompting strategy and automation workflow. Section 3.3 details the evaluation procedure. Section 3.4 reports the results, and Section 3.5 discusses the results and findings, and threats to validity. Finally, Section 3.6 concludes the chapter.

3.2 Packaging Approach: MicroPacker

In this work, we propose a systematic, tool-augmented approach to guide the migration of monolithic systems to microservices. The approach is designed to ensure a structured and iterative workflow, integrating feedback and automated validations to refine outputs and meet convergence criteria.

This section is divided into two subsections. Subsection 3.2.1, Approach Overview, provides a high-level description of the methodology and its structure. Subsection 3.2.2, Implementing the Approach, details how the methodology is applied, including the definition of prerequisites, inputs, and the initial prompts required for each task of the packaging phase.

3.2.1 Approach Overview

Figure 3.1 outlines our tool-augmented approach for migration, which is systematically applied to each task of the packaging phase. The approach is structured into two stages: *Prompt Development and Execution* and *Results Verification and Validation*.

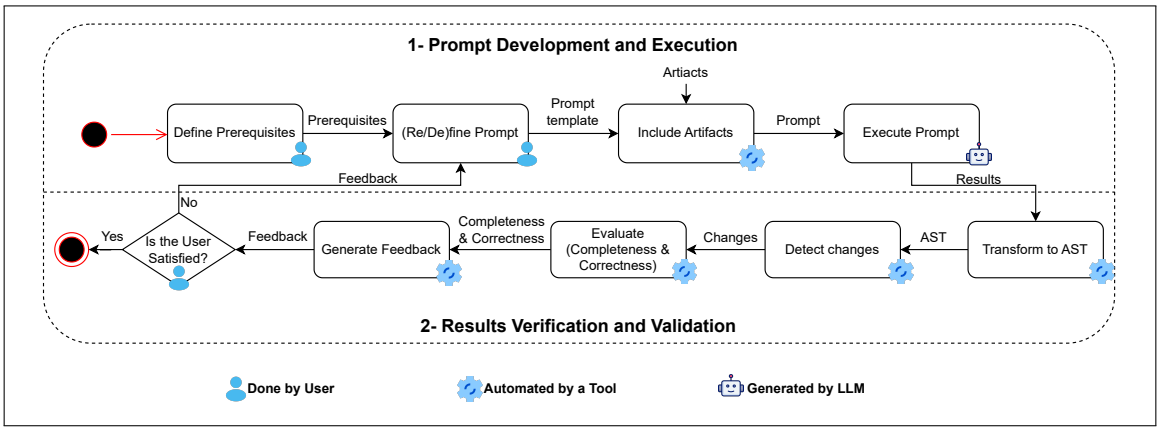

Figure 3.1 Two-stage tool-augmented workflow applied iteratively to each task of the packaging phase

3.2.1.1 stage 1: Prompt Development and Execution

This stage focuses on preparing and executing prompts that leverage LLMs to assist in specific tasks related to the migration process. It consists of the following steps:

1. **Define Prerequisites:** Identify and specify the foundational elements required for creating effective prompts, such as constraints, goals, steps, and high-level requirements.
2. **Define Prompt:** Formulate the initial prompt based on the defined prerequisites, ensuring it aligns with the migration objectives.
3. **Include Artifacts:** Provide the necessary contextual inputs, such as source code, call graphs, or dependency graphs, to support the execution of the prompt.
4. **Execute Prompt:** Run the prompt with the artifacts and prerequisites to generate outputs that address the migration tasks.

3.2.1.2 stage 2: Results Verification and Validation

This stage focuses on evaluating and validating the outputs generated in the first stage. It relies on user feedback and automated scripts to ensure the results align with predefined goals. The steps in this stage are:

1. **Transform to AST:** Convert the system's source code into an Abstract Syntax Tree (AST) for easier analysis of structural changes.
2. **Detect Changes:** Identify the differences between the original and transformed systems to ensure alignment with the migration objectives.
3. **Evaluate (Completeness and Correctness):** Assess whether the changes meet the defined criteria, including resolving all violations and adhering to constraints.
4. **Generate Feedback:** Provide detailed feedback to highlight areas of success and issues that need further refinement.
5. **Is the User Satisfied?** Allow the user to review the feedback and decide whether the results are satisfactory. If not, the process iterates with refined prompts to address unresolved issues.

3.2.2 Approach Implementation

In the following sections, we present the inputs required for each task of the packaging phase, which serve as the foundation for creating our prompts. We also detail the convergence criteria used to evaluate the output of these prompts.

Running Example

Through our analysis of the systems used for validation, we were unable to identify a subset that was both sufficiently simple for demonstration purposes and comprehensive enough to effectively illustrate the results of the approach on each step of the packaging phase. To address this, we designed a toy system consisting of three microservices: Order Management Service, Payment Processing Service, and Inventory Management Service. Each microservice represents a distinct

business function and demonstrates interactions such as method invocation, attribute access, object instantiation, and cross-service inheritance. Figure 3.2 illustrates the class diagram of the proposed system.

Microservice A: Order Management Service handles customer orders and their lifecycle. *Order*: Manages order details and status updates. *Customer*: Places orders and invokes payment processing.

Microservice B: Payment Processing Service that is responsible for processing payments and generating invoices. *Payment*: Handles payment authorization. *Invoice*: Inherits from *Order* to extend billing details.

Microservice C: Inventory Management Service manages stock and order fulfilment. *InventoryItem*: Tracks stock and updates inventory. FulfillmentService: Fulfils orders and processes payments.

Key Inter-Service Dependencies

Direct Method Invocation: *Customer* → *Payment* (tightly coupled).

Cross-Service Inheritance: *Invoice* extends *Order* (increased dependency).

Direct Attribute Access: *InventoryItem* → *Order* (encapsulation violation).

Direct Instantiation: *FulfillmentService* → *Payment* (tight coupling).

This toy system provides a simplified yet comprehensive model for evaluating migration challenges and architectural dependencies.

3.2.2.1 Task 1: Encapsulate Microservices

In order to encapsulate microservices effectively, we follow the same order of steps for resolving violations as proposed in Zaragoza *et al.* (2021). This structured approach ensures that key

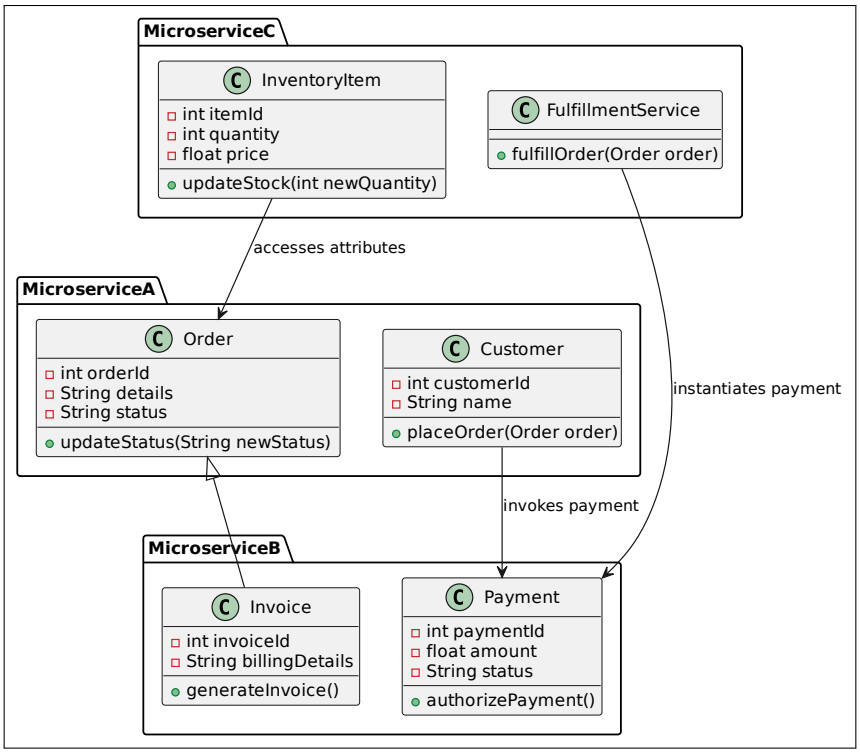

Figure 3.2 The class diagram of the running example

encapsulation challenges are systematically addressed.

3.2.2.1.1 Sub-task 1.1: Attribute Access Refactoring

This sub-task addresses violations where microservices directly access attributes defined in classes from other services. To enforce proper encapsulation, these direct accesses are replaced by method-based interactions using getter and setter methods. The refactoring is performed in two coordinated steps:

(1) Introduce getter and setter methods in the target class, keeping direct access temporarily to ensure backwards compatibility.

(2) Update all dependent classes to rely on these methods, then remove direct field access once updates are complete.

Prerequisites The goal is to replace direct attribute access between microservices with method-based interactions while preserving system behavior. The main constraint is to ensure backward compatibility. This is achieved by:

Artifacts The following artifacts:

Source Code: Full content of the target and dependent classes, retrieved using their fully qualified names.

Dependency List: A CSV file produced via AST analysis, listing (source class, variable, attribute, target class) pairs along with the associated microservice.

Prompt Task 1.1.1: Introduce Getter and Setter Methods

Listing 3.1: Introduce Getters/Setters in Target Class

```
You are a software refactoring assistant. Your task is to encapsulate the
    internal state of the class <Target_Class>
by introducing getter and setter methods for the following fields: <field_1>,
    <field_2>, ...

Instructions:
1. Define a public getter method that returns the value of each field.
2. Define a public setter method that assigns a value to each field
3. Keep the original field access temporarily to ensure backward compatibility.

4. Return the updated class code with the new methods clearly included.

Purpose:
This refactoring prepares the class for safe access by external services
    through encapsulated interfaces.
```

Prompt Task 1.1.2: Update Dependent Classes

Listing 3.2: Update Callers to Use Getter/Setter

```
You are updating the class <Source_Class> to follow encapsulation best
   practices when interacting with <Target_Class>.

Instructions:
1. Locate all direct accesses to fields in <Target_Class> instances (e.g., obj.
   field).
2. Replace reads with calls to the corresponding getter methods (e.g., obj.
   getField()).
3. Replace write operations with calls to the appropriate setter methods.
4. Ensure that no direct field access remains.

Note:
This change depends on the availability of getter and setter methods
   introduced in <Target_Class>.
```

Evaluation Validation is performed using AST-based scripts that verify the following:

- Getter and setter methods have been correctly defined .

- Source classes no longer contain direct field accesses across microservices.

- Method calls are correctly formed and correspond to the accessed attributes.

Feedback this feedback is generated automatically. It includes:

- Missing getter or setter methods for required attributes.

- Remaining direct accesses in dependent classes.

- Method name mismatches between defined and invoked accessors.

3.2.2.1.2 Sub-task 1.2: Inheritance Refactoring

This sub-task addresses violations where a class in one microservice inherits from a class in another, breaking encapsulation. To preserve inheritance semantics while decoupling microservices, we decompose inheritance into three mechanisms—definition extension, subtyping, and polymorphism—and refactor each using dedicated transformation strategies:

(1) Introduce a double proxy pattern to preserve parent–child behavior without direct inheritance.

(2) Recreate subtyping using interface inheritance, enabling loose coupling and contract-based design.

(3) Reestablish polymorphic assignment through interface-based references instead of concrete class instantiations.

Prerequisites The prerequisites for this refactoring include: Decouple inheritance across microservices while preserving behavioral correctness.

Artifacts The following artifacts are used to support the inheritance refactoring process:

Source Code: Full content of the parent and child classes, loaded using their fully qualified names.

Inheritance Dependencies: A CSV file listing (child class, parent class) pairs spanning microservices, extracted via AST analysis.

Prompt Task 1.2.1: Introduce Double Proxy Pattern

Listing 3.3: Introduce Double Proxy Pattern

```
You are a software refactoring assistant. Your task is to decouple the
    inheritance link between <Child_Class> and <Parent_Class>
by introducing a double proxy pattern.

Instructions:
1. Create a proxy class <Parent_Class>Proxy that implements the behavior of <
    Parent_Class>.
2. Modify <Child_Class> to extend <Parent_Class>Proxy.
3. Create another proxy, <Child_Class>Proxy, that extends <Parent_Class> and
    delegates to <Child_Class>.

Purpose:
This pattern avoids direct inheritance across microservices and supports
    dynamic delegation.
```

Prompt Task 1.2.2: Recreate Subtyping via Interface Inheritance

Listing 3.4: Recreate Subtyping with Interface Inheritance

```
You are a software refactoring assistant. Your task is to reestablish
    subtyping between <Child_Class> and <Parent_Class> by introducing interface
     inheritance.
Instructions:
1. Create an interface I<Parent_Class> that declares the public methods of <
    Parent_Class>.
2. Update <Child_Class> to implement this interface.
3. Ensure that subtyping relationships now use the interface abstraction
    rather than concrete types.
Purpose:
This supports flexible and loosely coupled contracts between microservices.
```

Prompt Task 1.2.3: Reestablish Polymorphic Assignment

Listing 3.5: Polymorphic Assignment via Interface

```
You are a software refactoring assistant. Your task is to ensure polymorphism
    is preserved
by using interface-based references.

Instructions:
1. Refactor assignments such that <Child_Class> instances are assigned to I<
    Parent_Class> variables.
2. Avoid use of concrete class names in polymorphic references.

Purpose:
This preserves dynamic dispatch without violating microservice boundaries.
```

Evaluation Validation is conducted via AST-based rules that confirm:

- Proxy classes have been defined and replace direct ‘extends‘ links.

- Interfaces have been implemented correctly by child classes.

- Polymorphic assignments rely on interface references, not concrete class instantiations.

Feedback Automatic feedback includes:

- Missing proxy class or interface definition.

- Incorrect or missing inheritance declarations.

- Use of concrete classes in polymorphic assignments instead of interfaces.

3.2.2.1.3 Sub-task 1.3: Class Reference Refactoring

This sub-task addresses violations where a class in one microservice directly instantiates a class from another, leading to tight coupling and reduced modularity. To enforce proper encapsulation and separation of concerns, we apply two design patterns:

(1) Introduce the Factory Pattern to centralise object instantiation and decouple client code from concrete implementations.

(2) Introduce the Proxy Pattern to control access and interactions between classes across microservice boundaries.

Prerequisites The prerequisites for this refactoring include: Eliminate direct object instantiation across microservices and introduce controlled delegation.

Artifacts The following artifacts are used to support the class reference refactoring process:

Source Code: Full content of both the source and referenced classes, accessed using their fully qualified names.

Class Instantiation Dependencies: A CSV file listing (source class, target class) pairs where instantiation occurs across microservices, extracted via AST analysis.

Prompt Task 1.3.1: Introduce Factory Pattern

```
You are a software refactoring assistant. Your task is to decouple direct
    instantiation of <Target_Class>
in <Source_Class> using the Factory Pattern.

Instructions:
1. Define a factory class <Target_Class>Factory that provides a public method
    to return instances of <Target_Class>.
2. Refactor <Source_Class> to call this factory method instead of using `new <
    Target_Class>()`.
3. Ensure the factory is reusable and encapsulates creation logic.

Purpose:
This pattern centralizes instantiation and decouples source classes from
    direct dependencies.
```

Prompt Task 1.3.2: Introduce Proxy Pattern for Access Control

```
You are a software refactoring assistant. Your task is to introduce the Proxy
    Pattern to mediate access between <Source_Class> and <Target_Class>.

Instructions:
1. Define a proxy class <Target_Class>Proxy that wraps or delegates calls to <
    Target_Class>.
2. Replace direct calls to <Target_Class> in <Source_Class> with calls to <
    Target_Class>Proxy.
3. Ensure that all method interactions are routed through the proxy.

Purpose:
This enforces encapsulation and enables controlled interactions between
    microservices.
```

Evaluation Validation is conducted using AST-based scripts that verify the following:

- No remaining *new* expressions referencing target classes from other microservices.

- Factory classes are defined and referenced in place of direct instantiation.

- Proxy classes mediate access to cross-microservice objects, and the original references are replaced.

Feedback Automatically generated feedback may include:

- Factory class missing or not used in place of *new*.

- Proxy class not defined or improperly injected.

- Unresolved instantiation still occurring directly across microservices.

Running example

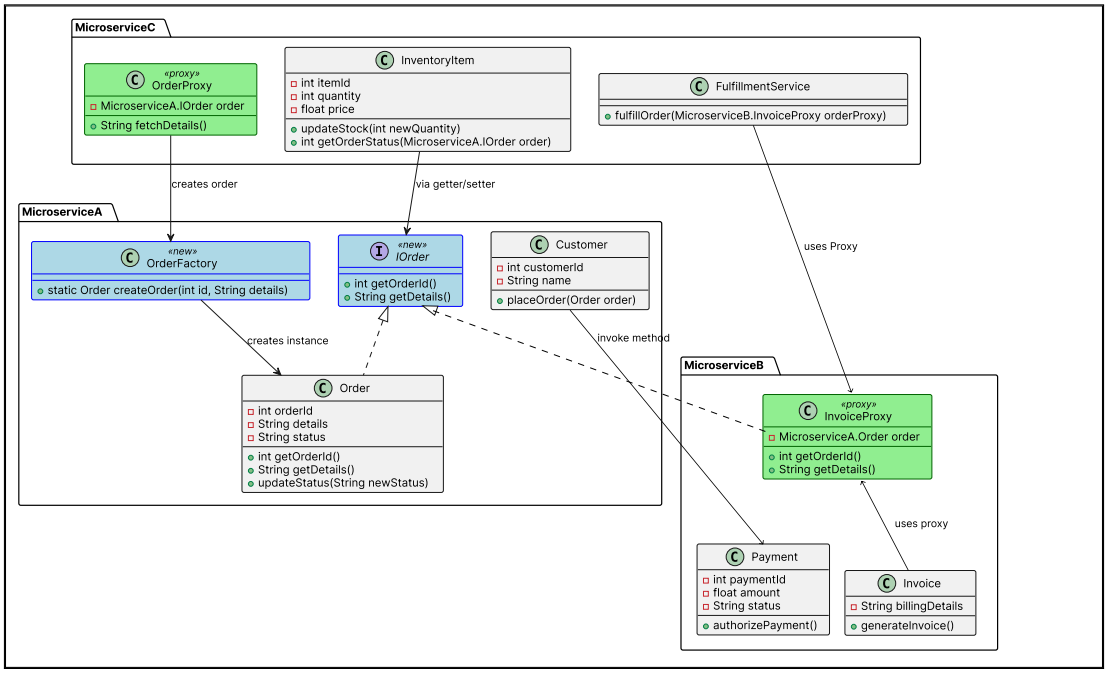

Figure 3.3 The class diagram after encapsulating microservices

The changes made to the running example 3.3 focus on improving **encapsulation** between microservices. In **MicroserviceA**, the *Order* class was refactored to implement the *IOrder* interface, providing getter methods for accessing the *orderId* and *details* attributes. This replaces direct attribute access with method calls, ensuring that attribute values are encapsulated within the *Order* class. Additionally, a new *OrderFactory* class was introduced to control the instantiation of *Order* objects, further enforcing encapsulation by centralising object creation.

In **MicroserviceB**, the *Invoice* class was restructured to use the **Proxy pattern**. Instead of directly inheriting from *Order*, *Invoice* now relies on the *InvoiceProxy*, which implements the *IOrder* interface. This change allows *Invoice* to access the *Order* details through getter methods, decoupling it from the direct attribute access of *Order*. This also ensures that *Invoice* no longer directly manipulates *Order* attributes,.

In **MicroserviceC**, the *InventoryItem* and *FulfillmentService* classes were updated to access *Order* attributes using getter methods from the *IOrder* interface rather than directly accessing *Order* attributes. The *InventoryItem* class, which once accessed *Order* attributes directly, now uses the encapsulated getter methods to fetch *Order* data. Additionally, the *FulfillmentService* now interacts with the *InvoiceProxy* instead of directly accessing *Invoice* data, ensuring that all inter-service communication respects the new encapsulation structure. The introduction of the **Proxy pattern** across microservices, such as in *InvoiceProxy* and *OrderProxy*, further reduces tight coupling between microservices. These proxies act as intermediaries, ensuring that microservices interact through well-defined interfaces and getter/setter methods, rather than direct attribute access. This improves the overall **modularity** and **maintainability** of the system, making it more flexible to future changes and easier to test in isolation. These changes also lay the groundwork for better scalability, as services can evolve independently without breaking interactions with other services.

