Performance-Aware and Data-Driven Evolution of Microservices Systems

Thakshila Imiya Mohottige

Submitted in total fulfilment of the requirements of the degree of
Doctor of Philosophy

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE, AUSTRALIA

May 2026

ORCID: 0009-0002-7422-2303

Copyright © 2026 Thakshila Imiya Mohottige
All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the author.

Performance-Aware and Data-Driven Evolution of Microservices Systems

Thakshila Imiya Mohottige
Principal Supervisor: Prof. Artem Polyvyanyy
Co-Supervisors: Prof. Rajkumar Buyya, Prof. Colin Fidge, Prof. Alistair Barros

---

Abstract

Software system architecture refers to the high-level design, structure, and interactions. It is a blueprint for the entire system that helps developers and other stakeholders to develop maintainable, reliable, scalable, and robust solutions. Legacy software systems often follow a monolithic architecture, in which the systems develop, test, and deploy as a single coherent unit. However, with the growing demand for software systems, legacy systems frequently encounter scalability constraints. Modern software systems use the microservices architecture, in which systems are developed as a combination of scalable, flexible, independent, decentralized, and loosely coupled services. Entirely redeveloping legacy systems to adopt the new microservices paradigm is not feasible due to the costs, effort, and risk of introducing errors. Hence, incremental migration to microservices is a pragmatic approach for modernizing legacy software systems.

Migrating legacy systems to microservices is an inherently complex task. Interdependent business logic, data access, and system components make it difficult to identify service boundaries. The dominant approach to addressing this challenge is static analysis, which examines computer program source code to identify cohesive clusters of classes and methods. However, static analysis has the fundamental limitation of incorporating source code segments that rarely or never execute in practice. In large legacy source code, significant portions may not be in current use. In contrast, dynamic analysis, which examines runtime traces from execution logs, captures insights into the actual operational behavior of the system. Dynamic analysis remains underinvestigated in the context of microservices migration, particularly by using pattern mining techniques. This gap motivates

investigating the impact of historical execution traces on microservice identification using sequential pattern mining techniques.

An architectural transformation should not degrade the system performance. Therefore, this thesis evaluates the performance impact of migrating microservices by measuring the factors that influence historical execution-based microservices identification and the resulting performance implications. This evaluation provides empirical evidence that microservice migration can achieve performance improvements while maintaining structural soundness.

Beyond migration, this thesis advances the architecture of microservices systems by proposing and evaluating systems of microservices with autonomous monitoring, analysis, and decision-making capabilities, referred to as service colonies, that adapt at runtime to improve performance and optimize resource utilization. Runtime execution data is the foundational input that facilitates these adaptations, as it provides real-time information on systems' performance.

Finally, the critical enabler for these contributions is execution data. However, the lack of standardization for execution data limits interoperability and increases the development effort and complexity of runtime-based microservices detection and self-adaptation. Hence, this thesis concludes with a unified data model for execution data that enhances its utility for both microservices migration and self-adaptive pipelines.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,
2. due acknowledgment has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies, and appendices.
4. the use of Claude AI and Grammarly for rephrasing, summarizing, and language improvements.

Thakshila Imiya Mohottige, May 2026

Preface

This thesis is submitted in total fulfillment of the requirements for the degree of Doctor of Philosophy at the University of Melbourne. The research presented in this thesis was conducted in the School of Computing and Information Systems, The University of Melbourne, under the supervision of Professor Artem Polyvyanyy and Professor Rajkumar Buyya. External supervision was provided by Professor Colin Fidge and Professor Alistair Barros from Queensland University of Technology. The work reported in this thesis is original, and I was the principal author of all papers, contributing more than 50% to each paper. My responsibilities included designing architectures, implementing and integrating respective tools, creating the deployment infrastructure, collecting datasets, conducting experiments, and analyzing the experimental results. My co-authors contributed by providing feedback on proposed frameworks and models and assisting with manuscript revisions.

Part of the content of Chapter 3 has been published in the following paper:

- **Thakshila Imiya Mohottige**, Artem Polyvyanyy, Colin Fidge, Rajkumar Buyya, Alistair Barros, Reengineering software systems into microservices: State-of-the-art and future directions, *Information and Software Technology*, Volume 183, 2025, 107732, ISSN 0950-5849.

Part of the content of Chapter 4 has been accepted for publication in the following paper:

- **Thakshila Imiya Mohottige**, Artem Polyvyanyy, Colin Fidge, Rajkumar Buyya, Alistair Barros, MIST: Microservices Identification from Sequential Traces, *IEEE International Conference On Web Services (ICWS 2026)*, July 13–18, 2026. *(in press)*

Part of the content of Chapter 5 has been published or submitted for publication in the following papers:

- **Thakshila Imiya Mohottige**, Artem Polyvyanyy, Colin Fidge, Rajkumar Buyya, Alistair Barros, Service Colonies: A Novel Architectural Style for Developing Software Systems with Autonomous and Cooperative Services, *2030 Software Engineering Workshop, ACM International Conference on the Foundations of Software Engineering (FSE 2024), Porto de Galinhas, Brazil, July 15–19, 2024.*

- **Thakshila Imiya Mohottige**, Artem Polyvyanyy, Colin Fidge, Rajkumar Buyya, Alistair Barros, Service Colonies: A Novel Architectural Style for Developing Software Systems with Autonomous and Cooperative Services, *ACM TOSEM Frontiers of Software Engineering, April 2026 (second review round).*

Part of the content of Chapter 6 has been submitted for publication in the following papers.

- **Thakshila Imiya Mohottige**, Artem Polyvyanyy, Colin Fidge, Rajkumar Buyya, Alistair Barros, A Unified Model of Software Execution Event Data, *Journal of Systems and Software (JSS), May 2026 (under review).*

Acknowledgments

I express my sincere gratitude to my esteemed supervisors, Professor Artem Polyvyanyy and Professor Rajkumar Buyya from The University of Melbourne, together with my external supervisors, Professor Colin Fidge and Professor Alistair Barros from Queensland University of Technology. The journey through my PhD candidature has been both academically and personally challenging. However, it was through their continuous support, patience, encouragement, and invaluable guidance that I was able to complete this research. I acknowledge the University of Melbourne for providing me with the scholarship and resources. My research is in part supported by the Australian Research Council project DP220101516.

I am deeply grateful for the support, discussions, and inspiration received from my past and present colleagues in the Process Science and Technology group, including Anandi Karunarathna, Deanna Coralage, Wenjun Zhou, Tian Li, Qingtan Shen, Andrei Tour, Victoria Sonnemans, Goran Faisal, Dr. Zahra Dasht Bozorgi, and Dr. Zihang Zu.

I also extend my heartfelt thanks to my friends Tanya, Prasanna, Croos, Jasmine, and Ravindika, especially for helping my family and me settle into life in Melbourne. In addition, I am grateful to Ruvini, Madhushika, and Supun for their support in many ways throughout this candidature.

I would like to express my deepest gratitude to my entire family, my parents, my brother and his family, extended family members, my husband, and my daughter, for their unwavering love, dedication, encouragement, and support throughout every stage of my life. I would especially like to thank my father, mother, and brother, who believed in me and supported my education despite the many challenges they faced in their lives. I am deeply grateful to my husband, Roshanta, for his continuous support and understanding. I am especially grateful to my beloved daughter, Lyanna, whose presence has been a constant source of strength, happiness, and inspiration during this journey. Their sacrifices, patience, and unconditional support made the completion of the thesis possible.

Finally, I would like to express my sincere appreciation for the free education system in Sri Lanka, which provided me with the invaluable opportunity to learn and pursue higher education. Without this foundation, this journey would not have been possible.