3.2.2.2 Task 2: Establishing Microservices API

We divided the process of establishing microservices APIs into two critical subtasks to ensure that the communication between services is properly defined and that services can interact with each other in a consistent, standardised manner.

3.2.2.2.1 Sub-task 2.1: Generate Endpoints

This sub-task ensures that all methods invoked by other microservices are exposed through standardised APIs. To enable modular communication, each externally called method is wrapped as a publicly accessible endpoint using a consistent protocol (e.g., REST). The transformation introduces endpoint definitions without altering the internal logic, preserving backwards compatibility.

Prerequisites Make all externally called methods accessible via standardised API endpoints. Each microservice must define clear API interfaces, and all externally called methods must be reachable through protocol-compliant, properly documented endpoints.

Artifacts The following artifacts drive this refactoring:

Source Code: Full implementation of each microservice, parsed to extract externally called methods.

External Method List: A CSV listing cross-microservice method calls, obtained through AST-based analysis.

API Specification: Defines the communication protocol (e.g., REST), naming conventions (e.g., */api/resource/{id}*), and data format rules (e.g., query parameters for GET, JSON body for POST).

Prompt Task 2.1.1: Generate API Endpoints

Listing 3.6: Generate API Endpoints

```
You are an API refactoring assistant. Your task is to generate standardized
    API endpoints for the provided microservice code, ensuring that all
    externally called methods are exposed correctly.

Instructions:
1. Identify the externally called methods from the external_methods list.
2. Define endpoints for each method using the provided API specification.
3. Apply the correct communication protocol (e.g., REST) and structure the
    endpoints accordingly.
4. Preserve the internal logic and wrap each method call inside an endpoint
    without altering its behavior.

Inputs:
- Source Code of the microservice: {source_code}
- List of externally called methods: {external_methods}
- API Specification: {api_spec}

Expected Output:
- Updated source code with annotated endpoints for all externally called
    methods.
- Structured list of generated endpoints including paths, HTTP methods, and
    data formats.
```

Evaluation Validation is performed through AST analysis to check:

- Whether all externally called methods are exposed via public endpoints.

- Whether each endpoint complies with the specified HTTP method, naming format, and data handling convention.

- Whether any unauthorised internal method was exposed.

Feedback If validation fails, feedback includes:

- *Missing Endpoints:* Some externally called methods were not exposed.

- *Protocol Violations:* Misuse of HTTP verbs (e.g., GET with body) or incorrect gRPC service definitions.

- *Data Format Errors:* Incorrect serialisation (e.g., XML used instead of JSON) or misplaced input (e.g., body content in GET).

3.2.2.2.2 Sub-task 2.2: Replace Direct Dependencies with Service Calls

In this sub-task each direct call between services must be refactored into a communication through an API.

Prerequisites The goal is to replace all direct dependencies between microservices with service calls that respect communication protocols.

Artifacts The sub-task relies on:

Source Code: Full source code of both the calling and the called classes, retrieved using fully qualified names.

External Calls CSV: A list of (source class, target class, method name) entries where a method of one microservice is invoked from another.

Prompt Task 2.2: Replace Direct Dependencies with Service Calls

Listing 3.7: Replace Direct Dependencies with Service Calls

```
You are a software refactoring assistant. Your task is to replace direct
   method calls between classes from different microservices with remote
   service calls, following proper API usage conventions.

Instructions:
1. In class <Source_Class>, locate all direct method invocations to <
   Target_Class> methods.
2. Replace these calls with service calls using REST or gRPC endpoints.
3. Respect data formatting and communication protocol guidelines.
4. Do not alter internal logic in <Target_Class>.
5. Return the updated code of <Source_Class> with the service calls properly
   integrated.

Input:
- Source Class Code: {caller_class_code}
- Target Class Code: {called_class_code}
- List of Called Methods: {method_list}

Expected Output:
- Refactored source code of <Source_Class> using API-based service calls
   instead of direct invocations.
```

Evaluation To validate the transformation, we perform two checks using AST analysis:

- We verify that no direct method calls from Source Class to Target Class remain.

- We confirm that a valid HTTP or remote call pattern is introduced (e.g., ‘RestTemplate‘, ‘WebClient‘, ‘HttpClient‘, or gRPC stubs).

Feedback If violations remain, automated feedback identifies:

- *Direct call still used:* The method is still invoked directly from the target class.

- *No protocol usage found:* No evidence of HTTP or gRPC usage is detected in the refactored code.

- *Source class not found:* Source class is missing or cannot be parsed due to read errors.

3.2.2.2.3 Running example

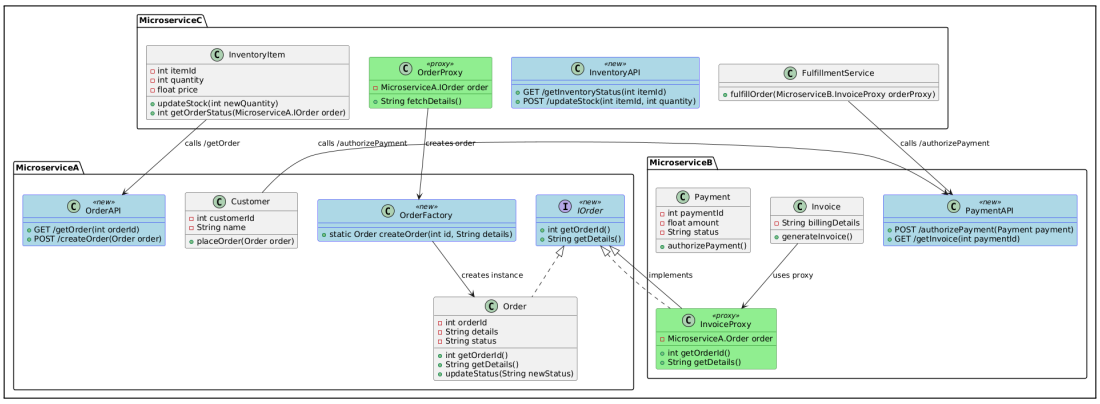

Figure 3.4 The class diagram after establishing Microservices API

In the updated example 3.4, the direct method calls between microservices are replaced with service calls through API endpoints, ensuring proper service decoupling. In **Microservice A** (Order Management Service), the internal methods for order management, such as retrieving order details and creating orders, are now exposed via the *OrderAPI* class with endpoints like *GET /getOrder* and *POST /createOrder*. **Microservice B** (Payment Processing Service) introduces the *PaymentAPI* class, which exposes methods like *POST /authorizePayment* and *GET /getInvoice* to interact with payment data. Additionally, **Microservice C** (Inventory Management Service) now interacts with the *OrderAPI* to retrieve order information and uses the *PaymentAPI* to authorise payments. The *InvoiceProxy* and *OrderProxy* classes act as intermediaries, allowing services to interact with one another through proxies instead of direct method calls. These

changes reflect the transition from direct calls to communication through well-defined API endpoints.

3.2.2.3 Task 3: Separate Libraries and Build Files

This task ensures that each microservice is associated with an independent and minimal build configuration, containing only the libraries it effectively uses. g., *pom.xml*, *build.gradle*) for each service.

Prerequisites The refactoring goal is to decouple microservices from the original monolithic configuration by generating self-contained build files that reflect actual library usage. The main constraint is to ensure correctness and completeness while avoiding redundant or unused dependencies.

Artifacts This task relies on the following inputs:

Source Code: The full source code of each microservice, grouped by the class-to-microservice mapping.

Monolithic Build File: The original system's build descriptor (e.g., *pom.xml*), which defines all possible dependencies and versions.

Prompt Task 3.2: Generate Build File Per Microservice

Listing 3.8: Generate Build File for a Microservice

```
You are a software refactoring assistant. Your task is to generate a build
   configuration file for the microservice <Microservice>
based on its implementation code and the original monolithic build descriptor.

Context:
- The original system was configured using the following build file:
<Monolithic_Build_File>
```

```
Instructions:
1. Analyze the following source code to detect all required libraries,
   including static imports and dynamic usage.
2. Derive a clean and minimal build configuration file (e.g., pom.xml, build.
   gradle, etc.).
3. Ensure all declared dependencies are complete (with groupId, artifactId,
   version) and relevant to this microservice only.
4. Ignore unused or redundant libraries. Maintain compatibility with the
   original build system.

Source Code:
<All_Classes_Code>
```

Evaluation Validation is performed by comparing the imports found in the source code with the libraries declared in the generated build file. Specifically:

- All imported libraries must be declared as dependencies.

- No dependency should be declared without corresponding usage.

- No cross-microservice or illegal shared library should be referenced unless explicitly configured as shared.

Feedback Automated feedback reports the following types of issues:

- *Missing libraries:* Import paths found in code but not present in the build configuration.

- *Incorrect or redundant dependencies:* Declared libraries not used by any class in the service.

- *Missing build file:* No configuration file was generated for the microservice.

This sub-task ensures each microservice is buildable and independently deployable while preventing library leakage and excessive coupling.

3.2.2.4 Task 4: Integrate Pattern

We divided this task into sequential subtasks to integrate essential runtime capabilities for scalable and resilient microservice architectures: centralised configuration, dynamic service discovery, and load balancing. These patterns are widely recognised as foundational to enabling flexibility, consistency, and fault tolerance in microservice deployments (Dragoni *et al.*, 2017). While other runtime patterns, such as circuit breakers, distributed tracing, and service mesh, also play significant roles, we focus on these three due to their role in enabling basic operational autonomy and inter-service communication.

3.2.2.4.1 Sub-task 4.1: Config Server Integration

This sub-task externalises microservice configurations by integrating a centralised configuration server. Each microservice will store its config in a remote repository and dynamically retrieve values at startup.

Prerequisites The goal is to decouple hardcoded configurations and enable centralised management. Validation requires each service to reference a remote config and remove in-code literals.

Artifacts The sub-task relies on:

Source Code: Full implementation of each microservice.

-*Config Server URI:* http://localhost:8888

-*Central Config Files:* External *.yml* files per service containing key-value properties.

-*bootstrap.yml:* Startup file linking the service to the config server.

Prompt The configuration server integration process is divided into three distinct prompt tasks to address the separation of concerns and ensure modular guidance for the LLM:

- **Prompt 4.1.0 (Config Server Implementation):** This prompt focuses on setting up the configuration server itself, which acts as the central authority for managing and distributing configuration files to all microservices. By isolating this task, the LLM can generate a dedicated standalone server without being influenced by specific microservice logic.
- **Prompt 4.1.1 (Externalising Configuration):** This prompt targets each microservice individually, guiding the extraction of hardcoded configuration values (e.g., port numbers, DB URLs) into external YAML files. It ensures a clean separation between code and configuration, improving maintainability.
- **Prompt 4.1.2 (Connecting to Config Server):** Once external configuration is created, this prompt instructs the microservice to connect to the centralised server at runtime via ‘bootstrap.yml‘ (or equivalent). It focuses on wiring up the integration without modifying internal logic or re-extracting config values.

Prompt Task 4.1.0: Implement a Configuration Server

Listing 3.9: Create Config Server

```
You are a system integrator. Your task is to implement a centralized
    configuration server
that serves config files to microservices at runtime.

Requirements:
1. Create a configuration server that listens on port 8888.
2. Serve YAML or properties files from a local directory or Git repository.
3. Map filenames to service names.
4. Support updates without restart if possible.

Expected Output:
- Running server on port 8888
- Config directory with one file per service
- API accessible by service name
```

Prompt Task 4.1.1: Externalise Configuration Values

Listing 3.10: Generate Config Files for External Repository

```
You are a software assistant. Your task is to externalize configuration for
    this microservice
by creating a YAML file stored in a centralized config server.

Requirements:
1. Extract all hardcoded values (ports, credentials, etc.) from the source
    code.
2. Replace them with placeholders (e.g., '${config.key}').
3. Generate a file named '{service-name}.yml' with the corresponding values.

Input:
- Microservice Source Code
- Config Server URI: http://localhost:8888

Expected Output:
- External config file with key-value pairs
- Refactored code using placeholders
```

Prompt Task 4.1.2: Connect Microservice to Config Server

Listing 3.11: Connect Microservice to Config Server

```
You are a software refactoring assistant. Your task is to configure the
    microservice
to retrieve its configuration from a central config server.

Requirements:
1. Create a 'bootstrap.yml' specifying:
   - 'spring.application.name: <service-name>'
   - 'spring.config.import: configserver:http://localhost:8888'
```

```
2. Ensure config is loaded on startup.
3. Add dependencies if required.

Expected Output:
- bootstrap.yml created with correct values
- Updated service code connected to config server
```

Evaluation Static validation checks:

- All placeholders used in code exist in the corresponding config file.

- No hardcoded literals remain.

- *bootstrap.yml* is present and correctly structured.

Feedback Reports:

- Missing or unresolved placeholders

- Missing or invalid *bootstrap.yml*

- Service not properly connected to config server

3.2.2.4.2 Sub-task 4.2: Service Registry Integration

This sub-task enables runtime service discovery by integrating a centralised service registry. Each microservice registers itself at startup and dynamically discovers other services without relying on hardcoded URLs.

Prerequisites: This allows services to register with a central registry and discover peers dynamically. We require proper registry configuration and the elimination of hardcoded communication links.

Artifacts: The sub-task relies on:

- Service Registry Tool (*registry_tool*): e.g., Eureka, Consul, Zookeeper.

- Discovery Strategy (*discovery_client*): Feign, LoadBalanced RestTemplate, etc.

- Communication Style (*comm_style*): REST, gRPC, messaging.

- Microservice Source Code (*source_code*)

- Known Microservices (*known_services*): List of all microservices for Feign validation.

- Configuration Files (*config_files*): application.yml or bootstrap.yml.

Prompt Task 4.2.1: Configure Service Registration

Listing 3.12: Configure Microservice Registration with Service Registry

```
You are a software refactoring assistant. Your task is to configure a
    microservice to register itself with a service registry.

Requirements:
1. Add the required discovery client dependency.
2. Define the service name and registry URI in application.yml or bootstrap.
    yml.
3. Annotate the application if necessary (e.g., @EnableDiscoveryClient).

Input:
- Registry Tool: {registry_tool}
- Configuration Files
- Source Code

Expected Output:
- application.yml or bootstrap.yml with proper configuration.
- Updated project with necessary dependencies and annotations.
```

Evaluate: Static validation is performed by a script that:

- Parses configuration files to check for registry URI and service name.

- Reports missing or invalid registry settings.

- **Feedback:** - Missing dependency or registry URI.

- Misconfigured service name or unrecognised registry endpoint.

- Proper configuration and discovery annotations detected.

Prompt Task 4.2.2: Enable Discovery-based Communication

Listing 3.13: Enable Service Discovery in Inter-Service Communication

```
You are a software assistant. Your task is to enable dynamic service discovery
    in a microservice and remove all hardcoded inter-service URLs.

Requirements:
1. Identify HTTP calls with hardcoded URLs.
2. Replace them with Feign clients or LoadBalanced RestTemplates using service
    names.
3. Ensure service names correspond to known registered services.

Input:
- Source Code
- Known Microservices
- Communication Style

Expected Output:
- Updated source code using discovery-based communication.
- Hardcoded service URLs replaced by logical service identifiers.
```

Evaluate: Static validation is performed using a script that:

- Scans for hardcoded service URLs.

- Detects and analyzes *FeignClient* usage.

- Cross-checks ‘name‘ attributes with known microservices.

Feedback:

- Hardcoded URLs detected (e.g., "http://localhost:8081/...").

- Feign client defined with unknown or inconsistent service name.

- Valid Feign client configuration and no static URLs found.

3.2.2.4.3 Sub-task 4.3: Load Balancer Integration

In this subtask, we integrate client-side load balancing to distribute requests across service instances dynamically and improve fault tolerance.

Prerequisites: It requires detection of configured load-balancer settings and the elimination of hardcoded service calls.

Artifacts: The sub-task relies on:

- Load Balancer Library (*lb_tool*): e.g., Spring Cloud LoadBalancer, Netflix Ribbon.

- Discovery Client Setup (*discovery_client*): must be active for dynamic instance resolution.

- Source Code of Service Caller (*source_code*): microservice that initiates outbound requests.

- Target Services List (*targets*): microservices that require load-balanced access.

- Application Configuration Files (*config*): e.g., application.yml, bootstrap.yml.

Prompt Task 4.3.1: Add Load Balancer Configuration

Listing 3.14: Add Load Balancer Configuration

```
You are a software refactoring assistant. Your task is to configure a
   microservice for load-balanced service consumption.

Requirements:
1. Add the required dependency for client-side load balancing.
2. Configure the load balancing strategy (e.g., round-robin) in the
   configuration file.
3. Integrate the load balancer with the service registry.

Input:
- Load Balancer Tool: {lb_tool}
- Discovery Client: {discovery_client}
- Configuration File: {config}

Expected Output:
- application.yml or equivalent file updated with load balancing strategy.
- Dependencies or annotations to enable the selected load balancer.
```

Evaluate: Static validation is performed by a script that:

- Parses the configuration file for the presence of load balancing strategy or load balancer libraries.

- Confirms whether the configuration integrates with service discovery.

Feedback: - No load balancing configuration found in config files.

- Configuration present but missing dependencies or annotations.

- Registry integration detected but no load balancing policy applied.

- Valid configuration for load balancing and discovery integration.

Prompt Task 4.3.2: Refactor Code to Use Load-Balanced Clients

Listing 3.15: Refactor Service Calls to Use Load Balancer

```
You are a software assistant. Your task is to refactor service calls in a
   microservice to use client-side load balancing.

Requirements:
1. Replace all hardcoded service URLs with logical service names.
2. Use load-balanced clients like @LoadBalanced RestTemplate or Feign.
3. Ensure the retry or fallback behavior is defined (if supported).

Input:
- Source Code of the Calling Microservice: {source_code}
- Target Services List: {targets}
- Load Balancer Tool: {lb_tool}

Expected Output:
- Updated service calls with discovery-based, load-balanced logic.
- All static URLs removed and replaced by logical service references.
```

Evaluate: Static validation is performed using a script that:

- Detects hardcoded URLs (e.g., ‘http://localhost‘, raw IPs).

- Checks for the presence of ‘@LoadBalanced‘ or ‘@FeignClient‘.

- Flags missing retry or fallback configurations (optional).

Feedback:

- Static service URLs found (e.g., "http://localhost:8082/...").

- Load-balanced clients not used or misconfigured.

- Retry or fallback strategies not present.

- Code uses load-balanced discovery clients with dynamic resolution.

3.2.2.4.4 Running example

In the updated example illustrated in Figure 3.5, the system integrates essential runtime patterns to support scalable and resilient communication between microservices. In **Microservice A** (Order Management Service), configuration settings such as ports and credentials are externalized and stored in a central repository accessed via a Config Server. The microservice's code has been updated to load configuration dynamically using placeholders, ensuring consistency across environments. **Microservice B** (Payment Processing Service) and **Microservice C** (Inventory Management Service) follow the same approach, loading their configuration from the centralised server at startup through dedicated *bootstrap.yml* files. Additionally, all services are now registered with a service registry (e.g., Eureka), allowing them to dynamically discover peers using logical service names instead of hardcoded URLs. For instance, **Microservice C** communicates with the Payment and Order services via discovery-based clients, enabling runtime flexibility and eliminating static endpoint dependencies. To further enhance fault tolerance and scalability, client-side load balancing is introduced in Microservice C. When invoking downstream services, requests are now distributed across multiple instances using a round-robin strategy configured via Spring Cloud LoadBalancer. These runtime integrations collectively establish a foundation for autonomous, resilient microservice execution with consistent configuration, discoverability, and balanced request handling.