Contents

1 Introduction **1**

1.1 Gaps and Motivations . . . 3

1.1.1 Microservices Identification . . . 3

1.1.2 Self-Adaptive Microservices . . . 4

1.1.3 Software Execution Data . . . 5

1.2 Research Questions and Contributions . . . 6

1.3 Thesis Organization . . . 8

2 Background **10**

2.1 Evolution of Software System Architecture . . . 11

2.1.1 Monolithic Architecture . . . 11

2.1.2 Service-Oriented Archiecture . . . 12

2.1.3 Microservices Architecture . . . 13

2.2 Legacy Systems to Microservices Migration . . . 14

2.2.1 Migration Challenges and Approaches . . . 14

2.2.2 Structural Quality Measures of Migrated Microservices . . . 15

2.3 Dynamic Software Analysis and Instrumentation . . . 17

2.3.1 Dynamic Software Analysis . . . 18

2.3.2 System Instrumentation . . . 18

2.3.3 The Kieker Monitoring Framework . . . 19

2.4 Software Execution Data . . . 20

2.4.1 Software Logs, Event Logs, XES Standard . . . 20

2.4.2 Software Logs to Event Conversion . . . 21

2.4.3 Events, Traces, and Pattern Properties . . . 21

2.4.4 Sequential Pattern Mining 23
2.4.5 Process Mining 24
2.5 Self-adaptive Software Systems 25
2.5.1 Autonomic Computing 25
2.5.2 The MAPE-K Control Loop 26
2.5.3 Self-Adaptive Microservices and Multi-Agent Systems 26
2.5.4 Runtime Performance Metrics For Self-Adaptive Systems 27
2.6 Summary 29

3 Related Work 31
3.1 Reengineering Legacy Systems To Microservices 32
3.1.1 Systematic Literature Review Process 33
3.1.2 Results: Monolithic to Microservices Migration 35
3.1.3 Summary of Study Gaps and Further Requirements 50
3.2 Self-Adaptive Software Systems 50
3.2.1 Review of Existing Self-Adaptive Service-based Systems 51
3.2.2 Results: Existing Self-Adaptive Service-Based Systems 52
3.2.3 State-of-the-Art Self-Adaptive Approaches 53
3.2.4 Summary of Study Gaps and Further Requirements 56
3.3 Formalization of Software Execution Event Data 56
3.3.1 Existing Execution Data Models and Standards 57
3.3.2 Review on Software Log Mining Applications 58
3.3.3 Results: Problems Domains Addressed Using Software Logs . . . 59
3.3.4 Summary of Study Gaps and Further Requirements 64
3.4 Summary 65

4 Performance-aware Microservice Migration Using Execution Data 67
4.1 Microservice Migration Using Sequential Traces 68
4.2 MIST Framework Overview 69
4.2.1 Execution Trace Analysis and Pattern Mining 69
4.2.2 Sequential Pattern Mining 73
4.2.3 Pattern Cost Calculation and Postprocessing 74
4.2.4 Microservice Selection and Code Generation 77

4.3 Evaluation of Microservices . . . 79
4.3.1 Performance Impact of Sequential Patterns on Microservices . . . 81
4.3.2 Structural Quality Evaluation . . . 85
4.4 Implementation and Reproducibility . . . 87
4.4.1 Instrumentation and Input Configuration . . . 87
4.4.2 Plugable Pattern Mining Algorithms . . . 87
4.4.3 Pattern Mining Output Files . . . 88
4.4.4 Code Rewriting Component . . . 88
4.4.5 Reproducibility . . . 89
4.5 Summary . . . 89

5 Runtime Data Driven Adaptation for Microservice Systems 91
5.1 Self-Adaptive Software Systems . . . 92
5.2 Service Colonies . . . 95
5.2.1 Components and Interactions . . . 95
5.2.2 Inhabitants . . . 97
5.2.3 Adaptations . . . 99
5.3 Proof of Concept and Evaluation . . . 102
5.3.1 Self-Adaptation Description Language for Service Colony . . . 103
5.3.2 Experimental Results . . . 107
5.4 Benefits and Challenges . . . 120
5.4.1 Benefits . . . 120
5.4.2 Challenges . . . 123
5.5 Implementation and Reproducibility . . . 125
5.5.1 System Components . . . 125
5.5.2 Reproducibility . . . 126
5.6 Summary . . . 126

6 A Unified Model of Software Execution Event Data 128
6.1 Software to Event Logs for Execution Data Mining . . . 129
6.2 Data Extraction and Analysis Methodology . . . 131
6.3 Results: From Software Logs to SEED Model . . . 132
6.3.1 Software log Properties Used For Mining Applications (DMRQ2) 132

6.3.2 From Log Properties Relationships to SEED Model (DMRQ3) . . 134
6.4 SEED Model Compatibility & Feasiblity 137
6.4.1 Dimensional Modeling 137
6.4.2 Compatibility with XES standards 138
6.4.3 Feasibility and Adoption Pathway 140
6.5 Compliance Rules and SEED Expressiveness 141
6.6 Implementation and Reproducibility 146
6.6.1 XES Converter 146
6.6.2 Reproducibility 146
6.7 Summary 147

7 Conclusions and Future Directions 149
7.1 Summary of Contributions 149
7.2 Future Research Directions 152
7.2.1 Performance-Aware Microservice Identification 152
7.2.2 AI-Assisted Microservice Code Generation 152
7.2.3 Behavioral Equivalence of Microservice Systems 153
7.2.4 Service Colonies 154
7.2.5 Adaptive and Privacy-Aware Logging for SEED 154
7.2.6 End-to-End Legacy Systems Modernization Pipeline 155
7.3 Final Remarks 156

A Literature Review Study List 184

List of Figures

1.1 Evolution of software architecture. 2
1.2 Summarized view of contributions. 6
1.3 Thesis organization. 9

2.1 Example monolithic architecture from the e-commerce domain. 12
2.2 Microservices architecture. 14
2.3 Example FP-tree for transaction database. 24
2.4 A typical self-adaptive system architecture. 25
2.5 MAPE-K control loop for architecture 27

3.1 Literature review study selection process. 35
3.2 Number of studies changes over time. 36
3.3 Classification of migration approaches. 38
3.4 Classification of legacy system modeling techniques. 42
3.5 Classification of microservice extraction techniques. 45
3.6 Classification of evaluation techniques. 47

4.1 Overview of the *MIST* framework. 70
4.2 JForum throughput analysis. 84
4.3 Apache Roller throughput analysis. 84
4.4 JForum latency analysis. 85
4.5 Apache Roller latency analysis. 85

5.1 Service colony adaptation process. 97
5.2 An example architecture of a service colony inhabitant. 98
5.3 Agreement for merging Inhabitants 4 and 5. 102

5.4 Agreement for splitting between Inhabitants 3, 5′, and 6. 102
5.5 Two service colony deployments. 104
5.6 ACML adaptation view. 105
5.7 Example ACML adaptation cases for service colonies. 106
5.8 Scenario 1: Response time analysis (splitting). 109
5.9 Scenario 1: Memory consumption analysis (splitting). 110
5.10 Scenario 2: Response time analysis (merging). 113
5.11 Scenario 2: Memory consumption analysis (merging). 114
5.12 Scenario 3: Response time analysis (splitting and merging). 115
5.13 Scenario 3: Inquiry queue size fluctuation. 116
5.14 Scenario 3: Memory consumption analysis (splitting and merging). 117
5.15 Scenario 4: Response time analysis (limited resources). 119
5.16 Scenario 4: Memory consumption (limited resources). 121

6.1 High-level usage of log properties in application areas. 134
6.2 SEED model. 136
6.3 SEED model mapping to XES standard. 139
6.4 Sample program to apply SEED model. 140
6.5 Sample XES output for the program. 140

List of Tables

3.1 Comparative review of reengineering software systems. 33
3.2 SLR paper classification details. 35
3.3 Tools for monolithic system analysis. 41
3.4 Additional tools used for system analysis. 42
3.5 Tools and levels of automation. 43
3.6 Evaluated applications. 46
3.7 Cross-system evaluated frameworks. 48
3.8 A comparison of existing self-adaptive software systems. 54

4.1 Sample output of pattern and cost. 77
4.2 Summary of evaluated applications. 81
4.3 Structural property evaluation. 86

5.1 Adaptation scenario evaluation results. 112

6.1 Software log properties used for mining. 133
6.2 SEED model nodes and attributes. 135
6.3 Compliance specifications 142

List of Abbreviations

AOP	Aspect-Oriented Programming
ART	Average Response Time
AST	Abstract Syntax Tree
CHD	Cohesion at Domain Level
CHM	Cohesion at Message Level
CRM	Customer Relationship Management
DDD	Domain-Driven Design
ER	Entity-Relationship
ESB	Enterprise Service Bus
FP	Frequent Pattern
IFN	Interface Number
IRR	Incoming Request Rate
JVM	Java Virtual Machine
LoC	Lines of Code
MAS	Multi-Agent System
PAIS	Process-aware Information System
QoS	Quality of Service
QS	Queue Size
RT	Response Time
SM	Structural Modularity
SOAP	Simple Object Access Protocol
UML	Unified Modeling Language
WSDL	Web Service Description Language
XES	eXtensible Event Stream

Chapter 1

Introduction

The architecture of a software system is the fundamental driving factor for a successful software product. It defines the high-level components and interactions of the software system. It is the primary enabler for non-functional attributes such as scalability, maintainability, deployability, and evolvability [1]. The impact of these quality attributes is further enhanced by the underlying infrastructure, such as cloud-based technologies. As illustrated in Fig. 1.1, the software architecture has evolved from monolithic architecture to (micro)service-based systems to facilitate these technical advances. Microservice architecture has emerged as the dominant architectural style in practice, enabling systems to be developed as collections of independent, loosely coupled, highly cohesive, and modular services that communicate via well-defined and lightweight protocols [2, 3]. Despite these technical advances, legacy monolithic software systems, built and deployed as a single coherent unit, remain in use due to their embedded critical business operations, substantial investments, and the risks and costs of rewriting [4]. The complete redevelopment of legacy monolithic systems is not feasible due to the costs and risks involved. Instead, the practical approach to modernizing monolithic systems is an incremental transition that identifies candidate microservices from the legacy system and progressively migrates operations to a microservices architecture by retiring the corresponding components from the monolithic system. However, identifying microservices within a monolithic system poses a high complexity and risk due to tightly coupled interdependencies in legacy source code, where business logic, data access, and system components are interconnected [4].