3.3 Study design

This section outlines the design of our empirical study conducted to evaluate our approach, MicroPacker. The evaluation focuses on three key aspects of the generated microservices: their

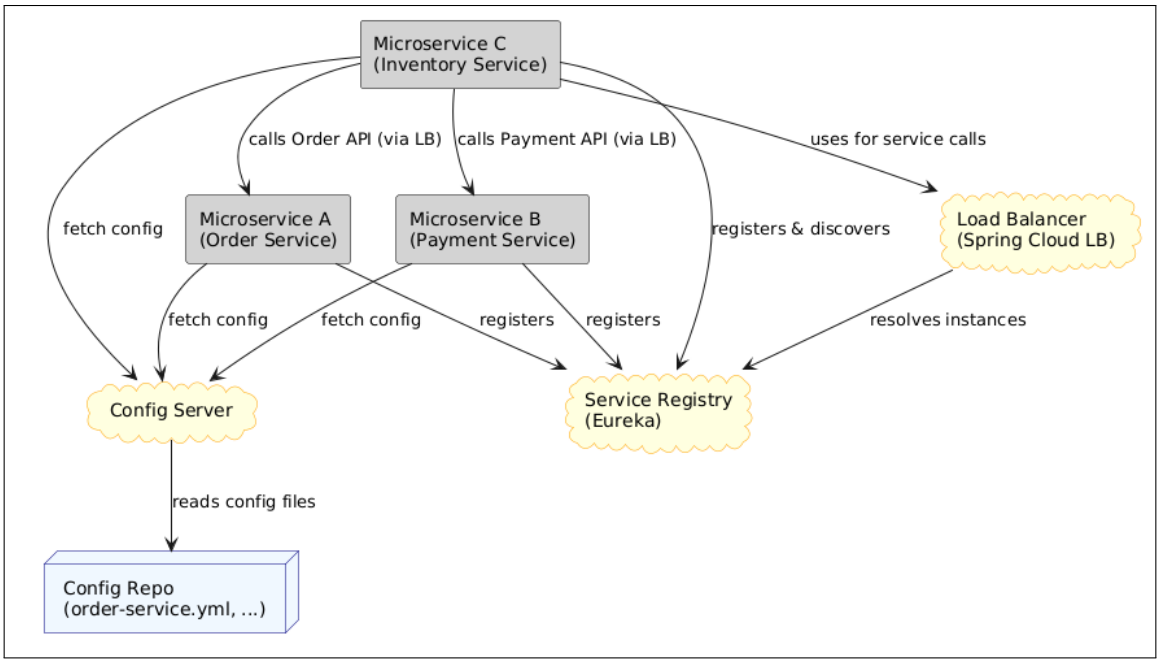

Figure 3.5 Integration of configuration server, service registry, and load balancing

completeness, *correctness*, and *operational readiness* across the various tasks involved in the packaging phase.

3.3.1 Case studies

To evaluate our approach, we selected two open-source Java-based systems: PetClinic[1] and FXML-POS[2]. These systems were chosen based on the following criteria: 1) Java-Based: Both systems are implemented in Java, ensuring compatibility with our analysis tools. 2) Monolithic Architecture: Both PetClinic and FXML-POS adopt monolithic architectures, being self-contained Java applications deployable as single JAR/WAR files with no distributed components or microservices. 3) Executable and Functional: The systems are fully executable, ensuring that our analysis is grounded in real-world, operational software.

[1] https://github.com/spring-projects/spring-petclinic

[2] https://github.com/sadatrafsanjani/JavaFX-Point-of-Sales

3.3.1.1 PetClinic

PetClinic is an open-source veterinary clinic management system with 36 classes, built using Spring Boot. Although Spring Boot supports modularity, the original PetClinic is a single-tier application. We selected it because it represents a real-world domain with clear functional modules (pet, owner, and visit management). We divide this system into 7 microservices.

3.3.1.2 FXML-POS

FXML-POS is an open-source inventory management system designed for enterprise resource planning (ERP). It includes 52 classes and provides functionalities such as purchase management, sales tracking, and supplier management. We divide this system into 9 microservices.

3.3.2 Evaluation Metrics

3.3.2.1 Completeness

We assess whether the generated outputs cover all the expected artifacts. Completeness is measured along two dimensions:

3.3.2.1.1 Artifact-Level Completeness

Before assessing completeness, we define the expected artifacts for each task of the packaging phase. A task is considered complete only if all associated artifacts are successfully generated. As shown in Table 3.1, each sub-task is associated with a set of expected artifacts. The table also reports the number of classes involved in producing these artifacts for both the PetClinic and FXML-POS systems, thus providing a quantitative basis for completeness evaluation.

Table 3.1 Expected Number of Artifacts and Involved Classes for PetClinic and FXML-POS

Task	Sub-task	Artifact	PetClinic		FXML-POS	
			Artifacts	Classes/Files	Artifacts	Classes/Files
1. Encapsulate	1.1	Getter and Setter Methods	2	2	4	3
		Updated Method Calls Using Accessors	2	2	7	5
	1.2	Proxy Classes	1	1	1	1
		Interface Definitions	1	1	1	1
		Polymorphic Assignments via Interfaces	1	1	2	2
	1.3	Factory Classes	4	3	10	6
		Proxy Classes for Instantiation	4	3	10	6
2. API	2.1	API Endpoints	13	6	15	7
	2.2	Service Call Implementations	42	15	29	12
3. Libraries	3.1	Build File For each Microservice	7	7	9	9
4. Runtime Patterns	4.1	External Configuration File	1	1	1	1
		Placeholder-Based Injection in Code	7	7	9	9
		Connection Setup (bootstrap.yml)	1	1	1	1
	4.2	Service Registration Configuration	1	1	1	1
		Discovery-Enabled Clients	7	7	9	9
	4.3	Load Balancing Configuration	1	1	1	1
		Client Integration	7	7	9	9

We evaluate whether each task of the packaging phase has produced all required artifacts. The evaluation is performed over multiple iterations to capture progressive improvements through prompt refinement. The metric is expressed as an *artifact coverage rate*, calculated as:

$$\text{Artifact Coverage Rate (\%)} = \frac{\text{\# Correctly Generated Artifacts}}{\text{\# Expected Artifacts}} \times 100 \tag{3.1}$$

A task is marked as fully complete only when all expected artifacts are present and conform to the required format. Results are reported in Table 3.2.

3.3.2.1.2 Class-Level Completeness

We further evaluate completeness at the class level to assess how comprehensively the transformations are applied across the system. Each class is categorised as:

- **Full**: All expected artifacts for the class are present.
- **Partial**: Some expected artifacts are missing.
- **Failed**: No critical artifacts were generated.

We compute the *class-level coverage rate* by measuring the proportion of classes that fall into each category. This metric highlights transformation coverage across the codebase. Results are presented in Table 3.3.

3.3.2.2 Correctness

To evaluate the correctness of the packaging, we examine the structural and syntactic correctness of the code produced by *MicroPacker* during the packaging phase. We use the same artifacts defined in Table 3.1 and apply correctness assessment along two dimensions.

3.3.2.2.1 Artifact-Level Correctness

We assess whether each generated artifact conforms to the structural expectations defined by its transformation pattern. Instead of a binary evaluation, we quantify correctness based on the number of structural or semantic violations per artifact. Each artifact is labeled as:

- **Correct**: The artifact fully implements the intended pattern with no structural or semantic violations (e.g., all required methods defined, correct signatures, proper integration).
- **Partially Correct**: The artifact implements the pattern but contains minor issues that do not fundamentally break functionality (e.g., naming mismatches, incomplete delegation, redundant code).
- **Incorrect**: The artifact fails to implement the pattern in a usable way, introduces major structural errors, or is entirely missing.

For evaluation purposes, correct and partially correct artifacts are treated as true positives (TP), while incorrect ones are considered false positives (FP) if generated but flawed, and false negatives (FN) if missing entirely. We compute:

$$\text{Precision} = \frac{TP}{TP + FP} \quad , \quad \text{Recall} = \frac{TP}{TP + FN} \tag{3.2}$$

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.3}$$

These metrics are computed per task and iteration, as reported in Table 3.4.

3.3.2.2.2 Class-Level Accuracy

At the class level, we evaluate whether all transformations applied to a class are structurally valid. Each class is categorised as:

- **Correct**: All artifacts expected for the class are correct.
- **Partial**: The class contains a mix of correct and partially correct artifacts, or at least one artifact is partially correct but none are incorrect.
- **Incorrect**: The class contains at least one incorrect artifact, or the majority of expected artifacts are missing.

We use the same precision, recall, and F1-score formulation at the class level, treating each correctly or partially transformed class as a true positive. Results are shown in Table 3.5.

3.3.2.3 Operationality

To evaluate operationality, we assess the manual effort required to make the generated system fully functional.

3.3.2.3.1 Manual Effort Metrics

We measure two indicators of manual effort:

- **Lines Added**: Number of new lines manually introduced to complete functionality.
- **Lines Changed**: Number of existing lines manually modified.

These values are extracted via a diff-based analysis between the generated and finalised system.

3.3.2.3.2 Effort Coverage Rate

To assess the overall operational completeness of the generated codebase, we define the following metric:

$$\text{Effort Coverage Rate (\%)} = \left(1 - \frac{\text{\# Manual Lines (Added + Changed)}}{\text{\# Total Lines in System}}\right) \times 100 \tag{3.4}$$

This metric reflects the percentage of the system that worked out-of-the-box without requiring manual fixes. A higher value indicates better operational quality. Results are summarised in Table 3.6.

3.3.3 LLM Configuration and Execution

All transformations and code generation steps were performed using GPT-4o[3]. Prompts were automatically generated per task and microservice, and outputs were validated across multiple iterations.

3.4 Results

This section presents the empirical results of applying our approach to the PetClinic and FXML-POS case studies, as described in Section 3.3.

[3] https://openai.com/index/gpt-4o-system-card/

3.4.1 Quantitative Results

3.4.1.1 Completeness

In this section, we present the completeness of *MicroPacker* output, demonstrating how the generated outputs align with the expected artifacts and classes.

3.4.1.1.1 Artifact-Level Completeness

Table 3.2 summarises artifact-level completeness over four iterations for both PetClinic and FXML-POS.

Table 3.2 Artifact Generation Completeness by Task and Iteration (% Coverage Rate)

			PetClinic			FXML-POS		
Task	**Sub-task**	**Artifact**	I1	I2	I3	I1	I2	I3
1. Encapsulate	1.1	Getter and Setter Methods	100%	100%	100%	100%	100%	100%
		Updated Method Calls Using Accessors	100%	100%	100%	85%	85%	100%
	1.2	Proxy Classes	100%	100%	100%	100%	100%	100%
		Interface Definitions	100%	100%	100%	100%	100%	100%
		Polymorphic Assignments via Interfaces	100%	100%	100%	100%	100%	100%
	1.3	Factory Classes	100%	100%	100%	70%	100%	100%
		Proxy Classes for Instantiation	75%	100%	100%	60%	100%	100%
2. API	2.1	API Endpoints	69%	92%	100%	80%	100%	100%
	2.2	Service Call Implementations	71%	88%	88%	72%	82%	96%
3. Libraries	3.1	List of Libraries Per Microservice	100%	100%	100%	100%	100%	100%
4. Runtime Patterns	4.1	External Configuration File (e.g., application.yml)	100%	100%	100%	100%	100%	100%
		Placeholder-Based Injection in Code	100%	100%	100%	100%	100%	100%
		Connection Setup (bootstrap.yml)	100%	100%	100%	100%	100%	100%
	4.2	Service Registration Configuration	100%	100%	100%	100%	100%	100%
		Discovery-Enabled Clients	100%	100%	100%	100%	100%	100%
	4.3	Load Balancing Configuration	100%	100%	100%	100%	100%	100%
		Client Integration with Dynamic Resolution	100%	100%	100%	100%	100%	100%

3.4.1.1.2 Class-Level Completeness

Table 3.3 summarises class-level completeness over four iterations for both PetClinic and FXML-POS. We evaluate completeness at the class level by categorising each class as full, partial, or failed.

Table 3.3 Class Completion per Task and System in the final iteration

Task	**Sub-task**	**PetClinic**			**FXML-POS**		
		Full	Partial	Failed	Full	Partial	Failed
1. Encapsulate	1.1 Attribute Refactoring	100%	0%	0%	100%	0%	0%
	1.2 Inheritance Refactoring	100%	0%	0%	100%	0%	0%
	1.3 Class Instantiation	100%	0%	0%	100%	0%	0%
2. API	2.1 Generate Endpoints	100%	0%	0%	100%	0%	0%
	2.2 Replace Calls	80%	20%	0%	82%	8%	8%
3. Libraries	3 Identify Libraries	100%	0%	0%	100%	0%	0%
4. Runtime Patterns	4.1 Config Server	100%	0%	0%	100%	0%	0%
	4.2 Service Registry	100%	0%	0%	100%	0%	0%
	4.3 Load Balancer	100%	0%	0%	100%	0%	0%

3.4.1.1.3 Findings

The evaluation of the completeness of *MicroPacker* output demonstrates that our approach achieves high completeness in generating the expected artifacts and applying the corresponding transformations. At the artifact level, the coverage rate consistently improved across iterations, reaching 100% in most cases by the third iteration. For both systems (PetClinic and FXML-POS), core artifacts such as getter/setter methods, proxy classes, API endpoints, configuration files, and service registry definitions were generated systematically. Minor gaps were observed in earlier iterations, particularly for Proxy Classes for Instantiation and Service Call Implementations,

where initial coverage was below 80%, but these were fully resolved by iteration three. This progression indicates MicroPacker's ability to converge toward full artifact-level completeness with generated feedback. At the class level, *MicroPacker* successfully applied transformations to all relevant classes in most subtasks. The final iteration shows 100% full completion for all subtasks except 2.2 Replace Calls, where a small proportion of classes (PetClinic: 20%; FXML-POS: 16%) were only partially transformed or left unchanged. Overall, the findings confirm that *MicroPacker* effectively generates the full set of expected artifacts and applies them across the targeted classes, thereby validating the completeness of the packaging phase in terms of artifact generation and transformation.

3.4.1.2 Correctness

This section reports the results of correctness evaluation, which measures how accurately the generated artifacts conform to their intended transformation patterns and how well the transformations were applied across relevant classes.

3.4.1.2.1 Artifact-Level Correctness

Table 3.4 presents the correctness for each artifact across migration tasks.

Table 3.4 Artifact Correctness by Task and Iteration (Precision / Recall / F1)

			PetClinic			FXML-POS		
Task	**Sub-task**	**Artifact**	Precision	Recall	F1	Precision	Recall	F1
1. Encapsulate	1.1	Getter and Setter Methods	1.00	1.00	1.00	1.00	1.00	1.00
		Updated Method Calls Using Accessors	1.00	1.00	1.00	1.00	0.86	0.92
	1.2	Proxy Classes	1.00	1.00	1.00	1.00	1.00	1.00
		Interface Definitions	1.00	1.00	1.00	1.00	1.00	1.00
		Polymorphic Assignments via Interfaces	1.00	1.00	1.00	0.50	1.00	0.67
	1.3	Factory Classes	0.75	1.00	0.86	0.80	1.00	0.89
		Proxy Classes for Instantiation	0.75	1.00	0.86	0.70	1.00	0.82
2. API	2.1	API Endpoints	0.75	1.00	0.86	0.87	1.00	0.93
	2.2	Service Call Implementations	0.92	0.87	0.89	0.89	0.96	0.93
3. Libraries	3	List of Libraries Used per Microservice	1.00	1.00	1.00	1.00	1.00	1.00
4. Runtime Patterns	4.1	External Configuration File	1.00	1.00	1.00	1.00	1.00	1.00
		Placeholder-Based Injection in Code	0.71	1.00	0.83	0.67	1.00	0.80
		Connection Setup (bootstrap.yml)	1.00	1.00	1.00	1.00	1.00	1.00
	4.2	Service Registration Configuration	1.00	1.00	1.00	1.00	1.00	1.00
		Discovery-Enabled Clients	0.86	1.00	0.92	0.67	1.00	0.80
	4.3	Load Balancing Configuration	1.00	1.00	1.00	1.00	1.00	1.00
		Client Integration	0.57	1.00	0.73	0.67	1.00	0.80

Overall, the artifact-level correctness results show that *MicroPacker* produces highly accurate code across most tasks and artifacts. All elements, such as getters/setters, proxy classes, interface definitions, and configuration files, achieve perfect precision and recall in both systems. Slight reductions are observed in artifacts requiring more context-sensitive generation. For instance, factory classes and proxy instantiations show F1-scores between 0.82 and 0.89 in FXML-POS, indicating minor mismatches or redundancy in their structure. Similarly, artifacts like placeholder injections and dynamic client integration yield lower precision (as low as 0.57 in PetClinic), reflecting variability in code formatting or integration patterns. Despite these outliers, the overall correctness remains consistently high across systems.

3.4.1.2.2 Class-Level Accuracy

The class-level correctness results are summarised in Table 3.5. Most classes were transformed correctly by the final iteration, with a small subset requiring minor fixes. Few classes remained incorrectly transformed, primarily due to artifacts that were syntactically invalid or semantically inconsistent.

Table 3.5 Class Correctness by System (Final Iteration)

Task	**Sub-task**	**PetClinic**			**FXML-POS**		
		Correct	Partial	Incorrect	Correct	Partial	Incorrect
1. Encapsulate	1.1 Attribute Refactoring	50%	50%	0%	62.5%	37.5%	0%
	1.2 Inheritance Refactoring	33.3%	66.7%	0%	50%	25%	25%
	1.3 Class Instantiation	66.7%	33.3%	0%	50%	41.7%	8.3%
2. API	2.1 Generate Endpoints	66.7%	33.3%	0%	57.1%	28.6%	14.3%
	2.2 Replace Calls	75%	8.3%	16.7%	58.3%	25%	16.7%
3. Libraries	3 Identify Libraries	0%	100%	0%	0%	100%	0%
4. Runtime Patterns	4.1 Config Server	0%	100%	0%	0%	100%	0%
	4.2 Service Registry	0%	100%	0%	0%	100%	0%
	4.3 Load Balancer	0%	100%	0%	0%	100%	0%

At the class level, correctness is similarly strong for core tasks such as attribute refactoring, endpoint generation, and class instantiation. In PetClinic, up to 75% of classes for Replace Calls sub-task were correctly transformed, and all remaining errors were either partial or minimal. FXML-POS showed comparable results but with slightly higher variance across sub-tasks (e.g., only 58.3% correct for service call replacements). Runtime-related tasks (e.g., config server, registry, load balancer) show 100% partial transformations, suggesting that while the required artifacts were introduced, they were not fully integrated at the class level.

3.4.1.2.3 Findings

The results of the correctness evaluation confirm that *MicroPacker* generates syntactically and structurally accurate code artifacts for most transformation tasks, across both PetClinic and FXML-POS systems.

Table 3.4 reveals consistently high scores across key transformation outputs. Core artifacts such as getter/setter methods, proxies, interface definitions, and configuration files achieved perfect scores (Precision, Recall, and F1 equal to 1.00) in both systems, indicating strong adherence to expected structural patterns. Minor deviations appear in more complex transformations. For instance, polymorphic assignments in FXML-POS achieved perfect recall but only 0.50 precision, leading to a reduced F1-score (0.67). Similarly, factory-related artifacts showed slightly lower precision (0.70–0.80), suggesting occasional mismatches in factory signature or placement. API-related transformations were largely correct (larger than 0.86), with service call rewrites in particular reaching 0.89 (PetClinic) and 0.93 (FXML-POS), indicating that endpoint mapping and invocation logic are accurately reconstructed. Runtime configurations (e.g., service discovery and load balancing) were highly accurate overall, though placeholder injections and dynamic client integration had lower precision (0.57–0.71), reflecting occasional inconsistencies in formatting or token placement.

Table 3.5 provides further insights into the distribution of transformation accuracy at the class granularity. Tasks like attribute refactoring and class instantiation exhibit high correctness across both systems (e.g., 50–66.7% correct in PetClinic, up to 62.5% in FXML-POS), with the remaining cases being mostly partially correct rather than incorrect. For API generation, correctness was strong but more varied; endpoint creation showed high accuracy (66.7% in PetClinic, 57.1% in FXML-POS), while service call rewriting had higher variance (e.g., 75% correct in PetClinic and 58.3% in FXML-POS), suggesting greater sensitivity to context. The library resolution and runtime configuration tasks consistently showed 100% partial transformations with no failed classes, indicating that *MicroPacker* is capable of initiating

required artifacts globally, but final integration at the class level remains superficial or incomplete in these tasks.

3.4.1.3 Operationality

This section presents the results of evaluating the operational readiness of the generated microservices. Our objective is to quantify the amount of manual engineering effort required to render the generated system fully functional, and to assess the overall usability of the output.

3.4.1.3.1 Manual Effort Metrics

We assess operational effort by measuring three key indicators: lines of code added, lines modified, and the number of source files affected. Table 3.6 summarises these metrics for both PetClinic and FXML-POS systems.

Table 3.6 Manual Code Modifications by System

Metric	**Type**	**PetClinic (Total)**	**FXML-POS (Total)**
Lines Modified	Added	172	247
	Changed	251	383
Files Affected	–	35	43

As shown in Table 3.6, the manual engineering effort remained moderate across both systems. PetClinic required 172 lines of additional code and 251 modified lines, while FXML-POS involved slightly more adjustments with 247 lines added and 383 lines changed. These edits affected 35 and 43 files, respectively.

3.4.1.3.2 Effort Coverage Rate

Table 3.7 presents the calculated coverage for both systems. Higher values indicate greater operational quality and lower reliance on manual post-processing.

Table 3.7 Effort Coverage Rate by System

System	Total Lines	Effort Coverage Rate
PetClinic	423	0.86
FXML-POS	630	0.82

3.4.1.3.3 Findings

The operational evaluation confirms that *MicroPacker* produces microservices that are largely functional with minimal manual effort. The manual code modifications were relatively light, requiring fewer than 400 lines of intervention in total. Most changes involved minor adjustments such as fixing import statements, completing dependency injections, or aligning method signatures. These results are reinforced by the high effort coverage rates (0.86 for PetClinic and 0.82 for FXML-POS), which quantify the proportion of the codebase that was immediately usable without edits. This strong coverage demonstrates that the generated output is not only syntactically and structurally correct but also operationally usable with limited post-generation engineering. Collectively, these findings validate MicroPacker's practical readiness for real-world integration

3.4.2 Qualitative Evaluation

In this subsection, we report on a qualitative evaluation conducted by the authors after applying *MicroPacker* to PetClinic and FXML-POS. This evaluation involved manually inspecting the

generated microservices to identify concrete examples of successes, limitations, and emerging patterns observed during the packaging phase.

One of the most successful aspects of *MicroPacker* was its ability to generate boilerplate code for recurring design patterns with high accuracy. For example, the generation of getter and setter methods (Task 1.1) and interface definitions (Task 1.2) consistently produced syntactically correct code, even for classes with multiple attributes. In PetClinic, for example, introducing accessors for the `Visit` class required no manual intervention, and the generated methods were correctly formatted and integrated with the rest of the system.

Similarly, the *MicroPacker* were highly effective in generating API wrappers (Task 2.1) for externally-invoked methods. In FXML-POS, the `PaymentService` class had several public methods that were directly called by other modules. *MicroPacker* successfully exposed these methods through RESTful endpoints, following naming conventions and HTTP verb guidelines. The prompts guided the model to preserve internal logic while wrapping methods inside annotated controller classes, which worked well across services.

Another successful outcome was observed in Task 3.3: Separate Libraries and Build Files, where *MicroPacker* generated self-contained `pom.xml` files for each microservice. For instance, in FXML-POS, the `InventoryService` microservice only required dependencies for Spring Web, Jackson, and SLF4J, and the generated build file correctly included only these libraries, omitting unused modules from the original monolith.

MicroPacker also performed well in generating runtime integration files, such as `bootstrap.yml` and external configuration files (Task 4.1). In both PetClinic and FXML-POS, *MicroPacker* correctly extracted configuration values like database URLs and port numbers from the source code and externalised them into central YAML files. The generated `bootstrap.yml` files linked correctly to the configuration server URI and loaded values dynamically at runtime. This integration was validated successfully during deployment, requiring minimal editing.

Moreover, *MicroPacker* could handle service registry configuration using Spring Cloud Eureka (Task 4.2) seamlessly. The LLM-generated annotations (e.g., `@EnableDiscoveryClient`) and dependency configurations and placed them were correctly.

Despite these benefits, certain tasks exposed the limitations of LLM-driven automation, particularly when transformations required maintaining fine-grained context across services. A notable example was observed in Task 2.2: Replace Direct Calls with Service Calls. While our approach correctly identified the target method in the `OrderService`, it occasionally generated incomplete or incorrect HTTP client code in the calling class. In one example, a generated `RestTemplate` call omitted query parameters required by the `getOrderById` endpoint, causing a runtime error that had to be corrected manually.

Similarly, in Task 1.3: Class Instantiation Refactoring, while our approach generated factory and proxy classes correctly, their integration into the existing source code was inconsistent. For example, in FXML-POS, the generated factory for the `Invoice` class correctly implemented the instantiation logic, but the `FulfillmentService` class continued using `new Invoice()` in one place, indicating that the transformation was incomplete and required a second iteration with a refined prompt.

MicroPacker sometimes made incorrect assumptions when code structure or naming deviated from common patterns. For example, in Task 4.3.2, when refactoring service calls to use client-side load balancing, *MicroPacker* generated a `FeignClient` interface but failed to recognise that the existing method signatures used complex DTOs with nested serialisation, leading to partially correct method mappings that failed during runtime.

Another common issue was over-generation, where *MicroPacker* included unnecessary or hallucinated code segments. For example, in Task 1.2, when applying the proxy pattern for inheritance, *MicroPacker* added additional utility methods to the proxy class that were not part of the original contract. In one case, the `OrderProxy` class included a `toJSON()` method, even though no other class invoked or needed such functionality. While this addition did not break the system, it introduced non-essential code that required manual cleanup.

Across all tasks, we observed that LLM *MicroPacker* output quality was highly sensitive to prompt structure and context. Prompts that included clear constraints, naming conventions, and step-wise instructions consistently yielded better results. For example, when we *MicroPacker* instructed the LLM to "retain the original field access temporarily" during the introduction of getters and setters, it correctly preserved backward compatibility. However, omitting such details often led to premature removal of field access, breaking dependent classes.

The iterative prompt refinement loop proved critical. In early iterations, placeholder-based configuration (Task 4.1.1) resulted in mixed formatting (e.g., `${{config.key}}` vs. `${config.key}`), leading to inconsistencies across services. After adjusting the prompt to enforce `${config.key}` syntax explicitly, consistency improved in subsequent outputs.

Overall, *MicroPacker* demonstrated good qualitative performance in generating structurally consistent, pattern-conforming code with operational usability. Its effectiveness was most notable in modular, pattern-based tasks (e.g., proxy generation, API exposure, configuration setup), while context-heavy transformations (e.g., dynamic service call adaptation) required manual refinement. These insights show the potential of LLMs in automating microservices packaging, while also highlighting the need for rigorous prompt design and validation loops to handle edge cases.

3.5 Discussion

In this section, we discuss our approach results and threats to their validity.

3.5.1 Results Discussion

This subsection discusses the implications of the results presented in Section 3.4, and reflects on the strengths, limitations, and broader impact of our approach.

The evaluation results demonstrate that *MicroPacker* effectively addresses the challenges of microservice packaging through prompt-based code transformation. In terms of *completeness*,

MicroPacker consistently generated all required artifacts and applied transformations to the appropriate classes, achieving full coverage across nearly all tasks by the final iteration. The progressive improvement across iterations underscores the strength of iterative prompt refinement. For *correctness*, the generated code was highly accurate in both syntactic structure and integration logic, with F1-scores reaching 1.00 for most core artifacts. Partial inaccuracies were primarily limited to context-sensitive tasks such as client integration and dynamic resolution, highlighting areas for targeted prompt enhancement. Finally, the *operationality* results confirm that *MicroPacker* outputs are practically usable with minimal manual effort. Fewer than 400 lines of intervention were needed per system, and over 80% of the codebase was directly operational, indicating strong alignment between generated structure and functional readiness.

MicroPacker was evaluated on two systems: PetClinic, a well-known benchmark system, and FXML-POS, a production-grade desktop-to-web migration use case. The tool's consistent performance across both systems can suggest good generalisability and adaptability to varied codebases. While the artifacts and transformations are consistent in purpose, the class structures, dependency depths, and service interactions differ, which supports the claim that MicroPacker's approach generalises beyond toy examples.

As stated in the introduction, our goal aligns with Kent Beck's principle: *"make it work, make it right, make it fast."* In this initial stage, our focus has been on "making it work" and "making it right", ensuring that the generated microservices are functionally complete, structurally correct, and operationally usable with minimal manual effort. While performance metrics such as latency, throughput, and resource usage are critical in production settings, they are intentionally not addressed in this work. Our results demonstrate that *MicroPacker* achieves a high level of automation in producing working, correctly packaged microservices, thereby laying the groundwork for future efforts toward performance tuning and deployment optimisation.

Despite the promising results, several limitations warrant attention. First, while correctness and completeness are high, certain subtasks—particularly service call replacements and placeholder injections—remain sensitive to code context. Addressing this may require enhanced prompt

conditioning, deeper code summarisation, or hybrid strategies that incorporate lightweight static analysis. Second, the current evaluation focuses on Java-based systems; broader applicability to polyglot environments remains to be studied. Third, while effort coverage provides a useful proxy for operationality, long-term maintainability and runtime performance were not directly assessed. Future work could integrate dynamic validation, such as test suite execution or runtime monitoring, to further validate operational robustness. Additionally, integrating *MicroPacker* into continuous integration (CI) workflows and testing it in larger-scale industrial migrations would offer valuable insights into its scalability and adoption readiness.

3.5.2 Threats to Validity

3.5.2.1 Internal Validity

One internal threat stems from the stochastic nature of LLMs (GPT-4o in our case), which can yield slightly different outputs for the same prompt due to context sensitivity and sampling variance. To mitigate this, we designed a structured prompt engineering pipeline and decomposed tasks into defined subtasks with fixed context windows. We also reused prompt templates consistently and validated outputs across iterations. Additionally, GPT-4o's response length limitations may hinder the generation of large code artifacts. To address this, we divided the transformation process into smaller, scoped subtasks, ensuring that no individual prompt exceeds token constraints.

3.5.2.2 External Validity

A key external threat lies in the non-reproducibility of results over time. As LLMs are frequently updated or fine-tuned, their outputs may evolve, potentially affecting the stability of our results. To address this, we provide complete replication material, including all prompt templates, inputs, and outputs. Another external threat concerns the lack of diversity in evaluation data; although we selected two structurally distinct systems (PetClinic and FXML-POS), more diverse case studies would strengthen the generalisability of the findings.

3.5.2.3 Construct Validity

Construct validity may be affected by how we measure correctness, completeness, and operationality. These dimensions are operationalised using syntactic metrics (e.g., presence of expected artifacts, precision/recall, manual edit counts), which do not fully capture behavioral correctness or runtime properties. While our metrics are practical for static analysis and early-stage validation, they do not substitute for dynamic testing, user satisfaction, or long-term maintainability, which should be addressed in future work. Although the approach aims to minimise human effort, manual actions were still involved in validating outputs, interpreting incomplete responses, and adjusting prompt templates. These interventions may introduce bias or variability. To reduce this threat, we have included automatically generated feedback and validation scripts.

3.5.2.4 Generalisability

Our evaluation is limited to two Java-based systems, which raises concerns about the generalisability of the approach to other domains, architectures, or programming languages. We mitigate this by providing a detailed methodology for prompt development that can be adapted to other contexts. Furthermore, we currently rely on a single LLM (GPT-4o), and results may vary across models. Future work will include cross-model comparisons (e.g., LLaMA, Claude, Gemini) and larger-scale evaluations across diverse domains to better assess robustness.

3.6 Conclusion

This chapter tackled the packaging phase of monolith-to-microservices migration, an essential yet often overlooked phase in existing literature. While prior research has predominantly emphasised the identification of microservice candidates, we focused on automating the transformation of those microservice candidates into independently deployable services through source code encapsulation, API generation, library separation, and the integration of essential runtime patterns.

We introduced *MicroPacker*, a tool-augmented approach powered by Large Language Models, which leverages systematic prompting to perform context-aware refactoring and artifact generation. Our evaluation on two real-world systems demonstrated that *MicroPacker* can achieve high levels of artifact completeness, correctness, and operational readiness with limited manual intervention, validating its effectiveness in supporting this transformation-heavy phase.

Our empirical evaluation on two Java-based monolithic systems, PetClinic and FXML-POS, demonstrates that *MicroPacker* delivers strong results across all three evaluation dimensions. In terms of completeness, artifact-level coverage reached 100% for most packaging tasks by the third iteration, with nearly all classes fully transformed across subtasks. For correctness, the majority of artifacts achieved high precision, recall, and F1-scores (above 0.90) with core transformations such as getter/setter insertion, API wrapping, and runtime configuration reaching perfect scores. Regarding operationality, over 80% of the generated codebase was functional without any manual intervention, while the remaining modifications required fewer than 400 lines of code changes per system.

These findings confirm that *MicroPacker* effectively automates the packaging phase while producing microservices that are complete, correct, and readily deployable.

The packaged microservices obtained in this work form a foundation for the next phase of the migration process, which is deployment. In the following chapter (Chapter 4), we build upon the packaged microservices to automatically generate deployment artifacts, such as Dockerfiles and Kubernetes manifests.

CHAPTER 4

MICROSERVICES DEPLOYMENT PHASE

4.1 Introduction

In the previous chapter, we introduced *MicroPacker*, a large language model–based approach for transforming monolithic code into independently packaged microservices. *MicroPacker* produces structurally complete and logically cohesive services that can now be deployed and used in production. This chapter addresses the next critical phase in the migration process: automating the generation of deployment artifacts, specifically Dockerfiles and Kubernetes (k8s) manifests to support scalable and maintainable deployment of microservices.

Deploying microservices remains a complex challenge, requiring developers to manually write and maintain deployment artifacts such as Dockerfiles and Kubernetes manifests (Vayghan, Saied, Toeroe & Khendek, 2018). These artifacts define how microservices are built, packaged, and orchestrated, making them critical to the success of any deployment (Vohra, 2016). Despite their importance, manually creating and maintaining these configurations is time-consuming, error-prone, and requires expertise, often leading to deployment failures and increased operational overhead (Mahajan & Mane, 2022).

Containerisation has emerged as a key enabler for microservices, providing portability, scalability, and consistency between development and production environments (Balalaie, Heydarnoori, Jamshidi, Tamburri & Lynn, 2018). However, existing deployment approaches, as shown in our SLR (presented in Chapter 1), have primarily focused on resource allocation optimisation and failure detection, neglecting the need for automating containerisation (Saucedo *et al.*, 2025). As a result, developers must still manually create Dockerfiles and Kubernetes configurations, a process that remains manual and error-prone (Hu, Peng, Wang & Gao, 2025).

While alternative orchestration and configuration tools such as Docker Swarm and Nomad exist, Docker and Kubernetes have emerged as the de facto standards in industrial deployment pipelines. Surveys report that up to 96% of organisations are using or evaluating Kubernetes,

with more than 60% running it in production (Cloud Native Computing Foundation, 2021). The widespread adoption of Kubernetes for microservices orchestration further amplifies these complexities. These include the need to understand numerous YAML configuration fields, maintain consistency across services, ensure compliance with best practices, and correctly handle service dependencies and scaling policies. The study by Zhu & Gehrmann (2022) highlights that Kubernetes use in production has risen to 83%, up from 78% in 2019, with companies such as Spotify, IBM, Nokia, and Yahoo relying on it for their microservices deployments. However, lessons learned from large-scale Kubernetes deployments show that misconfigurations and deployment failures remain significant challenges (Vayghan *et al.*, 2018). Furthermore, the shortage of experienced Kubernetes practitioners (Zhang *et al.*, 2024) has made it increasingly difficult for organisations to manage deployments effectively, highlighting the need for automated solutions that reduce reliance on expert knowledge.

Some tools have been proposed to simplify containerization and reduce manual effort, including template-based tools and configuration generators (Developers, 2024; Project, 2025; CLI, 2025). However, these methods suffer from significant limitations. Template-based tools like Helm (Project, 2025) rely on predefined configurations that fail to adapt to the unique requirements of different microservices, making them inflexible and difficult to maintain. Many configuration generators, such as Helm and Kustomize, still require substantial manual input to fine-tune generated configurations, limiting their practicality in large-scale deployments (Project, 2025; CLI, 2025). Additionally, existing tools lack adaptive learning mechanisms, preventing them from adjusting to evolving deployment needs (Rosa, Mastropaolo, Scalabrino, Bavota & Oliveto, 2023; Hu *et al.*, 2025). These limitations create a pressing need for automated, context-aware solutions that can generate high-quality deployment artifacts with minimal human intervention.

To overcome these limitations, we propose *MiDKo*, an approach for automating Dockerfile and Kubernetes manifest generation. *MiDKo* combines retrieval-augmented generation (RAG) (Lewis *et al.*, 2020) with topic modeling to create adaptive deployment support. While traditional tools rely on static templates, our SLR revealed the need for solutions that can incorporate past deployment knowledge in a reusable and flexible manner.

To evaluate our approach *MiDKo*, we conducted a comprehensive evaluation comprising (1) syntactic correctness (using linters such as Hadolint and kube-score), (2) instruction-level similarity to ground truth (precision and recall), and (3) a qualitative survey with 11 practitioners, including developers and DevOps engineers. Our evaluation was based on 317 microservice projects, encompassing 1,607 Dockerfiles and 155 manually written Kubernetes manifests. The results show that MiDKo-generated artifacts are more syntactically correct (91% clean Dockerfiles, 70% clean K8s manifests), exhibit higher instruction-level accuracy, and are perceived as more readable and useful than open-source artifacts. These findings demonstrate that *MiDKo* can effectively support deployment automation within microservices migration pipelines.

The remainder of this chapter is structured as follows. Section 4.2 details our *MiDKo* approach. Section 4.3 describes the experimental setup. Section 4.4 presents the quantitative and qualitative results of *MiDKo* approach. Section 4.5 discusses our findings and addresses threats to validity. Finally, Section 4.6 concludes the chapter.