Beyond migration, maintaining the quality of microservice-based systems in dynamic

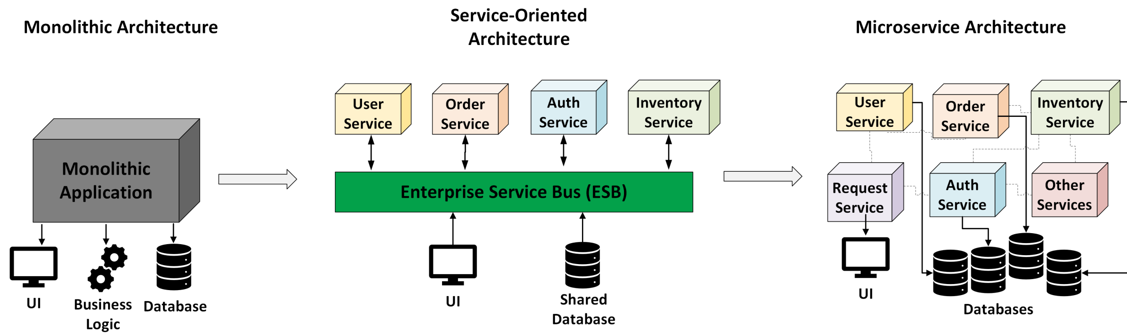


Figure 1.1: Evolution of software architecture.

operational environments presents additional challenges, as monitoring, fault detection, and optimization remain largely manual and costly activities [5]. The microservices architecture needs to be further enhanced to facilitate manual monitoring and maintenance in software systems by providing autonomous capabilities that self-adapt and optimize in response to uncertain operational conditions.

Runtime execution data captures method invocations, component interactions, state transitions, and operational behavior of a running system and provides an accurate representation of software execution [6, 7]. In contrast, static source code analysis reflects only the structural aspects of the code and can include deprecated or unused code segments, which are especially common in legacy systems [8, 9]. Both microservice migration and self-adaptation can leverage runtime execution data to gain insights into system behavior. Execution data is a broad term that encompasses all runtime-generated records, including execution traces, performance metrics, event logs, and state transitions. For microservice migration, execution traces, the ordered sequences of method invocations recorded during individual system executions, are the primary form of execution data used. Traces collected over time under various workloads and usage patterns can be used to accurately identify functional dependencies and interaction patterns, thereby deriving service boundaries from legacy software systems. Moreover, autonomous self-adaptive systems can use runtime data to identify precise system behavior, which can be used to detect anomalies, assess performance degradation, and make adaptive decisions [7, 5].

Microservices introduce network communication overhead, data serialization costs, and distributed coordination complexity absent in monolithic deployments [10]. These factors directly affect system latency, throughput, and response time. Existing studies

on microservice migration predominantly evaluate structural quality measures such as coupling, cohesion, and modularity. The performance implications of the identified microservices remain largely uninvestigated [11]. Similarly, self-adaptive microservice-based systems that autonomously re-architect the system structure must ensure that adaptation decisions improve or maintain system performance rather than introducing additional overhead [5, 12]. Hence, performance evaluation is essential for both microservice migration and self-adaptation without compromising system stability.

Software execution event data enables both the identification and self-adaptation of microservices. However, the lack of standardization of such data limits its utility across tools and domains. Execution data are captured in a variety of formats, including structured and unstructured logs, events, and traces, but there is no standardized conceptual model that governs their structure, relationships, or interoperability across tools and domains [13]. The lack of standardization limits the reusability of execution data, increases development overhead, and limits its application in microservice identification, self-adaptive decision-making, and performance analysis [7].

This thesis addresses three interconnected challenges: performance-aware microservices identification from historical execution traces, self-adaptation of microservices using runtime data, and the standardization of software execution event data.

The subsequent sections of this chapter are organized as follows: gaps and motivations(Section 1.1), research questions and contributions(Section 1.2) and outline of the thesis(Section 1.3)

1.1 Gaps and Motivations

This section discusses the gaps in microservices identification, self-adaptive microservices, and software execution data to motivate the contributions of this thesis.

1.1.1 Microservices Identification

Microservice migration is an active area with a wide range of techniques, including static source code analysis [14, 15, 16, 17], dynamic runtime data analysis [9, 6, 18, 19, 20], domain-driven design [21, 22], and semantic-based topic modeling [23, 24, 25]. The identification of microservices from legacy monolithic systems has been primarily addressed

through static source code analysis, which examines class and method dependencies to identify candidate microservices [16, 15, 14]. Legacy codebases frequently contain deprecated and unreachable code segments [8], leading to inaccurate microservice boundary identification using static source code analysis [6, 18]. Dynamic analysis, which derives service boundaries from runtime execution data, provides precise insights into actual system behavior [9]. However, coverage of execution data is a critical concern, as software systems exhibit diverse operating conditions, seasonal usage patterns, and varying workloads [26]. Therefore, migrating to microservices using execution data should not rely on real-time data; instead, historical execution data must be considered to capture complete operational behavior.

Microservices inherently introduce additional complexity to software systems by creating multiple services that communicate over the network [2]. Multiple services communicating over a network introduce additional latency during request processing, which can result in performance regressions in throughput, latency, and response time. Although performance evaluation is critical, existing studies on microservice migrations primarily focus on structural quality metrics such as coupling, cohesion, and modularity [27].

Existing microservice identification approaches do not combine dynamic execution-data-driven analysis of historical execution traces with performance-based evaluation of the resulting microservices. Specifically, no existing approach evaluates the impact of identification parameters, such as sequential patterns in historical execution data, on system throughput and latency.

1.1.2 Self-Adaptive Microservices

Although the microservices architecture provides substantial advantages, it introduces additional complexities and challenges. Maintenance of microservices systems is predominantly a manual activity, requiring substantial ongoing investments in monitoring, fault detection, and system optimization [5, 28]. Modern microservice deployments operate in highly dynamic environments characterized by fluctuating workloads, network failures, service degradation, and changing operational demands [12]. Monitoring, troubleshooting, and maintenance in these environments are increasingly complex due to the large number of heterogeneous deployment environments [29]. The further evolution of the microservice architecture by advancing autonomous operations is essential to reduce maintenance costs

and improve system reliability [12].

Self-adaptive systems, which autonomously monitor runtime conditions and respond to environmental changes by self-optimization, reduce complexities in dynamic operating environments [30, 31]. Existing self-adaptive approaches for microservices operate predominantly on the infrastructure layer using CPU and memory utilization for adaptive decision making, and are based on reactive and rule-based decision-making [5]. The use of autonomous agents with decentralized and proactive decision-making driven by execution-level data remains an underexplored area in the self-adaptive microservices literature [32]. Furthermore, the performance implications of self-adaptation, for example, whether the autonomous reconfiguration decisions improve or degrade system response time and resource utilization, are rarely validated in existing work.

Existing self-adaptive microservice systems rely on centralized, reactive, rule-based decision-making driven by infrastructure-level metrics. Hence, decentralized, proactive, agent-based self-adaptation, driven by execution-level data and empirically validated for performance implications, remains an open problem in the field.

1.1.3 Software Execution Data

Runtime data from software execution are immensely helpful in multiple domains and use cases. It can be captured in multiple formats, such as structured and unstructured logs, events, and traces. Logging frameworks and system instrumentation support the generation of execution logs [13]. However, the lack of standardization significantly affects the utilization of software execution data, limiting interoperability, increasing development overhead, and reducing reusability [33]. These limitations highly impact microservices and self-adapting software systems, where execution data from multiple services are collectively analyzed to derive program insights. Therefore, standardized software execution data is essential to improve its usability.