4.2 Deployment approach: MiDKo

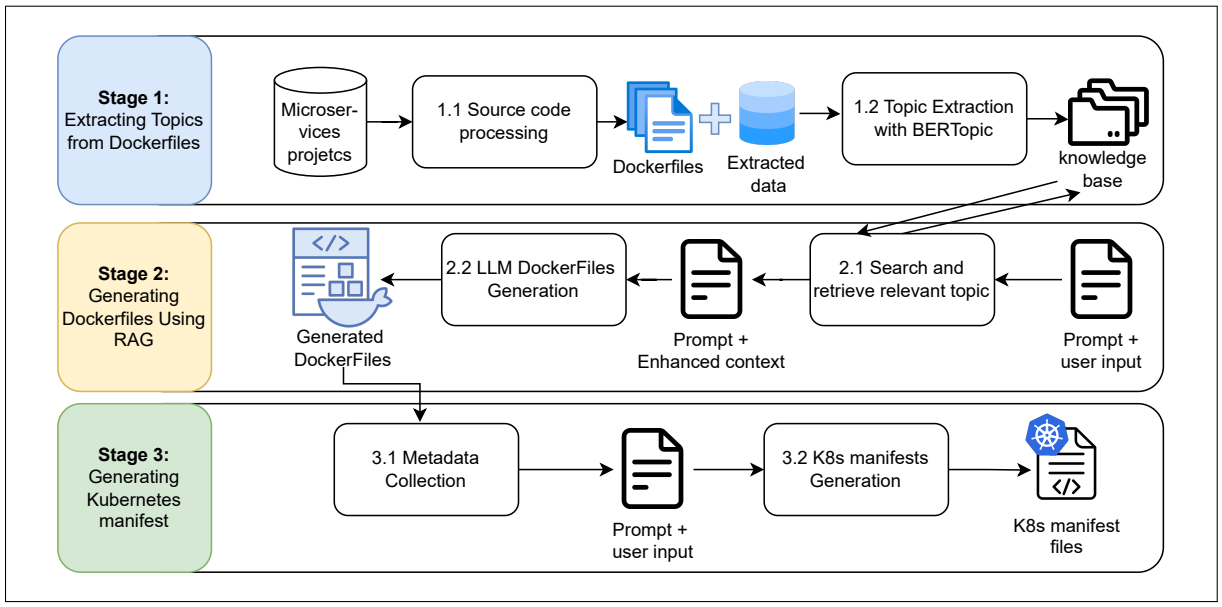

Figure 4.1 MiDKo Approach overview

We propose MiDKO, an approach that automates the generation of Dockerfiles and Kubernetes manifests for microservices by leveraging Retrieval-Augmented Generation (RAG) (Lewis *et al.*, 2020) and large language models. *MiDKo* follows a three-stage approach: (1) extracting topics from existing Dockerfiles, (2) generating Dockerfiles using LLMs and RAG, and (3) generating Kubernetes manifests while ensuring consistency with the generated Dockerfiles. Figure 4.1 provides an overview of the proposed approach. The following sections describe each stage in detail.

4.2.1 Stage 1: Extracting Topics from Dockerfiles

MiDKo begins by analysing a dataset of 378 microservice projects (Amoroso d'Aragona *et al.*, 2024) to extract relevant patterns from existing Dockerfiles. The extracted knowledge is then used to build a knowledge base that enhances the RAG-driven generation process.

4.2.1.1 Step 1.1 Source Code Processing

We first extract Dockerfiles from the selected projects to construct this knowledge base. After filtering out projects that only contain `docker-compose.yml` files, we obtain **1,607 Dockerfiles** across 317 repositories.

Additionally, we analyse the source code and configuration files to extract essential metadata about each microservice. Specifically, we collect information on:

- Programming language: Identified by analysing the source code structure and the presence of language-specific files.
- Framework: Determined by inspecting dependency files for known framework-related keywords.
- Dependencies: Extracted by fetching and analysing the content of dependency management files.
- Database technology: Inferred from configuration files and database-related dependencies.
- Build system: Identified based on the presence of build-related files and project structure.

- Configuration files: Detected by scanning for commonly used configuration files across different technologies.
- Static files: Identified by analysing the project structure and locating static assets such as images and scripts.

This metadata provides structured insights that guide the LLMs for the generation of Dockerfiles and Kubernetes manifests files.

4.2.1.2 Step 1.2 Topic Extraction with BERTopic

We begin by preprocessing the Dockerfiles by removing comments and tokenising them into 2- to 5-grams. Next, we apply BERTopic (Grootendorst, 2022) to extract distinct topics from the Dockerfiles. Each topic represents a recurring pattern or configuration commonly found in Dockerfiles. For example, one topic captures Dockerfiles for Python-based services using Flask, while another focuses on Node.js services with MongoDB integration. Finally, we register the extracted topics in the knowledge base, storing their key terms along with three representative Dockerfiles. These examples facilitate n-shot learning, enabling the LLM to generate more contextually accurate Dockerfiles.

4.2.2 Stage 2: Generating Dockerfiles Using RAG

In this stage, we generate Dockerfiles using Retrieval-Augmented Generation, which combines the flexibility of large language models with structured knowledge extracted from existing Dockerfiles. The RAG framework dynamically retrieves relevant topics from a knowledge base and incorporates them into the LLM-driven generation process. This approach ensures that the generated Dockerfiles adhere to best practices while being tailored to the specific requirements of each microservice.

4.2.2.1 Step 2.1 Searching and Retrieving Relevant Topics

The retrieval is a crucial component of our approach, as it enables the selection of the most relevant topic from the knowledge base to guide Dockerfile generation. The knowledge base consists of 54 topics, each representing common patterns in Dockerfile configuration. The retrieval process follows these key steps:

1. **Query Creation:** A structured query is generated using the extracted metadata from the microservice.
   Query Template: "Create a Dockerfile for a language application using framework and build system"
2. **Similarity Search:** The query is used to retrieve the most relevant topics from the knowledge base. We employ sparse retrieval with BM25 (Robertson, Walker, Beaulieu, Gatford & Payne, 1996) to rank and identify the best-matching topic.
3. **Topic Selection and Metadata Retrieval:** Once the most relevant topic is identified, it is returned along with representative Dockerfile examples. These examples are then used to enhance the prompt through n-shot learning and RAG-based augmentation.

4.2.2.2 Step 2.2 Dockerfile Generation

In this step, the LLM generates Dockerfiles using a structured prompt that integrates the retrieved topic, for example, Dockerfiles, microservice metadata, and user input. To ensure consistency and adaptability, we propose a standardised *prompt template* that provides a structured yet customizable approach to Dockerfile generation.

The proposed prompt template consists of the following key components:

- **Microservice Description:** A brief overview of the microservice, including its primary language, framework, and dependencies.
- **Functional Requirements:** Key specifications such as environment variables, required packages, and exposed ports. Those are provided by the user.

- **Retrieved Topic:** The topic retrieved from the knowledge base, serving as a foundation for the Dockerfile.
- **Example Dockerfiles:** Representative Dockerfiles relevant to the retrieved topic, used as n-shot examples.
- **Customisation Instructions:** Explicit guidance for modifying the retrieved template to align with the specific requirements of the microservice.

Listing 4.1: Proposed Prompt Template for Dockerfile Generation

```
"
You are an expert in DevOps and containerization. Your task is to generate an
    optimized and secure Dockerfile for a microservice based application the
    following extracted information from the source code and user-provided
    inputs.

## Information Extracted from the Source Code:
- Programming Language: {language}
- Framework: {framework}
- Dependencies: {dependencies}
- Database: {database}
- Build System: {build_system}
- Configuration Files: {config_files}
- Static Files: {static_files}

## Information Provided by the User:
- Base Image: {base_image}
- Security Considerations: {security}
- Deployment Environment: {deployment_environment}
- Networking: {networking}
- Health Checks: {health_checks}
- Exposed Ports: {exposed_ports}
- Build Arguments: {build_args}
```

```
- Copy Instructions: {copy_instructions}
- Execution Command: {execution}
- Run Commands: {run_commands}
- Users: {user}
- Environment variable: {envs}
- labels: {labels}

## Look to those retrieved topics and examples:
{retrieved_topic}

## Requirements:
1. Generate an optimised Dockerfile.
2. Set the correct environment variables and dependencies.
3. Ensure the correct entry point and CMD/ENTRYPOINT instructions.
4. Include the correct exposed ports and necessary copy instructions.

Return only the Dockerfile without any additional explanations or comments."
```

This structured prompt ensures a consistent and efficient approach for guiding the LLM in Dockerfile generation. The retrieved topic and example Dockerfiles provide a strong starting point, while the customisation instructions ensure that the final output aligns with the microservice's specific requirements and best practices.

4.2.3 Stage 3: Generating Kubernetes Manifests

In this stage, we generate Kubernetes manifests for each microservice using LLMs, enriched with metadata extracted from multiple sources, including the Dockerfile, source code, and user input. To ensure adherence to Kubernetes best practices, we use predefined templates derived from established references (Vohra, 2016). The LLM customises and extends these templates

based on the extracted metadata, ensuring that the generated manifests accurately reflect the deployment requirements of each microservice.

The generation process consists of two main steps:

4.2.3.1 Step 3.1 Metadata Collection

To generate accurate Kubernetes manifests, metadata is collected from the following sources:

- **Dockerfile**: Extracts the container image name, exposed ports, and environment variables.
- **Source Code**: Analyses dependencies, environment variables, and configuration files to infer additional service settings.
- **User Input**: Captures service-specific requirements such as resource limits, replica counts, storage configurations, and secrets.

4.2.3.2 Step 3.2 Kubernetes Manifest Generation

Using the extracted metadata, the LLM generates Kubernetes manifest files, including `Deployment.yaml` and `Service.yaml`. The predefined templates provide a structural foundation, while the LLM dynamically integrates the relevant metadata to generate customised and optimised manifests for each microservice.

4.3 Study design

In this section, we describe the dataset used to evaluate the generation of Dockerfiles and Kubernetes configuration files. We also outline our experimental procedure and the evaluation methodology. All code generation was performed using the OpenAI GPT-4 model via the OpenAI API.

4.3.1 Dataset

To evaluate our approach, we used a dataset of 378 open-source projects (Amoroso d'Aragona *et al.*, 2024). After processing the dataset, we retained only 317 projects that contained at least one Dockerfile, resulting in a total of 1,607 Dockerfiles written in various programming languages: JavaScript (427), Java (414), Python (257), Go (130), Ruby (35), PHP (34), and others. Additionally, our dataset analysis identified prevalent build systems, with npm (407), Maven (389), and pip (173) being the most commonly used. Furthermore, we examined database integration, revealing 514 Dockerfile configurations databases, with PostgreSQL (243) and MongoDB (115) emerging as the most frequently used.

Regarding Kubernetes configurations, our analysis revealed that 68 projects (21%) utilised Kubernetes, containing a total of 1,575 manifest files. After filtering, we identified 155 manually written Kubernetes manifests from 16 projects, specifically `Service.yaml` (104 files) and `Deployment.yaml` (51 files). This filtering was crucial to ensure a fair comparison by eliminating auto-generated files created with Helm, Kustomize, or Jsonnet, thereby preventing tool-specific artifacts from biasing our results.

The full dataset is available in the replication package [1].

4.3.2 Experimental Setup

To evaluate Dockerfile generation, we compared three approaches:

- **Dockerfiles Original:** Manually written Dockerfiles collected from the dataset.
- **Dockerfiles without Topics:** Dockerfiles generated by *MiDKo* without guidance from topic modeling.
- **Dockerfiles MiDKo:** Dockerfiles generated by *MiDKo* augmented with topic modeling and retrieval.

[1] https://anonymous.4open.science/r/MiDKo-E1AA

Our evaluation is based on a curated dataset of 378 open-source projects. After filtering, we retained 317 repositories containing at least one Dockerfile, resulting in a total of **1,607 Dockerfiles** written in various programming languages and deployment styles.

To emulate realistic developer input, we extracted key metadata—such as exposed ports, base images, and environment variables—from existing Dockerfiles and Kubernetes manifests. This metadata was used to populate the prompt inputs for both Dockerfile and Kubernetes configuration generation.

We adopted a 10-fold cross-validation strategy for Dockerfile evaluation to ensure robust and unbiased results. All 1,607 Dockerfiles were randomly partitioned into ten equally sized folds. In each iteration, 90% of the Dockerfiles were used to construct the topic model and retrieval knowledge base (training set), while the remaining 10% served as the test set. For each fold, a new BERTopic model was trained on the training Dockerfiles, yielding between 37 and 45 topics depending on the diversity and structure of the training data. This process was repeated 10 times, ensuring that each Dockerfile was used for testing exactly once. Dockerfile generation and instruction-level similarity comparisons were performed using only the training data within each fold. The resulting precision, recall, and F1-scores were averaged across all folds and are reported in Table 4.2.

In contrast, Kubernetes manifest evaluation was conducted on the complete set of available configuration files without cross-validation. This decision reflects the smaller number of examples and the absence of a retrieval-based generation component. We compared our generated manifests to 155 manually written Kubernetes configurations extracted after a cleaning process. This process decomposed compound YAML files (e.g., those containing both `Service.yaml` and `Deployment.yaml`) and split multi-service definitions, resulting in 809 individual manifest files suitable for evaluation.

4.3.3 Evaluation Metrics

To assess the effectiveness of our approach, we employed a combination of quantitative and qualitative metrics. Syntactic correctness and similarity metrics were computed using linters and instruction-level comparisons, while structural correctness, readability, and usefulness were evaluated through a user survey involving domain experts.

4.3.3.1 Syntactic Correctness

We evaluated the syntactic validity of the generated files using established linters. For Dockerfiles, we used *Hadolint* (hadolint, 2025), which detects syntax errors and deviations from best practices. For Kubernetes manifests, we used *kube-score* (kube score, 2025), which validates structural correctness based on Kubernetes schema definitions. A file is considered syntactically correct if it passes the linter without critical errors.

4.3.3.2 Precision and Recall-Based Similarity Analysis

We computed instruction-level precision and recall for Dockerfiles and component-level precision and recall for Kubernetes manifests. We define:

$$\text{Precision} = \frac{TP}{TP + FP} \quad , \quad \text{Recall} = \frac{TP}{TP + FN} \tag{4.1}$$

Where:

- TP (True Positives): Correctly generated elements that match the ground truth.
- FP (False Positives): Elements generated but not found in the ground truth.
- FN (False Negatives): Elements in the ground truth that were not generated.

Each configuration instruction (e.g., `FROM`, `RUN`) or manifest component (e.g., container image, selector) was treated independently. A match was considered valid only when both the instruction name and its associated argument matched exactly between the generated and manual files. For

example, `EXPOSE 8080` in the generated file counts as a match only if the manual configuration also contains `EXPOSE 8080`.

4.3.3.3 Structural Correctness

To complement the syntactic analysis, we evaluated the structural correctness of the generated configurations through expert reviews. Participants assessed whether the structure of each Dockerfile and Kubernetes manifest aligned with expected deployment practices. This included checking base image selection, instruction order, correct port exposure, environment configuration, and service-deployment consistency.

4.3.3.4 Readability

Readability was assessed to determine how clearly the generated configurations convey their intent. Participants evaluated whether the files were well-organized, logically structured, and easy to follow. This metric is critical for practical adoption, as maintainable configuration files reduce the risk of deployment errors and facilitate onboarding.

4.3.3.5 Usefulness

Usefulness reflects the practical value of the generated artifacts for real-world use cases. Participants rated the degree to which each configuration could be directly applied, modified, or extended in a typical software deployment scenario. This subjective metric captures the alignment of generated content with developers' expectations and operational needs.

4.4 Results

Table 4.1 presents the results of our experiment, comparing the Dockerfiles generated by MiDKO—with and without topic modeling—to the open-source Dockerfiles we collected. Additionally, it compares the generated Kubernetes manifest files to those extracted from open-source projects.

Table 4.1 Comparison of results in terms of syntax and survey results. An upward arrow (↑) indicates that higher values are better, while a downward arrow (↓) indicates that lower values are preferable

Approach	**Syntax Correctness (#files)**		**Survey (scale 1 (Very Poor) → 5 (Excellent))**		
	Clean ↑	Corrupted ↓	Correctness ↑	Readability ↑	Usefulness ↑
Docker_original	1,203 (74%)	404	4.0 (good)	2.3 (poor)	3.0 (average)
Docker_without_topics	1,448 (90%)	159	3.7 (good)	4.5 (excellent)	3.3 (average)
Docker_MiDKo	1,459 (91%)	148	3.9 (good)	4.7 (excellent)	4.0 (good)
K8s_original	72 (46%)	83	3.7 (good)	2.2 (poor)	3.0 (average)
K8s_MiDKo	109 (70%)	46	4.0 (good)	4.2 (good)	4.2 (good)

4.4.1 Quantitative Evaluation

Our quantitative evaluation focuses on assessing the quality of generated deployment artifacts through two primary criteria: syntactic correctness (based on linter outputs) and similarity to ground-truth files (measured via instruction-level precision, recall, and F1-score).

4.4.1.1 Syntactic Correctness

For the original Dockerfiles, we obtained 1203 syntactically correct files, representing 74% of the dataset. The total number of errors detected was 661, with 2549 warnings. The most common errors included `DL3020` (incorrect use of `COPY` commands leading to inefficiencies) with 468 occurrences and `DL4000` (improper use of `HEALTHCHECK` instructions) with 105 occurrences. For Dockerfiles generated without topics, 1448 files were free of syntax errors (90% of total generated files). The total number of errors decreased to 173, with 2132 warnings. The most frequent issues were `DL1000` (incorrect syntax in `Dockerfile` instructions), appearing 64 times and `SC1072` (unexpected token errors), occurring 35 times. For Dockerfiles generated with MiDKO, 1459 files were free of syntax errors (91%), showing further improvement. The total

errors were reduced to 153, with 2047 warnings. The most frequent errors included `DL1000` with 72 occurrences and `SC1072` with 22 occurrences. These results indicate that Dockerfiles generated with *MiDKo* demonstrate improved syntactic correctness, reducing both errors and warnings compared to the original and topic-model-free approaches.

For the original Kubernetes manifest files, we obtained 72 files without errors. The total number of errors detected was 406, indicating frequent misconfigurations, with the most common critical issues being Container Security Context User Group ID (51 occurrences), Container Ephemeral Storage Request and Limit (51 occurrences), and Pod NetworkPolicy (51 occurrences). For Kubernetes manifests generated with MiDKo, the quality of configurations improved significantly, with 109 files free of errors, representing a higher syntactic correctness rate. The total number of errors decreased to 265, showcasing a substantial reduction. The most frequent errors included Pod NetworkPolicy (48 occurrences), Container Security Context User Group ID (48 occurrences), and Container Image Pull Policy (26 occurrences). These results indicate that MiDKo-generated Kubernetes manifests achieve better syntactic correctness and fewer errors compared to the original files.

4.4.1.2 Precision and Recall-Based Similarity Analysis

Table 4.2 presents the detailed precision and recall values for each instruction in the Dockerfiles. We compare the precision and recall of different Dockerfile instructions across two approaches: Dockerfiles generated without topics and Dockerfiles generated with *MiDKo* The results indicate that *MiDKo* consistently outperforms the approach without topics in most instructions. For MiDKo, the `FROM` instruction achieves the highest precision (0.90) and recall (0.81). Other key instructions, such as `RUN` (precision: 0.63, recall: 0.63), `WORKDIR` (precision: 0.61, recall: 0.60), and `EXPOSE` (precision: 0.56, recall: 0.54), also show improvements compared to their counterparts in Dockerfiles generated without topics. While instructions such as `VOLUME` and `ADD` remain unused in both approaches (precision and recall less than 0.10), the improvements observed in frequently used instructions suggest that *MiDKo* effectively generates more precise and comprehensive Dockerfiles.

Table 4.2 Comparison of Precision and Recall for different Dockerfile instructions

Instruction	**Docker_without_topics**		**Docker_MiDKo**	
	Precision	Recall	Precision	Recall
`FROM`	0.86	0.81	0.90	0.81
`ENV`	0.52	0.48	0.58	0.51
`EXPOSE`	0.52	0.49	0.56	0.54
`WORKDIR`	0.56	0.56	0.61	0.60
`RUN`	0.55	0.59	0.63	0.63
`USER`	0.52	0.48	0.62	0.56
`COPY`	0.47	0.48	0.50	0.54
`ENTRYPOINT`	0.50	0.49	0.53	0.53
`HEALTHCHECK`	0.30	0.30	0.42	0.42
`LABEL`	0.29	0.26	0.35	0.30
`ARG`	0.29	0.27	0.27	0.26
`VOLUME`	0.08	0.08	0.10	0.10
`ADD`	0.10	0.07	0.10	0.06

Table 4.3 presents a comparative analysis of mean precision and recall for key components in Kubernetes manifests. Exposed Ports achieves the highest precision (0.95) and recall (0.93), indicating that port definitions are well-identified with minimal errors. Similarly, Selector demonstrates strong performance (precision: 0.85, recall: 0.80), reflecting its clear role in Kubernetes service discovery. Conversely, Replicas exhibits lower precision (0.30) and recall (0.25), suggesting challenges in accurately predicting the correct number of replicas since it is user specific. The Container Image component also shows moderate precision (0.65) but lower

recall (0.40), implying that while identified images are often correct, some valid images might be missed.