Existing standards in execution data address narrow concerns. OpenTelemetry [34] provides a vendor-neutral specification for distributed tracing, while the eXtensible Event Stream (XES) standard [35] provides a rich semantic model for event logs, but was designed for business process mining. Existing standards do not capture software-specific constructs such as method invocations, class instances, call stack depth, and thread identifiers. The absence of a standardized conceptual model limits the interoperability of execution data

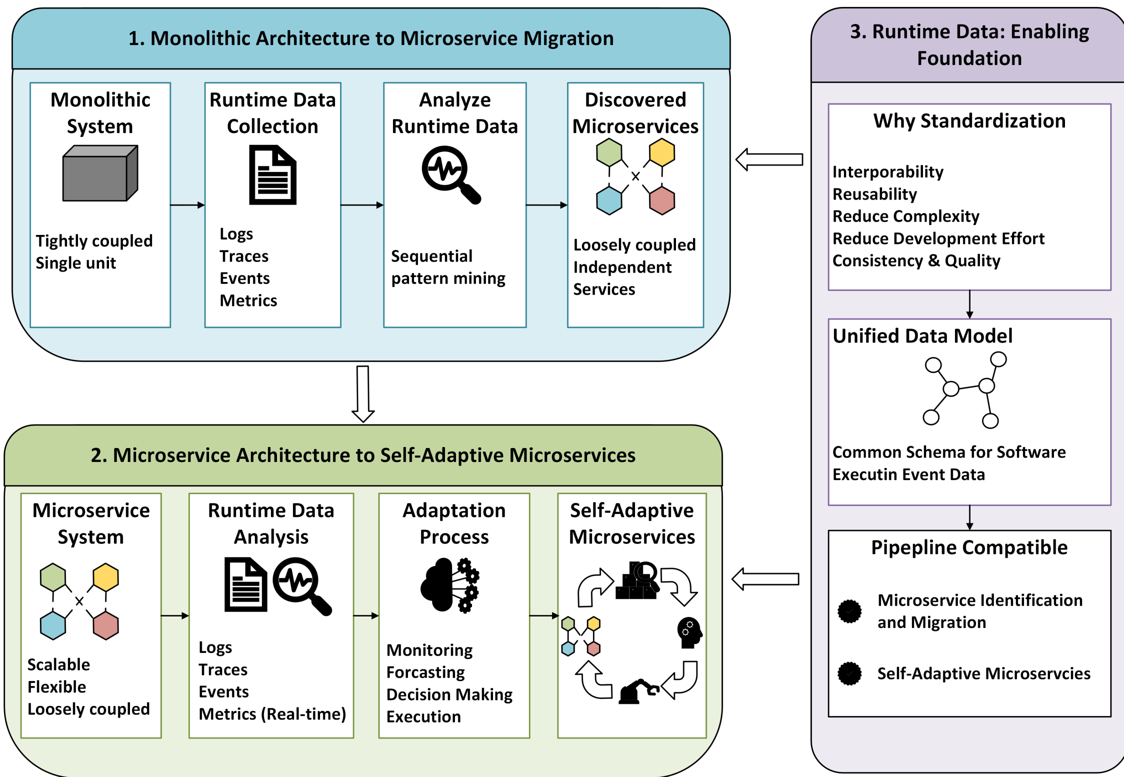


Figure 1.2: Summarized view of contributions.

across tools, increases development overhead when integrating data from heterogeneous sources, and reduces its reusability across different software domains.

Existing standards or models do not define a technology-independent conceptual data model that standardizes the representation of software execution events, traces, and interactions across the full range of entities involved in software execution. Hence, there is a gap in defining a conceptual data model for software execution data to increase log utilization and interoperability for microservice identification, self-adaptive decision-making, and other software system management activities.

1.2 Research Questions and Contributions

This section presents research questions and contributions grounded in the gaps identified in legacy monolithic systems to microservices migration, self-adaptive microservice systems, and standardization in software execution event data. A summary of the overall contribution

of the thesis illustrated in Figure 1.2, three interconnected areas with the common use of runtime execution data.

Research Question 1: How can historical executions of a software system be used to help migrate it to a microservice system that exhibits better performance?

The objective of RQ1 is to investigate how historical execution traces of legacy monolithic systems can be mined using sequential pattern mining techniques to identify microservice boundaries and empirically evaluate the structural quality and performance of microservices. This thesis makes the following contributions to RQ1:

- Design and implementation of a microservices identification framework grounded in sequential pattern mining techniques over historical executions of software systems.
- Evaluation of the quality of the identified microservices against existing structural quality measures, coupling, cohesion, and modularity.
- Empirical comparison of the influences of identification parameters, such as support, confidence, pattern length, call stack depth, and execution time, on the throughput and latency of the resulting microservices.

Research Question 2: How can a microservice system be effectively adapted in a dynamic environment based on its execution data?

The objective of RQ2 is to further enhance the microservices architecture to address the complexities in its operating environment. To achieve this objective, this thesis makes the following contributions:

- Design a self-adaptive microservices-based architecture, driven by decentralized, proactive decision-making, and utilizing runtime data.
- Evaluate the proposed approach for self-adaptive microservices to validate runtime performance, adaptation efficiency, and comparison of the results with the baseline system.

Research Question 3: How can the execution data of a software system be collected to maximize the utility of the data for the management of the system?

This research question is formulated to standardize runtime execution data to maximize the utility, interoperability, and reusability of execution data for software systems management activities, especially for microservices identification and self-adaptive systems. This thesis makes these contributions relevant to the research question:

- Design of a unified execution data model that standardizes the collection and representation of software runtime execution data across events, traces, and interaction with a focus on interoperability and the ability to integrate with existing standards.
- Validation of the proposed data model against existing rules for dynamic software analysis across different applications.

1.3 Thesis Organization

Figure 1.3 illustrates the organization of the thesis. While chapters one to three provide the introduction, background, and related work, chapters four to six contribute the content chapters that address answers to the research questions using execution data as a shared foundation. The thesis is structured into the following chapters:

Chapter 2 presents the foundational concepts required for three research contributions, covering microservice architecture, legacy system migration, dynamic software analysis and instrumentation, the Kieker monitoring framework, process mining and sequential pattern mining, self-adaptive systems, MAPE-K control loop, multi-agent systems, and data modeling concepts.

Chapter 3 reviews existing work on microservice reengineering approaches, techniques, and evaluation criteria, self-adaptive systems, and software execution data modeling, identifying the specific gaps that the three contributions of this thesis address.

Chapter 4 presents *MIST*, the performance-aware microservice identification framework that mines historical execution traces using sequential pattern mining and empirically evaluates the structural and performance characteristics of the identified microservices in terms of throughput and latency.

Chapter 5 presents Service Colonies, the self-adaptive microservices framework that models each microservice as an autonomous agent capable of decentralized, proactive decision-making driven by runtime execution data, and evaluates its operational performance based on response time and resource utilization.

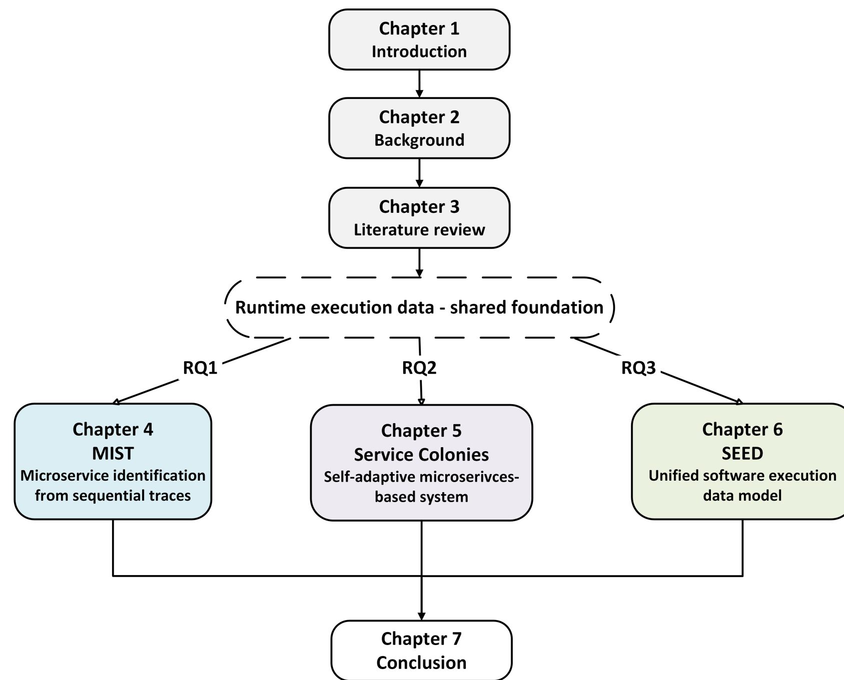


Figure 1.3: Thesis organization.

Chapter 6 presents SEED, the unified software execution event data model that standardizes the representation of runtime execution data across events, traces, and interactions, and validates the model against rules derived from existing studies that use software execution event data.

Chapter 7 summarizes the contributions and concludes the thesis by discussing the limitations and directions for future research.

The thesis proceeds by presenting the necessary background knowledge to establish a context and support the subsequent discussions. This fundamental chapter introduces key concepts, definitions, and measurements, ensuring a clear understanding of the domain. Detailed background discussion is provided in the next chapter.

Chapter 2

Background

The content of this chapter spans three interconnected areas: the automated identification of microservices from runtime execution data, the design of self-adaptive microservice-based systems, and standardizing software execution event data, providing the relevant foundations for the contributions of this thesis.

This chapter is organized as follows: Section 2.1 covers the evolution of software architecture from monolithic to microservices. Section 2.2 discusses the monolithic to microservices migration, associated challenges, and the evaluation of migrated microservices from the structural quality perspective. Section 2.3 presents dynamic software analysis, discussing its advantages, software system instrumentation, and a dynamic data-collection framework. Section 2.4 covers runtime execution data using software logs, event logs, and attributes of event logs. Moreover, this section discusses the areas of sequential pattern mining and process mining. Section 2.5 gives the background on self-adaptive software systems, including autonomic computing, the MAPE-K reference model, self-adaptive microservices, and performance metrics for self-adaptive systems.

2.1 Evolution of Software System Architecture

The field of software system architecture, design patterns, and technology is constantly evolving. Over the decades, technology has shifted from assembly and procedural paradigms to the object-oriented paradigm [36]. Object-oriented systems have evolved into component-based systems, alongside computing environments that have shifted from mainframes to distributed systems [37].

2.1.1 Monolithic Architecture

Initially, systems were developed using a monolithic architecture, as exemplified in Figure 2.1. In this architecture, the software system is built, compiled, packaged, deployed, and maintained as a single coherent unit. Within the source code, the user interfaces, business logic, and the data access layer are tightly coupled [38].

Monolithic applications run as a single process on the application server. Any change in the system triggers the rebuilding and redeployment of the entire application, replacing the previous deployment. The advantage of this architecture is simplicity. Monolithic applications are easy to implement, test, deploy, and maintain. In addition, a single database simplifies operations, eliminating the need for distributed coordination across multiple databases. All communications in the system are internal, primarily via internal method calls, which makes them fast. Consequently, monolithic architecture became the natural choice for building software systems [10].

As system users, code volume, functionality, and complexity grow, monolithic systems often face scalability, maintainability, and availability issues [2]. Modifying the tightly coupled system code becomes problematic, often leading to unexpected errors and behaviors in other functions. Monolithic systems also experience longer start-up times, increased development effort, and challenges with continuous deployment [10]. Moreover, with the growth of the application, the size of the development team increases, often leading to uneven utilization of the workforce and reduced productivity [38].

Scalability is a primary concern with monolithic systems because they lack fine-grained modules that can scale independently. Instead, the entire application must scale, often leading to resource overconsumption when only a subset of the system is under load. Furthermore, monolithic systems are technology-dependent, since all components share a

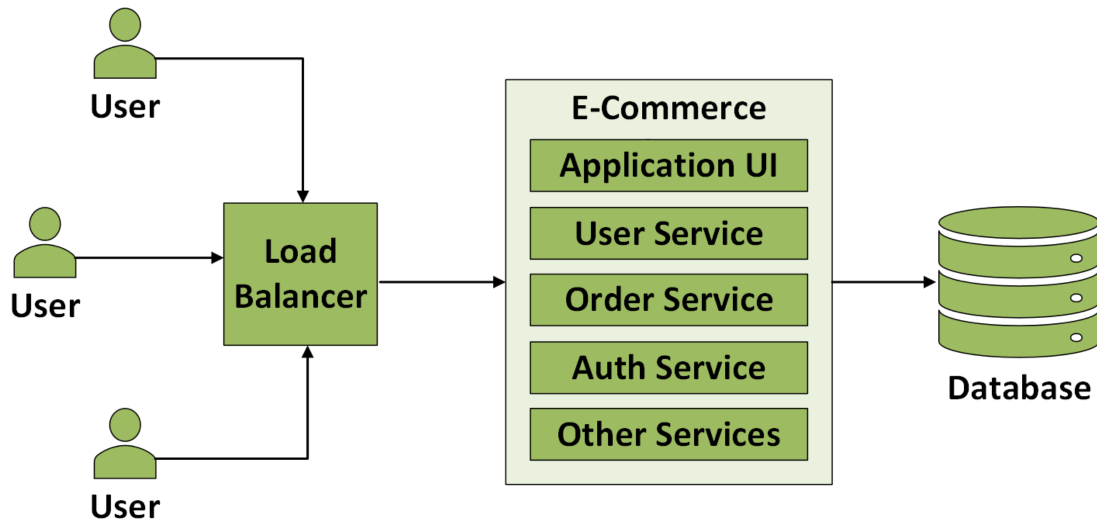


Figure 2.1: Example monolithic architecture from the e-commerce domain.

single runtime environment and programming language. The technology stack cannot be changed after the implementation. Moreover, a lack of fault tolerance and a single point of failure reduces system reliability, where an exception in one class or module can cause the entire system to crash.

2.1.2 Service-Oriented Archiecture

As the next phase in the evolution of software architecture, Service-Oriented Architecture (SOA) emerged to mitigate the issues inherent in monolithic architecture. SOA addressed structural rigidity and tight coupling in monolithic architecture by enabling flexible, reusable, and logically related services that communicate through standardized interfaces [39]. Unlike monolithic systems, these services can be independently developed, maintained, and reused, and support incremental modernization [40].

A distributed communication channel is required to implement a software system using SOA [41]. Typically, Enterprise Service Bus (ESB) is used with the Simple Object Access Protocol (SOAP) and Web Service Description Language (WSDL) [42, 43, 41]. This tightly coupled communication is the major constraint of SOA, especially when deploying a service, which requires redeploying dependent components and reintroducing communication coordination. Furthermore, reliance on centralized ESBs created bottlenecks.

The coarse granularity of the services limited independent scalability. These limitations led to the emergence of new architectural styles with finer-grained and independent services [44, 45]. Although legacy systems began migrating to SOA, their implementation was constrained by the complex inter-service communication.

2.1.3 Microservices Architecture

The microservices architecture has emerged to address the limitations of SOA. In contrast to the logically related operations in SOA, microservices are fine-grained, self-contained, and centered on functionality [3, 2]. Instead of the complex communication used in SOA, microservices communicate via well-defined lightweight protocols, mainly RESTful APIs [46].

Microservices are independent, allowing them to use different technical stacks based on their intended functionality and the requirements of individual microservices [3]. Microservices can be independently implemented, tested, deployed, and scaled. Based on the demand for individual microservices, resources can scale independently, enabling optimal resource utilization. This property accelerates the adoption of cloud-based technologies, continuous delivery, and DevOps practices in microservices, with decoupled release cycles between teams and reduced coordination overhead [47]. Furthermore, unlike in a monolithic architecture, failure of a single service does not cause cascading failures in a microservices system. The overall system continues to operate without the affected service. This provides high resilience and fault isolation [2]. To obtain the benefits of microservice architecture and remain competitive amid growing customer needs, companies began modernizing legacy monolithic systems into microservices. For example, major technology companies like Amazon, Netflix, Uber, eBay, and Spotify are early adopters of microservice architecture.

Despite the advantages of microservice architecture, it has its own associated complexities. Testing, monitoring, and maintenance of distributed microservice applications are more complex compared to monolithic systems. To test the overall functionalities, all required services must be up and running. Changes in communication protocols must be propagated to relevant microservices and migrated to production in a synchronous manner. Microservice-based systems communicate over the network. Hence, network communication overhead, data serialization, and distributed coordination complexities are

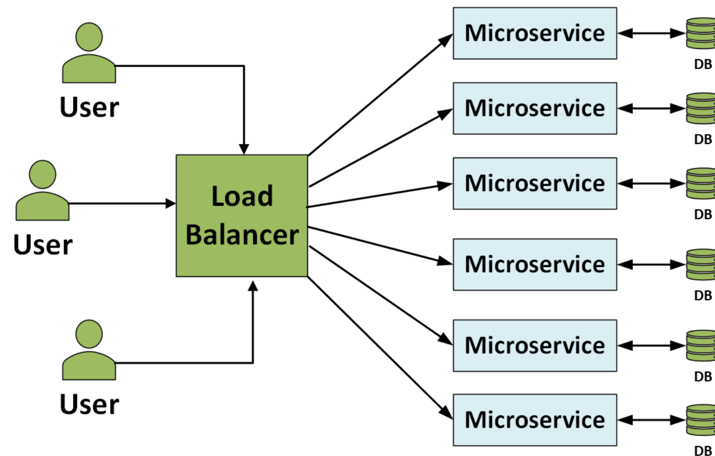


Figure 2.2: Microservices architecture.

introduced in microservice systems, which were not present in monolithic systems. These factors directly impact the performance of microservices systems [10].