Table 4.3 Comparison of Mean Precision and Recall for different Kubernetes manifest components

Component	Mean Precision	Mean Recall
Container Image	0.65	0.40
Exposed Ports	0.95	0.93
Protocol	0.60	0.58
Replicas	0.30	0.25
Selector	0.85	0.80
Service Type	0.50	0.48

4.4.2 Qualitative Evaluation

To assess the effectiveness of our approach, we conducted a qualitative evaluation through a survey involving 11 participants, including developers, DevOps engineers and researchers. The evaluation encompassed three projects and 39 Docker and K8s manifest files. The evaluation focused on three key aspects: correctness, readability, and usefulness.

4.4.2.1 Participant Demographics

The participant demographics highlight a balanced representation of professionals from various backgrounds. Out of the 11 participants, 27% (3/11) were researchers, while the majority, 73% (8/11), were industry professionals, including 55% (6/11) developers and 27% (3/11) DevOps engineers. Regarding geographical distribution, 45% (5/11) of the participants were from North America, 27% (3/11) from Africa, and 27% (3/11) from Europe. This diversity

ensures a comprehensive evaluation by capturing perspectives from both academic researchers and industry practitioners across different regions. Additionally, participants had varying levels of experience, ranging from 3 to 15 years in software development, with an average of 6 years of experience in Docker and Kubernetes. The project assignments were evenly distributed, with 36% (4/11) evaluating PR1, 27% (3/11) working on PR2, and 36% (4/11) assessing PR3. This distribution ensures that the generated configurations were tested across different project contexts, providing a well-rounded qualitative assessment.

Table 4.4 Participant Demographics

ID	Position	Experience (Years)	Location	Project
P1	Developer	4	Africa	PR1
P2	DevOps Engineer	8	North America	PR2
P3	Researcher	9	North America	PR1
P4	Developer	5	Europe	PR3
P5	DevOps Engineer	10	Africa	PR3
P6	Researcher	7	North America	PR1
P7	Developer	8	North America	PR2
P8	Developer	6	Europe	PR1
P9	DevOps Engineer	5	Europe	PR2
P10	Developer	3	North America	PR3
P11	Developer	4	Africa	PR3

4.4.2.2 Survey Conduction

To ensure a structured evaluation, we prepared three systems and selected one representative microservice from each. For each microservice, we provided three versions of the corresponding Dockerfile: One generated using MiDKO, another generated without topics for comparison, and the original Dockerfile from the system. Participants were not informed which files were generated and which were original to maintain unbiased assessments. Additionally, for Kubernetes configurations, we provided two generated YAML files (deployment.yaml and service.yaml) alongside their corresponding original versions. To collect responses, we shared a Google Form link [2] with the participants, along with access to a corresponding Google Drive folder containing the evaluation files. This approach ensured that participants could independently review and assess the correctness, readability, and usefulness of the configurations.

Figure 4.2 Survey results for correctness, readability, and usefulness

[2] https://shorturl.at/NHSmW

4.4.2.3 Correctness

Comparison of Correctness Ratings MiDKo-generated Dockerfiles received strong correctness ratings, with 3 out of 11 participants (27%) rating them as Excellent (5/5), as shown in figure 4.2, 5 (45%) as Good (4/5), and only 1 (9%) as Poor (2/5). This suggests that *MiDKo* effectively captures key configuration elements, ensuring a high level of correctness. Dockerfiles generated without topics showed a slightly lower correctness perception, with only 2 participants (18%) rating them as Excellent, and 5 (45%) as Good, but with a higher proportion in the lower ratings. This suggests that removing topic-based modeling leads to a loss of structural accuracy, making configurations less precise. Original Dockerfiles received the highest correctness ratings, with 4 participants (36%) rating them as Excellent, and none rating them as Poor, reinforcing the perception that manually created configurations are more reliable. For Kubernetes configurations, MiDKo-generated files were rated favorably, with 4 participants (36%) assigning an Excellent rating, 4 (36%) as Good, and none giving a Poor rating, indicating that the generated Kubernetes manifests closely follow best practices. Original Kubernetes configurations had a similar distribution, with 2 participants (18%) rating them as Excellent, 5 (45%) as Good, and 1 (9%) as Poor, demonstrating that MiDKo-generated Kubernetes configurations can perform comparably to ground truth files.

4.4.2.4 Readability

As shown in figure 4.2, MiDKo-generated Dockerfiles were rated the highest in readability, with 9 out of 11 participants (82%) rating them as Excellent (5/5), 1 (9%) as Good (4/5), and only 1 (9%) as Average (3/5). This suggests that MiDKo's structured approach enhances clarity and organisation. Dockerfiles generated without topics had lower readability ratings, with only 2 participants (18%) rating them as Excellent, while 5 (45%) rated them as Good and 3 (27%) as Average. This indicates that topic-based generation significantly contributes to readability by ensuring a well-structured format. Original Dockerfiles had a wider distribution of ratings, with no Excellent scores, 1 participant (9%) rating them as Good, 4 (36%) as Average, and 4 (36%)

as Poor (2/5). This suggests that manually written Dockerfiles can often be inconsistent or lack documentation, reducing their readability.

Kubernetes configurations generated by *MiDKo* also received strong readability scores, with 6 participants (55%) rating them as Excellent, 4 (36%) as Good, and only 1 (9%) as Poor. This demonstrates that *MiDKo* effectively structures Kubernetes manifests to improve clarity. Original Kubernetes configurations, similar to Dockerfiles, exhibited lower readability scores, with no Excellent ratings, 1 (9%) as Good, 3 (27%) as Average, and 5 (45%) as Poor (2/5). This confirms that manually written manifests often suffer from readability issues due to a lack of comments.

4.4.2.5 Usefulness

As shown in Figure 4.2, MiDKo-generated Dockerfiles received strong usefulness ratings, with 5 participants (45%) rating them as Excellent (5/5), 3 (27%) as Good (4/5), and only 1 (9%) as Poor (2/5). This suggests that *MiDKo* produces configurations that align well with user expectations. Dockerfiles generated without topics were perceived as slightly less useful, with only 1 participant (9%) rating them as Excellent, while 5 (45%) rated them as Good and 3 (27%) as Average. This indicates that topic-based modeling contributes to practical applicability. Original Dockerfiles showed a more mixed perception of usefulness, with no Excellent ratings, 2 (18%) as Good, 5 (45%) as Average, and 2 (18%) as Poor. This suggests that some manually written configurations may lack standardisation or optimisation.

Kubernetes configurations generated by *MiDKo* received strong usefulness ratings, with 6 participants (55%) rating them as Excellent, 2 (18%) as Good, and none giving Poor ratings. This demonstrates that MiDKo-generated manifests align well with industry practices. Original Kubernetes configurations had a wider spread, with no Excellent ratings, 4 (36%) as Good, 3 (27%) as Average, and 3 (27%) as Poor. This suggests that manually written Kubernetes configurations often vary in quality and completeness.

4.5 Discussion

This section presents an analysis of our findings and their implications. We discuss key insights derived from our study and examine potential limitations that could impact the validity of our results.

4.5.1 Discussion and Recommendations

Our evaluation results provide several key insights into the effectiveness of *MiDKo* for generating Dockerfiles and Kubernetes manifests. The discussion is structured around three main themes: (1) improvements in syntactic correctness and best-practice adherence, (2) impact of topic-based retrieval on configuration accuracy, and (3) qualitative feedback from practitioners.

4.5.1.1 Syntactic Correctness and Best-Practice Adherence

The quantitative evaluation demonstrates that MiDKo-generated Dockerfiles and Kubernetes manifests exhibit higher syntactic correctness compared to manually written configurations. The reduction in syntax errors and warnings suggests that *MiDKo* effectively enforces best practices.

For Dockerfiles, the use of Hadolint to assess correctness revealed that MiDKo-generated files contained significantly fewer errors than their manually written counterparts. In particular, common misconfigurations such as inefficient use of `COPY` commands (DL3020) and missing health check instructions (DL4000) were notably reduced. Similarly, Kubernetes configurations generated by *MiDKo* had fewer critical misconfigurations, such as missing security context specifications and improper resource allocation. These findings indicate that the knowledge-driven approach embedded in *MiDKo* leads to more standardised and robust configurations.

While large language models are effective at generating syntactically correct configurations, manually written open-source configurations may suffer from accumulated inconsistencies. Since multiple developers contribute to open-source projects over time, some configurations become lengthy due to redundant or outdated commands that are not always removed. This

can result in excessive complexity and maintenance challenges. In contrast, MiDKo-generated configurations remain concise and structured, reducing unnecessary bloat while adhering to best practices.

4.5.1.2 Precision and Recall-Based Similarity Analysis

Table 4.2 presents the detailed precision and recall values for each instruction in the Dockerfiles. We compare the precision and recall of different Dockerfile instructions across two approaches: Dockerfiles generated without topics and Dockerfiles generated with *MiDKo*. The results indicate that *MiDKo* consistently outperforms the approach without topics in most instructions. For MiDKo, the `FROM` instruction achieves the highest precision (0.90) and recall (0.81). Other key instructions, such as `RUN` (precision: 0.63, recall: 0.63), `WORKDIR` (precision: 0.61, recall: 0.60), and `EXPOSE` (precision: 0.56, recall: 0.54), also show improvements compared to their counterparts in Dockerfiles generated without topics. While instructions such as `VOLUME` and `ADD` remain unused in both approaches (precision and recall less than 0.10), the improvements observed in frequently used instructions suggest that *MiDKo* effectively generates more precise and comprehensive Dockerfiles.

The variation across instructions reflects their inherent predictability. Common instructions like `FROM`, `RUN`, and `WORKDIR` follow relatively stable patterns across systems, which explains their higher precision and recall. In contrast, instructions such as `COPY`, `LABEL`, or `HEALTHCHECK` are project-specific and thus harder to infer automatically. Rarely used instructions such as `VOLUME` and `ADD` contribute to the lower overall averages.

Table 4.3 presents a comparative analysis of mean precision and recall for key components in Kubernetes manifests. Exposed Ports achieves the highest precision (0.95) and recall (0.93), indicating that port definitions are well-identified with minimal errors. Similarly, Selector demonstrates strong performance (precision: 0.85, recall: 0.80), reflecting its clear role in Kubernetes service discovery. Conversely, Replicas exhibits lower precision (0.30) and recall (0.25), since scaling requirements are user-dependent and not derivable from code alone.

Container Image also shows moderate precision (0.65) but lower recall (0.40), as registry paths and tags are often system-specific.

4.5.1.3 Qualitative Insights from Practitioners

The survey results from software engineers, DevOps professionals, and researchers provide further validation of MiDKo's effectiveness. Three key insights emerged from the qualitative evaluation:

4.5.1.3.1 Readability and Maintainability

MiDKo-generated files were found to be highly readable, as they often include structured, line-by-line comments that improve clarity. In contrast, many open-source configurations lacked adequate documentation, making them harder to understand. Additionally, we observed that in open-source configurations, commands were often written in a compact multi-line format (e.g., `RUN` commands spanning multiple lines using backslashes), which, while efficient, reduces readability. The absence of clear comments in such cases further exacerbates the challenge of maintaining these configurations over time.

4.5.1.3.2 Correctness Perception

While MiDKo-generated Dockerfiles and Kubernetes manifests received high correctness ratings, a minority of participants expressed concerns about over-reliance on inferred defaults. This feedback suggests that while *MiDKo* effectively captures best practices, further refinement is needed to allow for user-specified overrides where necessary.

4.5.1.3.3 Practical Usefulness

The majority of participants found MiDKo-generated configurations to be useful for real-world deployment scenarios. However, some participants noted that while MiDKo-generated configurations are well-structured, they sometimes lack project-specific optimisations that

experienced developers might include manually. This suggests a potential opportunity to enhance *MiDKo* by incorporating feedback loops that allow users to fine-tune generated configurations.

4.5.2 Threats to Validity

Despite the promising results of MiDKo, our approach is subject to several potential threats that could impact the validity of our findings. We classify these threats into internal, external, and construct validity concerns.

4.5.2.1 Internal Validity

Internal validity refers to factors that may affect the correctness and reliability of our results. These threats arise from potential biases in topic extraction, retrieval mechanisms, and the performance of large language models.

- *MiDKo* relies on BERTopic to extract relevant patterns from Dockerfiles. However, variations in Dockerfile syntax, noise in the dataset, or an insufficient number of training examples for certain configurations could lead to suboptimal topic clustering. This could result in misclassification, affecting the retrieval of relevant deployment templates.
- The retrieval process plays a crucial role in RAG-based generation. BM25 is used for ranking and selecting the most relevant topic from the knowledge base, but its effectiveness depends on query quality. Poorly constructed queries or an incomplete knowledge base may lead to the selection of less relevant topics, impacting the correctness of generated deployment artifacts. To mitigate this threat, we define a structured query template to ensure consistency in query formulation, improving retrieval accuracy and reducing ambiguities.
- Our evaluation focuses on syntactic correctness, alignment with the ground truth, and usability of generated artifacts. However, we do not conduct large-scale empirical testing in live production environments. As a result, factors such as performance under high-load conditions, security vulnerabilities, and long-term maintainability remain unverified.

- One key limitation of our study lies in the use of the same dataset for both topic extraction and evaluation. This overlap may introduce bias, as the generated deployment artifacts could be influenced by examples already seen during the construction of the retrieval-based knowledge base. To mitigate this risk and assess generalisability, we employed a 10-fold cross-validation strategy at the Dockerfile level. This approach ensures that each Dockerfile is evaluated only when it has not been used in the training set, thereby reducing the potential for data leakage and overfitting.

4.5.2.2 External Validity

External validity concerns the generalizability of *MiDKo* to broader microservices ecosystems and deployment environments.

- The dataset used in this study consists of 378 repositories and 1,607 Dockerfiles, primarily featuring widely used technologies such as Python, Node.js, and Java. While this provides good coverage of common deployment practices, it may not fully generalise to less common architectures, frameworks, or industry-specific deployment scenarios.
- Kubernetes deployment practices vary across organisations. While *MiDKo* generates standard Kubernetes manifests, it does not explicitly account for more advanced deployment strategies, such as custom security policies, service meshes, or cloud provider-specific optimisations.

4.5.2.3 Construct Validity

Construct validity pertains to the extent to which our evaluation accurately measures the effectiveness of MiDKo.

- The evaluation is primarily based on syntactic correctness, functional correctness, and usability. While these metrics are useful, they do not capture aspects such as security best practices, resilience under failure scenarios, or compliance with specific DevOps policies, which are critical in real-world deployments.

- The usability assessment involves manual inspection of generated artifacts, which introduces an element of subjectivity. Although predefined evaluation criteria are used to mitigate bias, different users may have varying perceptions of correctness and usability.

4.6 Conclusion

In this chapter, we presented *MiDKo*, an automated approach for generating Dockerfiles and Kubernetes manifests using retrieval-augmented large language models. This approach addresses several key challenges in microservices deployment, including the lack of automation in configuration generation, the rigidity of template-based tools, and the absence of context-aware synthesis mechanisms. By constructing a knowledge base from 1,607 Dockerfiles collected across 317 open-source microservice projects and extracting 54 deployment topics through BERTopic, MiDKo enables large language models to produce tailored, high-quality deployment artifacts.

The evaluation demonstrated that MiDKo substantially improves the syntactic correctness, structure, and practical relevance of generated configurations. Specifically, 91% of the Dockerfiles and 70% of the Kubernetes manifests generated by MiDKo were syntactically clean, outperforming both manually written and non-topic-guided configurations. For Dockerfiles, results were computed using a 10-fold cross-validation setup, ensuring that the evaluation was both robust and generalisable. Instruction-level precision and recall metrics confirmed that the approach effectively captures common deployment patterns, while a practitioner survey highlighted improvements in readability and usefulness. Participants with industry and research backgrounds consistently rated MiDKo-generated artifacts as clearer and more usable compared to their open-source counterparts.

Despite these promising results, MiDKo still has limitations. It does not currently support multistage builds, and some Dockerfile instructions such as `ARG` and `ADD` are underused or mispredicted. The approach also depends on existing data and does not yet adapt to emerging tools or deployment paradigms without manual updates. Nevertheless, the findings confirm that

MiDKo is a practical and effective tool for reducing the manual effort required in containerisation and deployment of microservices.

Together with the approaches introduced in previous chapters—*MicroMiner* for service identification and *MicroPacker* for service packaging—*MiDKo* completes the proposed AI-driven migration pipeline from monolithic systems to microservices. In the next and final chapter, we summarise the main contributions of this thesis, reflect on its limitations, and discuss promising directions for future research.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Summary

This thesis set out to investigate whether the migration from monolithic systems to microservices can be fully automated, while ensuring accuracy and minimising manual effort, by leveraging machine learning and large language models across three core technical phases: service identification, service packaging, and deployment configuration.

To this end, we introduced and evaluated an ML-based migration framework composed of three dedicated approaches, each targeting a key phase of the process:

- **The Identification phase:** ***MicroMiner*** addresses the identification phase. It uses supervised learning to classify service types and semantic-aware clustering to generate cohesive microservice candidates. Evaluated on four open-source monoliths, *MicroMiner* achieved a precision of 68.15% and a recall of 77%, outperforming two state-of-the-art baselines in terms of modularity, architectural relevance, and functional independence.
- **The Packaging phase:** ***MicroPacker*** targets the packaging phase. It leverages LLMs to automate source code refactoring, API generation, and integration of microservice design patterns. Applied to two real-world systems (PetClinic and FXML-POS), *MicroPacker* reduced manual packaging effort by over 60%, while maintaining a correctness score of over 85% for generated artifacts across multiple steps. Experts confirmed the architectural soundness and functional completeness of the produced services.
- **The Deployment phase:** ***MiDKo*** supports the deployment phase by generating Dockerfiles and Kubernetes manifests using a retrieval-augmented generation (RAG) strategy. Built on topic modeling and LLM prompting, *MiDKo* produced syntactically valid and complete configurations for systems used in evaluation. Generated artifacts met over 80% of precision and recall, with expert assessments confirming their reusability and extensibility.

The evaluation of this framework across multiple systems confirms that, while complete automation in every scenario remains constrained by factors such as system heterogeneity, domain-specific requirements, and runtime complexity, a high degree of automation is both achievable and effective. ML- and LLM-based techniques proved capable of significantly reducing expert intervention, improving decomposition quality, and accelerating artifact generation across all three migration phases.

With respect to the thesis statement, we conclude that monolith-to-microservices migration can be largely automated using machine learning and large language models, achieving strong accuracy and substantial reductions in manual effort. While this thesis does not directly optimise for performance efficiency (e.g., runtime or cost), it lays the foundation for scalable and tool-supported modernisation pipelines. Future work can extend this foundation to address behavioral verification, database decomposition, and resource-aware deployment.

Together, the results of this thesis represent a practical and forward-looking contribution toward enabling fully automated software migration.

5.2 Discussion

This thesis proposes an ML-based framework for automating the migration of monolithic systems to microservices, focusing specifically on the phases of service identification, packaging, and deployment. The framework is grounded in the analysis of source code, which remains the most accessible and reliable artifact in legacy systems. By relying on structural dependencies, method interactions, and semantic relationships between classes, the approach enables a transformation that preserves core business logic while improving modularity and maintainability.

The decision to base the entire framework on source code analysis was guided by the practical need to support real-world systems that often lack up-to-date documentation or runtime instrumentation. While effective in many scenarios, this code-driven strategy excludes insights from high-level domain models or business processes. Top-down strategies—which rely on business use cases or domain-driven design—can complement code-level perspectives by reinforcing meaningful

service boundaries. Although not explored in this work, future research could integrate such information to enhance the contextual alignment of microservices.