2.2 Legacy Systems to Microservices Migration

This section discusses challenges associated with the migration of systems to microservices and structural quality measures of microservices.

2.2.1 Migration Challenges and Approaches

Most long-lived legacy applications in industry are developed using a monolithic architecture with obsolete technology [4]. These monolithic systems require significant resources and investment to maintain [4]. Despite their well-known drawbacks, legacy systems remain essential for enterprises because they support core business operations. Replacing such systems is difficult, as they reliably execute critical business functions [42]. The domain and operational knowledge in these systems has significant business value. Migrating monolithic systems to microservices is also challenging due to the lack of adequate documentation, a shortage of skilled employees, and limited resources [42]. Moreover, substantial investments already made in legacy monoliths make complete redevelopment infeasible.

Despite the growing adoption of microservice architecture, migrating legacy monolithic

systems remains an active and largely unsolved challenge in the industry [45]. A legacy monolithic system can be migrated to microservices architecture either by complete redevelopment or by incrementally extracting microservices [42]. Redeveloping a legacy system is problematic due to time-to-market pressures, redevelopment costs, limited visibility into the existing architecture, insufficient documentation, and resource constraints (e.g., human resources). Instead, incremental extraction of microservices is a promising approach, in which microservices are identified and introduced to the system incrementally rather than through a one-off migration [48]. This approach makes good use of the investment in the original system, which is often considerable and spans several decades. Moreover, only certain parts of the system may be suitable for migration, while others cannot benefit or may even degrade when moved to the new architectural style.

Multiple approaches have been used to identify microservices from legacy systems, including static source code analysis [14, 15, 16], dynamic runtime analysis [49, 6, 18], domain-driven design [50, 51], and hybrid techniques that combine these approaches. Static analysis uses source code structure and dependencies between classes/methods to identify connected clusters that can be extracted as microservices [14, 15, 16]. It is the most widely adopted approach [52]. However, legacy systems frequently contain deprecated, non-executing code segments that static analysis cannot distinguish from executing code, due to limitations in source code analysis [49, 6, 18]. This leads to inaccurate identification of service boundaries. Dynamic analysis, which derives service boundaries from actual runtime execution data, addresses this limitation.

2.2.2 Structural Quality Measures of Migrated Microservices

The quality of migrated microservices is primarily assessed based on their structural properties [27]. Four measures are often used: Structural Modularity (SM), Interface Number (IFN), Cohesion at Message Level (CHM), and Cohesion at Domain Level (CHD) [6, 51, 19, 53, 18].

1. **Structural Modularity** (SM) measures structural cohesiveness using the intra-connectivity within the microservices and inter-connectivity between the microservices [6, 18, 51, 19, 53]. Higher SM values contribute to increased cohesion within microservices.

$$SM = \frac{1}{N}\sum_{i=1}^{N}\frac{u_i}{N_i^2} - \frac{1}{N(N-1)/2}\sum_{i\neq j}^{N}\frac{\sigma_{i,j}}{2(N_i \times N_j)},$$

where N is the number of microservices, u_i is the number of connections and dependencies inside microservice i, $\sigma_{i,j}$ is the number of edges between the microservice i and j, and N_i and N_j are the numbers of entities (e.g., classes and functions) inside microservices i and j, respectively.

2. **Interface Number** (IFN) average the number of interfaces exposed by microservices [19, 6, 18, 53]. Lower values indicate better microservices due to the reduction of coupling among the microservices.

$$IFN = \frac{1}{N}\sum_{j=1}^{N}|I_j|,$$

where $|I_j|$ is the number of interfaces exposed by the microservice j and N is the number of microservices in the system.

3. **Cohesion at the message level** (CHM) defines the cohesiveness of the interfaces published by a microservice at the message level [6, 18, 53]. Higher CHM values indicate more cohesive services.

$$CHM = \frac{\sum_{j=1}^{\sum_{i=1}^{K} n_i} CHM_j}{\sum_{i=1}^{K} n_i}, \text{ with}$$

$$CHM_j = \begin{cases} \frac{\sum_{(k,m)} f_M(Op_k, Op_m)}{|I_i| \times (|I_i| - 1)/2} & \text{if } |I_i| \neq 1 \\ 1 & \text{if } |I_i| = 1 \end{cases}$$

$$f_M(Op_k, Op_m) = \frac{1}{2}(J(res_k + res_m) + J(pas_k + pas_m)),$$

where n_i is the number of interfaces provided by the microservice i, K is the number of microservices identified from the monolithic system, CHM_j is the cohesion of the interface I_i, Op_k and Op_m are operations provided by the interface I_i with $k \neq m$,

res_k is the set of response parameter types of operation Op_k, pas_k is the set of input parameter types of operation Op_K, J is the Jaccard coefficient, and f_M computes similarity of Op_k and Op_m operations.

4. **Cohesion at the domain level** (CHD) defines the cohesiveness of the interfaces provided by the domain level [6, 18, 53]. The higher the CHM value, the more cohesive the services.

$$CHD = \frac{\sum_{j=1}^{\sum_{i=1}^{K} n_i} CHD_j}{\sum_{i=1}^{K} n_i}, \text{ with}$$

$$CHD_j = \begin{cases} \dfrac{\sum_{(k,m)} f_D(Op_k, Op_m)}{|I_i| \times (|I_i| - 1)/2} & \text{if } |I_i| \neq 1 \\ 1 & \text{if } |I_i| = 1 \end{cases}$$

$$f_D(Op_k, Op_m) = J(T_{Op_k}, T_{Op_m}),$$

where n_i, K, Op_k, Op_m, I_i, and J are as defined for CHM. CHD_j measures the cohesion of the interface I_i, T_{Op_k} is the set of domain terms extracted from the signature of operation Op_k, T_{Op_m} is the domain terms extracted from the operation Op_m, and f_D computes the Jaccard similarity between the domain term sets of two operation.

2.3 Dynamic Software Analysis and Instrumentation

The traces of the software system executions provide the precise behavioral information rather than the static source code and its structure. This section introduces the dynamic software analysis, software instrumentation, the Kicker monitoring tool used in this thesis, the structure of execution traces and logs, and the challenges in distributed tracing. These background details are essential for understanding the collection of execution data for microservices identification, runtime self-adaptation, and establishing the requirements for data model execution data.

2.3.1 Dynamic Software Analysis

Runtime data of program executions is collected and analyzed by using dynamic software analysis. It provides an accurate picture of a software system by exposing its actual behavior [54]. Moreover, it provides precise execution information and depends on program input data [55]. In contrast to static analysis, which analyzes the static source code structure (classes, methods, call graphs, and dependencies), dynamic analysis relies on actual execution paths, components, and interaction sequences.

The primary advantage of dynamic analysis is the precision of the data produced [55]. Legacy systems typically contain inactive, deprecated, and unreachable code segments that static analysis cannot capture [6, 18]. Dynamic analysis addresses this issue by observing only the code segments that are actually executing. This makes dynamic analysis well-suited to microservice identification in legacy monolithic systems, due to the large volumes of legacy code, poor maintenance of system documentation, and lack of expert knowledge about the legacy system [56].

The primary drawback of dynamic analysis is that its accuracy relies on the coverage of the use cases. All system use cases based on user behavioral patterns and workload fluctuations need to be captured to achieve full coverage of system behavior [54]. This drawback motivates the development of a framework that can support historical execution traces, since real-time execution data analysis alone does not cover the complete behavior. Execution traces accumulated over time in different use cases and workload conditions are required to identify accurate boundaries in software systems.

2.3.2 System Instrumentation

A software log is the typical output of a software system. These logs often contain developer-inserted logs, explicitly defined in the source code. However, these logs are not comprehensive for a detailed analysis of execution behavior. Although manually inserting logs into the source code is an option, it is not feasible for a large codebase. Instrumentation is the practical approach that enables dynamic analysis by inserting a monitoring probe into the software system to record execution data without affecting system functionality [54]. Instrumentation determines the type, granularity, and format of the data collected. Hence, designing and configuring the instrumentation is vital for dynamic analysis.

Instrumentation can be applied at different levels: bytecode, aspect-oriented program-

ming (AOP) instrumentation, and agent-based instrumentation. Bytecode instrumentation injects probes into compiled bytecode during runtime class loading. AOP defines monitoring perspectives as aspects and integrates them into the system at compile or load time. Agent-based instrumentation uses platform instrumentation APIs to attach monitoring agents that intercept execution.

Method-level data collection is significant because it provides the finest-grained information on input and output data, timestamps, and call sequences for individual method invocations. Although functionality is not affected, instrumentation approaches introduce additional overhead based on granularity, volume, and the type of data collected from the software system

2.3.3 The Kieker Monitoring Framework

Kieker[1] is a Java-based open-source tool for monitoring and dynamic analysis [57]. It provides a comprehensive instrumentation infrastructure for collecting method-level runtime execution data, making it well-suited for dynamic-analysis-based studies. It has been widely used in existing microservice identification studies that employ dynamic analysis [9, 6, 18]. Kieker is used in this thesis as an instrumentation mechanism for dynamic analysis.

The Kieker architecture has three components. The *monitoring controller* acts as the central coordination component that activates and deactivates the monitoring probes, manages sessions, and routes information. *Probes* are the instrumentation points, defined using AOP via the Java Instrumentation API. *Monitoring writer* is responsible for collecting and writing log records to the file system.

Kieker can be integrated with a Java project via dependency configuration alone, without any changes to the source code. The log records captured by Kieker include a fully qualified method signature, session identifier, trace identifier, entry and exit timestamps, execution order, and call stack depth, which can be used for comprehensive analysis of the underlying software system. Kieker logs are written as text files, and each method invocation record has its own entry. An ordered sequence of method invocations can be constructed for each user request or trace using these log records. These execution traces share a common trace identifier. The resulting traces are ordered sequences of method

[1] `https://kieker-monitoring.net/`

invocations, annotated with timing information and call stack depth, that represent the complete execution path of a single request through the system. This trace structure is the fundamental unit of analysis for microservice identification in this thesis, representing the recurring sequential patterns of method executions in historical execution traces.

2.4 Software Execution Data

2.4.1 Software Logs, Event Logs, XES Standard

Software log records contain runtime execution details in a human-readable, structured format for operational purposes such as debugging and error reporting. Software logs vary significantly in format, verbosity, and content across different frameworks, programming languages, and deployment environments, making it difficult to analyze them consistently or at scale [13].

Event logs are typically used for advanced analysis of runtime data, providing a structured, standardized representation of runtime behavior. A sequence of events associated with a single case forms a trace, and a collection of traces formulates an event log [58, 59]. A case is a logical unit of execution, such as a request, transaction, or user session [58, 59]. Each event corresponds to a discrete occurrence, such as a method invocation, and includes a case ID, an activity descriptor, and a timestamp [59]. Additional attributes can be included in the event depending on the application domain.

This structure is essential for applying advanced analytical operations to software systems, such as sequential pattern mining, process mining, and specification mining. These operations require consistent, machine-interpretable records that can be grouped into traces, compared across executions, and mined for recurring patterns. Hence, converting software logs to event logs transforms unstructured operational data into a format that supports principled, repeatable analysis of system behavior.

The XES standard [35] defines a format for event logs and is considered the de facto standard for event data processing. Event logs generated under the XES standard are used as the primary input for established process mining tools such as ProM[2] and Disco[3] [58]. Hence, it is essential that event logs and execution data be represented in a

[2] `https://promtools.org/`
[3] `https://fluxicon.com/disco/`

format compatible with the XES standard.

2.4.2 Software Logs to Event Conversion

Process-aware Information Systems (PAISs), such as Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM) systems, are software systems that explicitly manage and record the execution of business processes [60]. PAISs inherently support structured event logs because they are designed around process executions.

Most software systems, however, are non-process-aware, are not designed around process models, and do not natively produce event logs [61]. Enabling execution data mining in these systems requires explicitly converting software logs to event logs. The key challenge in this conversion is data quality, as software logs are incomplete, inconsistently structured, and lack the process-level information required to correlate individual log entries into meaningful execution traces [62].

Obtaining event logs suitable for mining requires an instrumentation strategy and a well-defined data model that specify what data to collect, the granularity, and the structure. Previous work has proposed instrumentation strategies and domain-specific meta-models for event data collection [63, 56], but these efforts address specific instrumentation contexts rather than providing a general, standardized conceptual model for software execution event data.

2.4.3 Events, Traces, and Pattern Properties

A software execution *event* e in a software system log is a tuple representing a single method invocation recorded during system execution, defined as:

$$e = (i, x, t, a, d, c, \delta), \text{ where:}$$

- i is the unique identifier of the event,
- x denotes the execution instance of the system that triggered the event,
- t is the timestamp that indicates when the event occurred,
- a is the system method (i.e., activity) that invoked the event,

- d is the duration of execution of the method that triggered the event,
- c is the position of the event in the sequence of events generated by the same execution instance x, and
- δ is the depth of the method call that triggered the event within the execution stack.

A *trace* τ of a system is a finite, ordered sequence of events generated by all method invocations during a single execution, and events are ordered by timestamp from earliest to latest. All events in a trace are associated with the same execution instance x, which is also referred to as the *trace identifier*.

An *event log* E_L of a system is a finite collection of its traces $E_L = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where each τ_i represents a complete execution trace of the system. Each trace is associated with a unique execution instance identifier x, and the event log captures the operational history across n execution traces.

A *pattern*, P, is an ordered sequence of events $\langle e_1, e_2, \ldots, e_k \rangle$ from an event log E_L. Events from P can appear in multiple traces in the event log. A *sequential pattern* $P = \langle e_1, e_2, \ldots, e_k \rangle$ occurs in the trace $\tau = \langle e'_1, e'_2, \ldots, e'_m \rangle$ if there exist indices $1 \leq i_1 < i_2 < \ldots < i_k \leq m$ such that $e_j = e'_{i_j}$ for all $1 \leq j \leq k$. The *pattern length* $|P| = k$ is the number of events in the pattern.

The *support* of a sequential pattern X in the event log E_L, denoted by Sup(X), is defined as the number of traces in E_L that contain X as a subsequence, where matching is performed on the activity attribute a of each event, regardless of the specific timestamp at each event.

$$\text{Sup}(X) = |\{\tau \in E_L \,|\, X \sqsubseteq \tau\}|$$

A pattern X is frequent if $\text{Sup}(X) > \sigma_{min}$, where σ_{min} is a user-defined minimum support threshold that controls the minimum number of traces in which a pattern must appear to be considered significant.

The *confidence* of a sequence Y relative to sequence X, denoted by $\text{Conf}(X \rightarrow Y)$, is the conditional probability that a trace containing X also contains Y.

$$\text{Conf}(X \rightarrow Y) = \frac{\text{Sup}(X \cup Y)}{\text{Sup}(X)},$$

where $X \cup Y$ denotes the traces that contain X and contain Y. It holds that $\mathrm{Conf}(X \rightarrow Y) \in [0, 1]$.

For a frequent pattern $P = \langle e_1, e_2, \ldots, e_k \rangle$ that occur n times in E_L, let $\delta_{j,i}$ denote the call depth of the i-th event in the j-th occurrence of P. The minimum call depth of occurrence j is $min_i(\delta_{j,i})$, denoting the lowest call stack depth of a method during the j-th occurrence. The *average depth* of the pattern P is defined as:

$$\text{average depth}(P) = \frac{1}{n} \sum_{j=1}^{n} \min(\delta_{j,i}),$$

where it averages the minimum depth of each pattern across the number of occurrences.

The *average execution* time of pattern P across the n occurrences is defined as:

$$\text{average execution time}(P) = \frac{1}{n} \sum_{j=1}^{n} \left(\sum_{i=1}^{k} d_i \right),$$

where d_i is the duration of the i-th event in the pattern P with the k events during the j-th occurrence. It sums up the duration of each event in the pattern and averages across all occurrences in the event log.

2.4.4 Sequential Pattern Mining

Sequential pattern mining uncovers meaningful sequences within datasets, helping to enhance system diagnostics, performance optimization, and predictive maintenance strategies [64]. It was initially developed to extract sequential patterns from transactional databases [65], where the relevant patterns do not need to be contiguous. These algorithms are mainly classified into Apriori-based, vertical-format–based, and pattern-growth algorithms [66]. Apriori-based algorithms, such as GSP(Generalized Sequential Pattern), generate candidate sequences iteratively by extending frequent sequences of length k to produce candidates of length $k + 1$, pruning infrequent subsequences [67]. It uses the principle that all subsets of a frequent sequence must also be frequent. This approach incurs a high computational cost due to repetitive database scans and the exponential growth of combinations in large datasets. Vertical format-based algorithms, like SPADE, transform the database into a vertical id-list structure where each sequence lists its occurrences, enabling efficient counting through intersection operations [68] without repetitive scan.

Pattern growth algorithms such as PrefixSpan [69], avoid candidate generation entirely by recursively projecting the database into prefix-projected subsets and growing patterns directly from frequent prefixes by using a Frequent Pattern(FP)-tree structure. PrefixSpan is well-suited to execution-tracing mining due to its efficient sequence mining, its performance, and its avoidance of repetitive database scans and data transformations in Apriori-based and Vertical format-based algorithms. For example, consider the transaction database entries $\langle A, B, C\rangle$, $\langle A, B\rangle$, $\langle A, C\rangle$, $\langle B, C\rangle$, $\langle A, B, C\rangle$. The FP-Tree compresses these transactions by grouping shared prefixes into a tree structure, as represented in Fig. 2.3.