Our work on microservice identification explored multiple levels of granularity. In particular, we developed and evaluated a method-level approach, *Magnet*, which leverages graph neural networks to analyse fine-grained interactions between individual methods. While this approach allowed for precise detection of behavioral patterns, it often resulted in highly fragmented service candidates that were harder to manage, interpret, and package. In contrast, our class-level approach, implemented in *MicroMiner*, focused on grouping semantically and structurally related classes, leading to more coherent, modular, and domain-aligned microservices. This granularity provided a better balance between precision and manageability, making it more suitable for subsequent automation phases such as packaging and deployment, where clarity of service boundaries are crucial. Another key decision in this thesis is the adoption of a refactoring-based migration strategy. Instead of rewriting the system from scratch, we focus on extracting and restructuring reusable code into well-defined microservices. This decision is based on the assumption that much of the legacy system's logic remains valid and valuable. However, in cases of poor code quality or obsolete technologies, rewriting part of the system may be necessary. Hybrid migration strategies that combine refactoring and partial rewriting should be explored further, with decision criteria grounded in software metrics such as cohesion, coupling, and maintainability.

Code quality has a noticeable impact on the success of automated migration. In our evaluation, systems with well-structured classes, low interdependence, and meaningful naming led to higher-quality outputs across all phases. In contrast, tangled or inconsistent codebases require more manual intervention. These observations suggest that integrating code quality assessment into migration tools could help flag components that may need special attention or recommend rewrites instead of refactoring.

System size also plays an important role in shaping migration outcomes. While our framework can scale to large codebases without fundamental limitations, larger systems naturally produce

a greater number of microservice candidates and deployment artifacts. This increases the burden of validation and oversight, as developers must ensure that the generated services remain coherent and manageable. In such contexts, automation becomes even more valuable, since manually handling thousands of lines of code or dozens of services would be infeasible. At the same time, complementary strategies such as incremental migration or clustering services into higher-level domains may help organisations deal with the cognitive overhead introduced by very large systems. Thus, system size does not constrain the applicability of our approach, but it does shape the way results are consumed and refined in practice.

Although this work targets microservices as the final architecture, we believe that the techniques introduced—particularly in packaging and deployment—can be extended to support other paradigms such as serverless or event-driven systems. In such cases, the identification step would need to detect finer-grained units (e.g., functions or event handlers), but many aspects of packaging and deployment would remain applicable. This highlights the flexibility and adaptability of the proposed automation strategies.

A recurring practical question is which migration approach an industrial partner should adopt. Our results indicate that this depends on several contextual dimensions. The first is the available artifacts: code-only systems benefit from code-centric strategies, whereas additional documentation or traces can improve alignment with business processes. A second is the level of automation: fully automated pipelines maximise efficiency but reduce user input, while semi-automated approaches allow more control. The system's quality and complexity also matter, as structured codebases are more amenable to automation than tangled ones. Equally important are the organisation's maturity and expertise, with experienced teams better positioned to adopt full automation. Migration goals further influence the choice: speed and cost reduction favour automation, while maintainability and domain alignment may require oversight. Finally, the deployment environment shapes the usefulness of generated artifacts, with cloud-native settings benefiting most. Taken together, these dimensions show that there is no single best approach, but rather a spectrum of approaches that organisations can adapt to their context.

Finally, while the goal of this thesis is to reduce the manual effort involved in migration, we recognise that automation does not eliminate the need for developer oversight. Generated artifacts must be reviewed, validated, and adapted to system-specific constraints. Future tools should aim to better balance automation with developer control, possibly through explainable AI techniques or interactive refinement mechanisms. The ultimate goal is not to replace human judgment, but to support it, making the migration process faster, safer, and more accessible.

5.3 Future Work

While this thesis introduces an AI-driven framework to automate key phases of monolith-to-microservices migration, i.e., service identification, code packaging, and deployment configuration, several important areas remain open for further investigation.

A significant avenue concerns the large-scale validation of the proposed automation framework. While this work demonstrates the effectiveness of ML-based migration in controlled scenarios, applying the approach in industrial settings would help assess its impact at scale. Future research can focus on measuring reductions in development time, migration costs, and operational overhead in industrial systems. Such evaluation would also provide insights into the trade-offs between automation and human oversight, while identifying areas where developer intervention remains critical. Understanding the long-term maintainability, team productivity, and system robustness of automated migrations would also help establish trust and adoption in the industry.

A primary direction for future research involves the decomposition of monolithic databases. This thesis does not address the migration of data storage, which is essential for achieving a fully decoupled microservices architecture. Dividing a monolithic database into service-specific introduces challenges related to data ownership, consistency, distributed transactions, and schema transformation. Proposing frameworks that combine static analysis, query tracing, and machine learning could enable automated and semantically meaningful database decomposition, aligned with microservice boundaries. Managing data access across services and ensuring transactional guarantees in distributed settings are key issues that require specialised attention.

The current work assumes a REST-based microservices architecture as the primary modernisation target. Beyond microservices, alternative architectural targets also merit exploration. For instance, serverless architectures enable function-level deployment and dynamic scaling; event-driven systems promote decoupling through asynchronous messaging; and service mesh–based architectures move cross-cutting concerns (e.g., security, observability, routing) into the infrastructure layer. With the increasing adoption of edge-native and cloud-native systems, there is a growing need to investigate migration strategies tailored to these emerging deployment models. Additionally, hybrid migration strategies that temporarily combine multiple architectural styles can offer more flexible and incremental modernisation paths, reducing risk while adapting to organisational and technical constraints.

Finally, there is a pressing need for standardised benchmarks and evaluation methods. At present, the lack of common datasets and metrics makes it difficult to compare the effectiveness of different migration approaches. This could contribute by curating annotated monolithic systems with ground-truth microservices decompositions and defining standardised metrics for automation quality, scalability, and maintainability. Establishing reproducible pipelines and public benchmarks would facilitate more rigorous comparisons across tools and encourage progress in the field.

BIBLIOGRAPHY

Abdellatif, M., Hecht, G., Mili, H., Elboussaidi, G., Moha, N., Shatnawi, A., Privat, J. & Guéhéneuc, Y.-G. (2018). State of the practice in service identification for soa migration in industry. *Service-Oriented Computing: 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings 16*, pp. 634–650.

Abdellatif, M., Tighilt, R., Moha, N., Mili, H., El-Boussaidi, G., Privat, J. & Guéhéneuc, Y. (2020a). A Type-Sensitive Service Identification Approach for Legacy-to-SOA Migration. *Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings*, 12571(Lecture Notes in Computer Science), 476–491. doi: 10.1007/978-3-030-65310-1_34.

Abdellatif, M., Tighilt, R., Moha, N., Mili, H., El Boussaidi, G., Privat, J. & Guéhéneuc, Y.-G. (2020b). A type-sensitive service identification approach for legacy-to-SOA migration. *International Conference on Service-Oriented Computing*, pp. 476–491.

Abdellatif, M., Shatnawi, A., Mili, H., Moha, N., El-Boussaidi, G., Hecht, G., Privat, J. & Guéhéneuc, Y. (2021a). A taxonomy of service identification approaches for legacy software systems modernization. *J. Syst. Softw.*, 173, 110868. doi: 10.1016/j.jss.2020.110868.

Abdellatif, M., Shatnawi, A., Mili, H., Moha, N., El Boussaidi, G., Hecht, G., Privat, J. & Guéhéneuc, Y.-G. (2021b). A taxonomy of service identification approaches for legacy software systems modernization. *Journal of Systems and Software*, 173, 110868.

Abdullah, M., Iqbal, W. & Erradi, A. (2019a). Unsupervised learning approach for web application auto-decomposition into microservices. *Journal of Systems and Software*, 151, 243–257.

Abdullah, M., Iqbal, W., Erradi, A. & Bukhari, F. (2019b). Learning Predictive Autoscaling Policies for Cloud-Hosted Microservices Using Trace-Driven Modeling. *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 119–126.

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

Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Yilmaz, M., Buckley, J. & Clarke, P. (2023b). Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. *IEEE Transactions on Software Engineering*, 4213-4242.

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S. et al. (2023). Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 1–100.

Ahmadvand, M. & Ibrahim, A. (2016). Requirements reconciliation for scalable and secure microservice (de) composition. *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*, pp. 68–73.

Al-Debagy, O. & Martinek, P. (2019). A new decomposition method for designing microservices. *Periodica Polytechnica Electrical Engineering and Computer Science*, 63, 274-281.

Al-Debagy, O. & Martinek, P. (2021a). A microservice decomposition method through using distributed representation of source code. *Scalable Computing: Practice and Experience*, 22(1), 39–52.

Al-Debagy, O. & Martinek, P. (2021b). Dependencies-based microservices decomposition method. *Scalable Computing: Practice and Experience*, 22, 39–52.

Allamanis, M., Barr, E. T., Devanbu, P. & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4), 1–37.

Alshammari, A., Almadhor, A., Qasem, S., Alkhateeb, J. & Amjad, K. (2023). High-performance computing-enabled probabilistic framework for migration from monolithic to microservices architecture using genetic algorithms. *Soft Computing*, 793.

Amari, S.-i. & Wu, S. (1999). Improving support vector machine classifiers by modifying kernel functions. *Neural Networks*, 12(6), 783–789.

Amoroso d'Aragona, D., Bakhtin, A., Li, X., Su, R., Adams, L., Aponte, E., Boyle, F., Boyle, P., Koerner, R., Lee, J. et al. (2024). A dataset of microservices-based open-source projects. *Proceedings of the 21st International Conference on Mining Software Repositories*, pp. 504–509.

Bajaj, D., Bharti, U., Gupta, I., Gupta, P. & Yadav, A. (2024). GTMicro—microservice identification approach based on deep NLP transformer model for greenfield developments. *International Journal of Information Technology*, 2751–2761.

Bajaj, D., Bharti, U., Goel, A. & Gupta, S. C. (2020). Partial migration for re-architecting a cloud native monolithic application into microservices and FaaS. 111–124.

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D. A. & Lynn, T. (2018). Microservices migration patterns. *Software: Practice and Experience*, 48(11), 2019–2042.

Bezdek, J. C., Ehrlich, R. & Full, W. (1984). FCM: The fuzzy c-means clustering algorithm. *Computers & geosciences*, 10(2-3), 191–203.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R. & Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10), P10008.

Brito, M., Cunha, J. & Saraiva, J. (2021). Identification of microservices from monolithic applications through topic modelling. *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pp. 1409–1418.

Bruneliere, H., Cabot, J., Dupé, G. & Madiot, F. (2014). Modisco: A model driven reverse engineering framework. *IST*, 56(8), 1012–1032.

Bushong, V., Abdelfattah, A. S., Maruf, A. A., Das, D., Lehman, A., Jaroszewski, E., Coffey, M., Cerny, T., Frajtak, K., Tisnovsky, P. et al. (2021). On microservice analysis and architecture evolution: A systematic mapping study. *Applied Sciences*, 11(17), 7856.

Cai, Y., Han, B., Su, J. & Wang, X. (2021). TraceModel: An Automatic Anomaly Detection and Root Cause Localization Framework for Microservice Systems. *2021 17th International Conference on Mobility, Sensing and Networking (MSN)*, 512–519.

Calçado, P. [Accessed: 2025-05-23]. (2012). Building Products at SoundCloud, Part 1: Dealing with the Monolith. Retrieved from: https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith.

Cao, L. & Zhang, C. (2022). Implementation of Domain-oriented Microservices Decomposition based on Node-attributed Network. *Proceedings of the 2022 11th International Conference on Software and Computer Applications*, 136–142.

Capuano, R. & Muccini, H. (2022). A systematic literature review on migration to microservices: a quality attributes perspective. *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pp. 120–123.

Chen, J., Liu, F., Jiang, J., Zhong, G., Xu, D., Tan, Z. & Shi, S. (2023a). TraceGra: A trace-based anomaly detection for microservice using graph deep learning. *Comput. Commun.*, 204, 109–117.

Chen, L., Guang, M., Wang, J. & Yan, C. (2023b). Dynamic and static feature-aware microservices decomposition via graph neural networks. *2023 Knowledge Science, Engineering and Management (KSEM)*, 150–163.

Chen, L., Dang, Q., Chen, M., Sun, B., Du, C. & Lu, Z. (2023c). BertHTLG: Graph-Based Microservice Anomaly Detection Through Sentence-Bert Enhancement. *Web Information Systems and Applications*, pp. 427–439.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G. et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 1–35.

Chen, R., Ren, J., Wang, L., Pu, Y., Yang, K. & Wu, W. (2022). MicroEGRCL: An Edge-Attention-Based Graph Neural Network Approach for Root Cause Localization in Microservice Systems. *Service-Oriented Computing: 20th International Conference, ICSOC 2022*, 13747, 264–272.

Chou, J., Al-Masri, E., Kanzhelev, S. & Fattah, H. (2021). Detecting Security and Privacy Risks in Microservices End-to-End Communication Using Neural Networks. *2021 IEEE 4th International Conference on Knowledge Innovation and Invention (ICKII)*, 105–110.

Chow, K.-H., Deshpande, U., Seshadri, S. & Liu, L. (2022). DeepRest: Deep Resource Estimation for Interactive Microservices. *EuroSys 2022 - Proceedings of the 17th European Conference on Computer Systems*, 181–198.

CLI, K. S. (2025). Kustomize - Kubernetes Native Configuration Management. Retrieved from: https://kustomize.io.

Cloud Native Computing Foundation. (2021). *CNCF Annual Survey 2021*. Retrieved from: https://www.cncf.io/wp-content/uploads/2022/02/CNCF-AR_FINAL-edits-15.2.21.pdf.

Cohen, J. (1968). Weighted Kappa: Nominal Scale Agreement Provision For Scaled Disagreement Or Partial Credit. *Psychological bulletin*, 70(4), 213.

Cooke, A., Smith, D. & Booth, A. (2012). Beyond PICO: the SPIDER Tool For Qualitative Evidence Synthesis. *Qualitative health research*, 22(10), 1435–1443.

Daoud, M., El Mezouari, A., Faci, N., Benslimane, D., Maamar, Z. & El Fazziki, A. (2020a). Towards an Automatic Identification of Microservices from Business Processes. *2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 42-47.

Daoud, M., Mezouari, A. E., Faci, N., Benslimane, D., Maamar, Z. & Fazziki, A. E. (2020b). Automatic Microservices Identification from a Set of Business Processes. *Smart Applications and Data Analysis*, 299–315.

Daoud, M., El Mezouari, A., Faci, N., Benslimane, D., Maamar, Z. & El Fazziki, A. (2021). A multi-model based microservices identification approach. *Journal of Systems Architecture*, 118, 102200.

Dehghani, M., Kolahdouz-Rahimi, S., Tisi, M. & Tamzalit, D. (2022). Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning. *Software and Systems Modeling*, 21, 1115–1133.

Desai, U., Bandyopadhyay, S. & Tamilselvam, S. (2021). Graph Neural Network to Dilute Outliers for Refactoring Monolith Application. *Thirty-fifth AAAI Conference on Artificial Intelligence, Thirty-third Conference on Innovative Applications of Artificial Intelligence and the Eleventh Symposium on Educational Advances in Artificial Intelligence*, 35, 72–80.

Developers, K. (2024). Kompose: Convert Docker Compose to Kubernetes. Retrieved from: https://kompose.io.

Di Francesco, P., Lago, P. & Malavolta, I. (2018). Migrating towards microservice architectures: an industrial survey. *2018 IEEE international conference on software architecture (ICSA)*, pp. 29–2909.

Ding, S., E, Y., Zhang, J., Li, L., Zhang, L. & Ge, J. (2024). Trace Anomaly Detection for Microservice Systems via Graph-based Semi-supervised Learning. *2024 27th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2375-2380.

Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R. & Safina, L. (2017). Microservices: How to make your application scale. *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pp. 95–104.

Du, Q., Xie, T. & He, Y. (2018). Anomaly Detection and Diagnosis for Container-Based Microservices with Performance Monitoring. *Lecture Notes in Computer Science*, 560–572.

Dybå, T. & Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9-10), 833–859.

Dyba, T., Dingsoyr, T. & Hanssen, G. K. (2007). Applying Systematic Reviews To Diverse Study Types: An Experience Report. *First international symposium on empirical software engineering and measurement (ESEM 2007)*, pp. 225–234.

El Boussaidi, G., Belle, A. B., Vaucher, S. & Mili, H. (2012). Reconstructing architectural views from legacy systems. *2012 19th Working Conference on Reverse Engineering*, pp. 345–354.

Enslen, E., Hill, E., Pollock, L. & Vijay-Shanker, K. (2009). Mining source code to automatically split identifiers for software analysis. *2009 6th IEEE International Working Conference on Mining Software Repositories*, pp. 71–80.

Eski, S. & Buzluca, F. (2018). An automatic extraction approach: transition to microservices architecture from monolithic application. *Proceedings of the 19th International Conference on Agile Software Development: Companion*, 1–6.

Faria, V. & Silva, A. R. (2023). Code vectorization and sequence of accesses strategies for monolith microservices identification. 19–33.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D. et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 1–12. doi: 10.48550/arXiv.2002.08155.

Floyd, R. W. (1962). On ambiguity in phrase structure languages. *Communications of the ACM*, 5(10), 526.

Fritzsch, J., Bogner, J., Zimmermann, A. & Wagner, S. (2018). From monolith to microservices: A classification of refactoring approaches. *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pp. 128–141.

Fritzsch, J., Bogner, J., Wagner, S. & Zimmermann, A. (2019a). Microservices migration in industry: intentions, strategies, and challenges. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 481–490.

Fritzsch, J., Bogner, J., Wagner, S. & Zimmermann, A. (2019b). Microservices migration in industry: intentions, strategies, and challenges. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 481–490.

Fritzsch, J., Bogner, J., Zimmermann, A. & Wagner, S. (2019c). From monolith to microservices: A classification of refactoring approaches. *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, March 5-6, 2018, Revised Selected Papers 1*, pp. 128–141.

Gan, Y., Liang, M., Dev, S., Lo, D. & Delimitrou, C. (2021). Sage: Practical and scalable ML-driven performance debugging in microservices. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 135-151.

Gan, Y., Zhang, Y., Hu, K., Cheng, D., He, Y., Pancholi, M. & Delimitrou, C. (2019). Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 19–33.

Gan, Y., Liang, M., Dev, S., Lo, D. & Delimitrou, C. (2022). Enabling Practical Cloud Performance Debugging with Unsupervised Learning. *Operating Systems Review (ACM)*, 56, 34–41.

Gao, X.-B., PEI, J.-h. & XIE, W.-x. (2000). A study of weighting exponent m in a fuzzy c-means algorithm. *Acta electronica sinica*, 28(4), 80–83.

Grootendorst, M. (2022). BERTopic: Neural topic modeling with a class-based TF-IDF procedure. *arXiv preprint arXiv:2203.05794*, 1–10.

Gysel, M., Kölbener, L., Giersche, W. & Zimmermann, O. (2016a). Service cutter: A systematic approach to service decomposition. *Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings 5*, pp. 185–200.

Gysel, M., Kölbener, L., Giersche, W. & Zimmermann, O. (2016b). Service Cutter: A Systematic Approach to Service Decomposition. *Lecture Notes in Computer Science*, 185–200.

hadolint. (2025). hadolint: Haskell Dockerfile Linter. Retrieved from: https://hub.docker.com/r/hadolint/hadolint.

Hassan, S., Ali, N. & Bahsoon, R. (2017). Microservice ambients: An architectural meta-modelling approach for microservice granularity. *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 1–10.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J. & Wang, H. (2023). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 1–79.

Hu, R., Peng, C., Wang, X. & Gao, C. (2025). An LLM-based Agent for Reliable Docker Environment Configuration. *arXiv preprint arXiv:2502.13681*, 1–25.

Huang, J., Yang, Y., Yu, H., Li, J. & Zheng, X. (2023). Twin Graph-Based Anomaly Detection via Attentive Multi-Modal Learning for Microservice System. *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 66–78.

Iyyer, M., Manjunatha, V., Boyd-Graber, J. & Daumé III, H. (2015). Deep unordered composition rivals syntactic methods for text classification. *Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers)*, pp. 1681–1691.

Jain, H., Zhao, H. & Chinta, N. R. (2004). A spanning tree based approach to identifying web services. *International Journal of Web Services Research (IJWSR)*, 1(1), 1–20.

Jin, W., Liu, T., Cai, Y., Kazman, R., Mo, R. & Zheng, Q. (2019). Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering*, 987-1007.

Joseph, C., Martin, J., Chandrasekaran, K. & Kandasamy, A. (2019). Fuzzy Reinforcement Learning based Microservice Allocation in Cloud Computing Environments. *TENCON 2019 - IEEE Region 10 Conference (TENCON)*, 1559–1563.

Kalia, A. K., Xiao, J., Lin, C., Sinha, S., Rofrano, J., Vukovic, M. & Banerjee, D. (2020). Mono2Micro: an AI-based toolchain for evolving monolithic enterprise applications to a microservice architecture. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1606–1610.

Kalske, M., Mäkitalo, N. & Mikkonen, T. (2018). Challenges when moving from monolith to microservice architecture. *Current Trends in Web Engineering: ICWE 2017 International Workshops, Liquid Multi-Device Software and EnWoT, practi-O-web, NLPIT, SoWeMine, Rome, Italy, June 5-8, 2017, Revised Selected Papers 17*, pp. 32–47.

Kamimura, M., Yano, K., Hatano, T. & Matsuo, A. (2018). Extracting Candidates of Microservices from Monolithic Application Code. *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, 571–580.

Kazanavičius, J. & Mažeika, D. (2019). Migrating legacy software to microservices architecture. *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pp. 1–5.

Khadka, R., Saeidi, A., Jansen, S. & Hage, J. (2013). A structured legacy to SOA migration process and its evaluation in practice. *2013 IEEE 7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pp. 2-11. doi: 10.1109/MESOCA.2013.6632729.

Khan, S. A., Abdullah, M., Iqbal, W., Butt, M. A., Bukhari, F. & Hassan, S.-U. (2023). Automatic Migration-Enabled Dynamic Resource Management for Containerized Workload. *IEEE Systems Journal*, 17, 2378–2389.

Kipf, T. N. & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)*, 1–14.

Kitchenham, B. A., Madeyski, L. & Budgen, D. (2022). SEGRESS: Software Engineering Guidelines For Reporting Secondary Studies. *IEEE Transactions on Software Engineering*, 1273-1298.

Klock, S., Van Der Werf, J. M. E., Guelen, J. P. & Jansen, S. (2017). Workload-based clustering of coherent feature sets in microservice architectures. *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 11–20.

Kong, H., Li, T., Ge, J., Zhang, L. & Li, L. (2024). Enhancing fault localization in microservices systems through span-level using graph convolutional networks. *Automated Software Engineering*, 1–26.

kube score. (2025). kube-score: Kubernetes object analysis. Retrieved from: https://kube-score.com/.

Lapuz, N., Clarke, P. & Abgaz, Y. (2021). Digital Transformation and the Role of Dynamic Tooling in Extracting Microservices from Existing Software Systems. *European Conference on Software Process Improvement*, pp. 301–315.

Lewis, G., Morris, E., O'Brien, L., Smith, D. & Wrage, L. (2005). SMART: The service-oriented migration and reuse technique. *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, pp. 222–229.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T. et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33, 9459–9474.

Li, M., Tang, D., Wen, Z. et al. (2021a). Microservice anomaly detection based on tracing data using semi-supervised learning. *International Conference on Artificial Intelligence and Big Data*, 38–44.

Li, N., Tan, Y., Wang, X., Li, B. & Luo, J. (2022). SCORE: A Resource-Efficient Microservice Orchestration Model Based on Spectral Clustering in Edge Computing. *Service-oriented Computing (ICSOC 2022)*, 13740, 186–202.

Li, R., Du, M., Chang, H., Mukherjee, S. & Eide, E. (2021b). Deepstitch: Deep Learning for Cross-Layer Stitching in Microservices. *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*, 25–30.

Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J. & Babar, M. A. (2021c). Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. *Information and software technology*, 131, 106449.

Liang, X., Li, L. & Peng, H. (2024). Unsupervised Microservice Log Anomaly Detection Method Based on Graph Neural Network. *International Conference on Swarm Intelligence*, 197–208.

Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C. et al. (2024). Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 1–53.

Liu, B., Lu, J., Zhang, F., Zhang, W. & Wang, M. (2020a). Method of Microservices Division for Complex Business Management System Based on Dual Clustering. *2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, 2259–2268.

Liu, J. & Zhang, C. (2024). Migrating Monolith System to Microservices with Directed Graph Attention Neural Network. *Third International Conference on High Performance Computing and Communication Engineering (HPCCE 2023)*, 13073, 1–6.

Liu, P., Xu, H., Ouyang, Q., Jiao, R., Chen, Z., Zhang, S., Yang, J., Mo, L., Zeng, J., Xue, W. & Pei, D. (2020b). Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks. *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 48–58.

Luan, S. & Shen, H. (2024). Minimize Resource Cost for Containerized Microservices Under SLO via ML-Enhanced Layered Queueing Network Optimization. *2024 14th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, 631–637.

Lv, W., Wang, Q., Yang, P., Ding, Y., Yi, B., Wang, Z. & Lin, C. (2022). Microservice Deployment in Edge Computing Based on Deep Q Learning. *IEEE Transactions on Parallel and Distributed Systems*, 33, 2968–2978.

Lv, W., Yang, P., Zheng, T., Lin, C., Wang, Z., Deng, M. & Wang, Q. (2024). Graph-Reinforcement-Learning-Based Dependency-Aware Microservice Deployment in Edge Computing. *IEEE Internet of Things Journal*, 11, 1604-1615.

Mahajan, V. B. & Mane, S. B. (2022). Detection, analysis and countermeasures for container based misconfiguration using docker and kubernetes. *2022 International Conference on Computing, Communication, Security and Intelligent Systems (IC3SIS)*, pp. 1–6.

Mathai, A., Bandyopadhyay, S., Desai, U. & Tamilselvam, S. (2022). Monolith to Microservices: Representing Application Software through Heterogeneous Graph Neural Network. *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)*, 3905–3911.

Morais, G., Bork, D. & Adda, M. (2021). Towards an Ontology-driven Approach to Model and Analyze Microservices Architectures. *Proceedings of the 13th International Conference on Management of Digital EcoSystems (MEDES)*, 79–86.

Mparmpoutis, A. & Kakarontzas, G. (2022). Using Database Schemas of Legacy Applications for Microservices Identification: A Mapping Study. *Proceedings of the 6th International Conference on Algorithms, Computing and Systems*, pp. 1–7.

Nadareishvili, I., Mitra, R., McLarty, M. & Amundsen, M. (2016). *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.".

Nakazawa, R., Ueda, T., Enoki, M. & Horii, H. (2018). Visualization Tool for Designing Microservices with the Monolith-First Approach. *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, 32–42.

Newman, S. (2015a). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc.

Newman, S. (2015b). *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".

Nitin, V., Asthana, S., Ray, B. & Krishna, R. (2022). CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022*, 1–12.

Nunes, L., Santos, N. & Rito Silva, A. (2019). From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts. *Lecture Notes in Computer Science*, 11377, 37–52.

Page, M. J., McKenzie, J. E., Bossuyt, P. M., Boutron, I., Hoffmann, T. C., Mulrow, C. D., Shamseer, L., Tetzlaff, J. M., Akl, E. A., Brennan, S. E. et al. (2021a). The PRISMA 2020 Statement: An Updated Guideline For Reporting Systematic Reviews. *International journal of surgery*, 88, 105906.

Page, M. J., McKenzie, J. E., Bossuyt, P. M., Boutron, I., Hoffmann, T. C., Mulrow, C. D., Shamseer, L., Tetzlaff, J. M., Akl, E. A., Brennan, S. E. et al. (2021b). The PRISMA 2020 Statement: An Updated Guideline For Reporting Systematic Reviews. *International journal of surgery*, 88, 105906.

Page, M. J., Moher, D., Bossuyt, P. M., Boutron, I., Hoffmann, T. C., Mulrow, C. D., Shamseer, L., Tetzlaff, J. M., Akl, E. A., Brennan, S. E. et al. (2021c). PRISMA 2020 Explanation And Elaboration: Updated Guidance And Exemplars For Reporting Systematic Reviews. *bmj*, 372, 1–6.

Peng, S., Kalliamvakou, E., Cihon, P. & Demirer, M. (2023). The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*, 1–19.

Peterson, L. E. (2009). K-nearest neighbor. *Scholarpedia*, 4(2), 1883.

Ponce, F., Márquez, G. & Astudillo, H. (2019). Migrating from monolithic architecture to microservices: A Rapid Review. *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pp. 1–7.

Project, T. H. (2025). Helm - The Kubernetes Package Manager. Retrieved from: https://helm.sh.

Qian, L., Li, J., He, X., Gu, R., Shao, J. & Lu, Y. (2023). Microservice extraction using graph deep clustering based on dual view fusion. *Information and Software Technology*, 158, 1–11.

Rathod, T., Joseph, C. & Martin, J. (2023). Improving Industry 4.0 Readiness: Monolith Application Refactoring using Graph Attention Networks. *Proceedings - 23rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing Workshops, CCGridW 2023*, 223–230.

Ray, K., Banerjee, A. & Narendra, N. (2023). Learning-Based Microservice Placement and Migration for Multi-Access Edge Computing. *IEEE Transactions on Network and Service Management*, 17, 1–10.

Razzaq, A. & Ghayyur, S. A. (2023). A systematic mapping study: The new age of software architecture from monolithic to microservice architecture—awareness and challenges. *Computer Applications in Engineering Education*, 31(2), 421–451.

Rish, I. et al. (2001). An empirical study of the naive Bayes classifier. *IJCAI 2001 workshop on empirical methods in artificial intelligence*, 3, 41–46.

Robertson, S. E., Walker, S., Beaulieu, M., Gatford, M. & Payne, A. (1996). Okapi at TREC-4. *Nist Special Publication Sp*, 73–96.

Romani, Y., Tibermacine, O. & Tibermacine, C. (2022). Towards Migrating Legacy Software Systems to Microservice-based Architectures: a Data-Centric Process for Microservice Identification. *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, 15–19.

Rosa, G., Mastropaolo, A., Scalabrino, S., Bavota, G. & Oliveto, R. (2023). Automatically Generating Dockerfiles via Deep Learning: Challenges and Promises. *2023 IEEE/ACM International Conference on Software and System Processes (ICSSP)*, pp. 1–12.

Saidani, I., Ouni, A., Mkaouer, M. W. & Saied, M. A. (2019). Towards automated microservices extraction using muti-objective evolutionary search. *International Conference on Service-Oriented Computing*, pp. 58–63.

Saidi, M., Tissaoui, A., Benslimane, D. & Faiz, S. (2022). Automatic Microservices Identification Across Structural Dependency. *Lecture Notes in Networks and Systems*, 386–395.

Saidi, M., Tissaoui, A. & Faiz, S. (2023). A DDD Approach Towards Automatic Migration To Microservices. *2023 IEEE International Conference on Advanced Systems and Emergent Technologies (IC_ASET)*, pp. 01-06.

Santos, W., Sampaio, A. J., Rosa, N. & Cavalcanti, G. (2024). Microservices performance forecast using dynamic Multiple Predictor Systems. *Engineering Applications of Artificial Intelligence*, 243, 107649.

Saucedo, A. M., Rodríguez, G., Rocha, F. G. & dos Santos, R. P. (2024). Migration of monolithic systems to microservices: A systematic mapping study. *Information and Software Technology*, 107590.

Saucedo, A. M., Rodríguez, G., Rocha, F. G. & dos Santos, R. P. (2025). Migration of monolithic systems to microservices: A systematic mapping study. *Information and Software Technology*, 177, 107590.

Sellami, K., Ouni, A., Saied, M. A., Bouktif, S. & Mkaouer, M. W. (2022a). Improving microservices extraction using evolutionary search. *Information and Software Technology*, 151, 106996.

Sellami, K., Saied, M. A. & Ouni, A. (2022b). A hierarchical dbscan method for extracting microservices from monolithic applications. *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, 201-210.

Selmadji, A., Seriai, A.-D., Bouziane, H. L., Oumarou Mahamane, R., Zaragoza, P. & Dony, C. (2020). From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach. *2020 IEEE International Conference on Software Architecture (ICSA)*, 157–168.

Shafi, N., Abdullah, M., Iqbal, W., Erradi, A. & Bukhari, F. (2024). Cdascaler: a cost-effective dynamic autoscaling approach for containerized microservices. *Cluster Computing - The Journal of Networks Software Tools and Applications*, 5195–5215.

Shahini, S. & Momeni, H. (2024). Autoencoder-based Anomaly Detection in Microservices using Distributed Tracing. *2024 20th CSI International Symposium on Artificial Intelligence and Signal Processing (AISP)*, 1-6.

Shi, K., Li, J., Liu, Y., Chang, Y. & Li, X. (2022). BSDG: Anomaly Detection of Microservice Trace Based on Dual Graph Convolutional Neural Network. *Service-oriented Computing (ICSOC 2022)*, 13740, 171–185.

Silva Filho, H. C. d. & Figueiredo Carneiro, G. d. (2019). Strategies reported in the literature to migrate to microservices based architecture. *16th International Conference on Information Technology-New Generations (ITNG 2019)*, pp. 575–580.

Song, C., Xu, M., Ye, K., Wu, H., Gill, S., Buyya, R. & Xu, C. (2023). ChainsFormer: A Chain Latency-Aware Resource Provisioning Approach for Microservices Cluster. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 14419, 197–211.

Song, Y., Xin, R., Chen, P., Zhang, R., Chen, J. & Zhao, Z. (2024). Autonomous selection of the fault classification models for diagnosing microservice applications. *Future Generation Computer Systems*, 153, 326–339.

Sooksatra, K., Maharjan, R. & Cerny, T. (2022). Monolith to Microservices: VAE-Based GNN Approach with Duplication Consideration. *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 1–10.

Stojanovic, T. & Lazarević, S. D. (2023). The application of ChatGPT for identification of microservices. *E-business technologies conference proceedings*, 3(1), 99–105.

Sun, X., Boranbaev, S., Han, S., Wang, H. & Yu, D. (2022). Expert system for automatic microservices identification using API similarity graph. *Journal of Software: Evolution and Process (JSEP 2022)*, e13158.

Sun, Y., Lin, Z., Shi, B., Zhang, S., Ma, S., Jin, P., Zhong, Z., Pan, L., Guo, Y. & Pei, D. (2024). Interpretable Failure Localization for Microservice Systems Based on Graph Autoencoder. *ACM Transactions on Software Engineering and Methodology*, 33, 326–339.

Swain, P. H. & Hauska, H. (1977). The decision tree classifier: Design and potential. *IEEE Transactions on Geoscience Electronics*, 15(3), 142–147.

Tan, R. & Li, Z. (2024). MAAD: A Distributed Anomaly Detection Architecture for Microservices Systems. *38th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 1009-1021.

Tong, G., Meng, C., Song, S., Pan, M. & Yu, Y. (2023). GMA: Graph Multi-agent Microservice Autoscaling Algorithm in Edge-Cloud Environment. *2023 IEEE International Conference on Web Services (ICWS)*, 393–404.

Toumi, N., Bagaa, M. & Ksentini, A. (2023). Machine learning for service migration: a survey. *IEEE Communications Surveys & Tutorials*, 25(3), 1991–2020.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F. et al. (2023). Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 1-27.

Trabelsi, I., Abdellatif, M., Abubaker, A., Moha, N., Mosser, S., Ebrahimi-Kahou, S. & Guéhéneuc, Y.-G. (2023). From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis. *Journal of Software: Evolution and Process*, 35, e2503.

Trabelsi, I., Moha, N., Gueheneuc, Y.-G. & Geffard, L. (2024a). Magnet: Method-Based Approach Using Graph Neural Network for Microservices Identification. *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, 1–11.

Trabelsi, I., Popa, B., Pereyrol, J., Beaulieu, P.-O. & Moha, N. (2024b). MicroMatic: Fully Automated Microservices Identification Approach From Monolithic Systems. *Proceedings of the ACM/IEEE 6th International Workshop on Software Engineering Research & Practices for the Internet of Things*, 7–13.

Vaithilingam, P., Zhang, T. & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. *Chi conference on human factors in computing systems extended abstracts*, pp. 1–7.

Varshneya, K. (2022). Design of Microservices Architecture at Netflix. Retrieved from: https://www.techaheadcorp.com/blog/design-of-microservices-architecture-at-netflix/.

Varshneya, K. (2023). Decoding Software Architecture of Spotify: How Microservices Empowers Spotify. Retrieved from: https://www.techaheadcorp.com/blog/decoding-software-architecture-of-spotify-how-microservices-empowers-spotify/.

Vayghan, L. A., Saied, M. A., Toeroe, M. & Khendek, F. (2018). Deploying microservice based applications with kubernetes: Experiments and lessons learned. *2018 IEEE 11th international conference on cloud computing (CLOUD)*, pp. 970–973.

Velepucha, V. & Flores, P. (2021). Monoliths to microservices-migration problems and challenges: A SMS. *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*, pp. 135–142.

Vera-Rivera, F., Puerto, E., Astudillo, H. & Gaona, C. (2021). Microservices Backlog-A Genetic Programming Technique for Identification and Evaluation of Microservices From User Stories. *IEEE Access*, 9, 117178–117203.

Vohra, D. (2016). *Kubernetes microservices with Docker*. Apress.

Vural, H., Koyuncu, M. & Guney, S. (2017). A systematic literature review on microservices. *Computational Science and Its Applications–ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part VI 17*, pp. 203–217.

Wang, J., Li, Y., Qi, Q., Lu, Y. & Wu, B. (2024). Multilayered Fault Detection and Localization With Transformer for Microservice Systems. *IEEE Transactions on Reliability*, 1502-1515.

Wang, Y., Wang, W., Joty, S. & Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 1–13. doi: 10.48550/arXiv.2109.00859.

Xiao, Z., Wijegunaratne, I. & Qiang, X. (2016). Reflections on SOA and Microservices. *2016 4th International Conference on Enterprise Systems (ES)*, pp. 60–67.

Xu, F. F., Alon, U., Neubig, G. & Hellendoorn, V. J. (2022). A systematic evaluation of large language models of code. *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, pp. 1–10.

Xu, Y., Qiu, Z., Gao, H., Zhao, X., Wang, L. & Li, R. (2023). Heterogeneous Data-Driven Failure Diagnosis for Microservice-Based Industrial Clouds Towards Consumer Digital Ecosystems. *IEEE Transactions on Consumer Electronics*, 1–1.

Yang, Z., Nguyen, P., Jin, H. & Nahrstedt, K. (2019). MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows. *Proceedings - International Conference on Distributed Computing Systems*, 122–132.

Zaragoza, P., Seriai, A.-D., Seriai, A., Shatnawi, A., Bouziane, H.-L. & Derras, M. (2021). Materializing microservice-oriented architecture from monolithic object-oriented source code. *International Conference on Software Technologies*, pp. 143–168.

Zeng, H., Wang, T., Li, A., Wu, Y. & Zhang, W. (2023). Topology-Aware Self-Adaptive Resource Provisioning for Microservices. *2023 IEEE International Conference on Web Services (ICWS)*, 28–35.

Zhang, C., Peng, X., Sha, C., Zhang, K., Fu, Z., Wu, X., Lin, Q. & Zhang, D. (2022a). DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 623–634.

Zhang, K., Zhang, C., Peng, X. & Sha, C. (2022b). PUTraceAD: Trace Anomaly Detection with Partial Labels based on GNN and PU Learning. *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, 239–250.

Zhang, Y., Meredith, R., Reeves, W., Coriolano, J., Babar, M. A. & Rahman, A. (2024). Does Generative AI Generate Smells Related to Container Orchestration?: An Exploratory Study with Kubernetes Manifests. *Proceedings of the 21st International Conference on Mining Software Repositories*, pp. 192–196.

Zhong, T., Teng, Y., Ma, S., Chen, J. & Yu, S. (2023). A Microservices Identification Method Based on Spectral Clustering for Industrial Legacy Systems. *2023 IEEE Globecom Workshops (GC Wkshps)*, 1331-1337.

Zhu, H. & Gehrmann, C. (2022). Kub-Sec, an automatic Kubernetes cluster AppArmor profile generation engine. *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pp. 129–137.