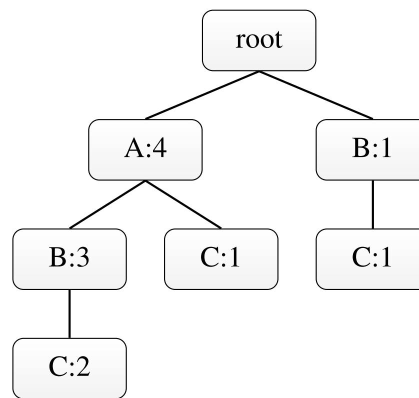


Figure 2.3: Example FP-tree for transaction database.

2.4.5 Process Mining

Process mining is an active discipline that bridges data science and process management, enabling process discovery, analysis, and improvement using the events recorded by PAIs [70]. It encompasses three primary activities: process discovery, conformance checking, and process enhancements [71]. Although process mining typically applies to PAIs, it can be applied to software systems to capture the runtime behavior of applications, using execution event logs [56, 72]. It enables the discovery of execution patterns, the identification of component interactions, and the reconstruction of software models using runtime data.

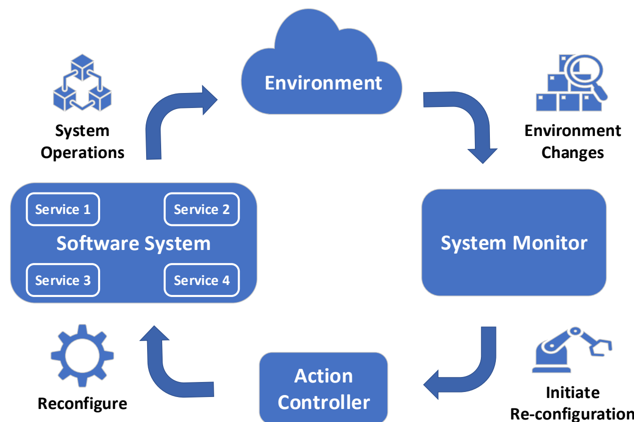


Figure 2.4: A typical self-adaptive system architecture.

2.5 Self-adaptive Software Systems

The monitoring and maintenance of software systems are mainly manual activities that require ongoing investment in fault detection, performance optimization, and operational management [29]. With the growth of distributed, cloud-native microservice architecture, the number of services, the heterogeneity of environments, and the frequency of operational events make manual monitoring and maintenance challenging [5]. Self-adaptive systems enable software to monitor its own behavior and autonomously respond to changing environmental conditions. Figure 2.4 illustrates the self-adaptive system architecture. It monitors environmental changes, identifies and performs reconfigurations, operates in the system environment, and the process continues.

2.5.1 Autonomic Computing

As a solution for the growing complexities in the software systems, IBM introduced the concept of autonomic computing [29, 31]. It enables software systems to automatically manage themselves, by dynamically adapting to changing environment conditions to achieve business objectives [29, 31]. It possesses four properties: self-configuration, in which adaptation is based on a changing environment; self-healing, in which automatic

recovery from failures is achieved; self-optimization, in which the system continuously optimizes itself by observing its behavior; and self-protection, in which proactive decision-making is used to avoid failures. The autonomic computing provides the foundation for self-adaptive software systems by applying self-configuration, self-healing, self-optimization, and self-protection in the software engineering context [30].

2.5.2 The MAPE-K Control Loop

The MAPE-K control loop performs monitoring, analysis, planning, and execution activities, with a shared knowledge base, as shown in Fig. 2.5. This control loop architecture is used as the dominant approach for implementing self-adaptive software systems [29].

- Monitor: The system continuously collects runtime data from the environment, including execution events, resource metrics, and operational data.
- Analyze: The collected data is analyzed to determine whether the system has anomalies, performance degradations, or requirement violations.
- Plan: If the analysis phase identifies a requirement for adaptation, the plan phase determines the appropriate corrective action. This involves selecting a predefined set of adaptive operations/configuration changes.
- Execution: The execution phase implements the planned adaptation and restarts the new monitoring cycle.

The shared Knowledge stores the models, policies, historical data, and system state required for other phases of the loop [31].

2.5.3 Self-Adaptive Microservices and Multi-Agent Systems

Applying self-adaptation to microservice-based systems presents both opportunities and challenges [28]. The independent deployability of microservices enables individual service-level adaptation. A service can be reconfigured, scaled, or replaced without affecting the rest of the services. However, the availability of a large number of distributed heterogeneous services increases the complexity of self-adaptation.

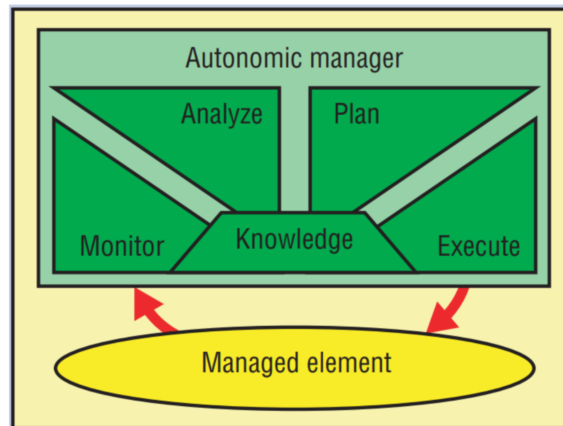


Figure 2.5: MAPE-K control architecture [29].

A multi-agent system (MAS) is a collection of distributed agents that interact within a shared environment [73]. MAS architectures are particularly suited to problems characterized by distribution, heterogeneity, and decentralized decision-making. In software systems, the MAS model has been applied to service-oriented and distributed architectures to enable adaptive behavior without central coordination [74]. Each service or component is modeled as an agent with its own perception of local state, its own decision-making logic, and the ability to communicate with neighboring agents. This decentralized model provides resilience: the failure of one agent does not compromise the decision-making capacity of others and scalability, as each agent operates locally without requiring global knowledge.

2.5.4 Runtime Performance Metrics For Self-Adaptive Systems

To evaluate self-adaptive systems in the dynamic operating environment and measure the effectiveness of self-adaptation, the following metrics are defined:

- **Response Time (RT)** is the total time it takes to send a request and the response. Requests can be initiated by either a user or a request generator.
- **Average Response Time (ART)** reflects the overall user-perceived latency of the responses follows:

$$ART = \frac{1}{N}\sum_{i=1}^{N} RT_i\,,\ \text{where:}$$

N is the total number of requests, and RT_i is the response time of i-th reqeust.

- **Adaptation Effectiveness (AE)** measures the gain in average response time after restructuring the system as follows:

$$AE = \frac{ART_{\text{before}} - ART_{\text{after}}}{ART_{\text{before}}}\,,\ \text{where:}$$

$$ART_{\text{before}} = \text{Average response time in the interval before adaptation, and}$$
$$ART_{\text{after}} = \text{Average response time in the steady state after adaptation.}$$

  The adaptation effectiveness values are interpreted as follows:

  - $AE > 0$: effective adaptation - the restructuring reduces the average response time,
  - $AE = 0$: no impact - the restructuring does not affect the average response time, and
  - $AE < 0$: ineffective adaptation - restructuring increases the average response time.

- **Throughput (TP)** is defined as N/T_{total}, where N is the total successfully processed requests and T_{total} is the total time taken to complete the requests.

- **Queue Size (QS)** in a system that contains a request queue to admit system requests before being served; the QS defines the number of requests waiting in the queue to be served at a specific point of time t. The QS indicates the system performance; the higher the QS, the lower the system performance.

- **Incoming Request Rate (IRR)** is defined as the external load imposed on a service at a unit of time. It is represented as $IRR = N/\Delta t$, where Δt represents the sampling interval and N denotes the requests received during the Δt sampling interval.

- **Memory Consumption (MC)** represents the amount of heap memory actively utilized by the Java Virtual Machine (JVM). It is defined as $H(t) - F(t)$ where $H(t)$ denotes the total heap memory and $F(t)$ denotes the total free memory. These metrics are collected programmatically during the system execution using the Java Runtime API.

- **Memory Gain (MG)** quantifies the relative reduction in total memory footprint achieved by the restructured system compared to the baseline system over the execution time window. Time weighted mean memory usage, $\bar{M}$ for the duration T is calculated as follows:

$$\bar{M} = \frac{1}{T} \int M(t)\, \mathrm{dt}\,.$$

  Then, Memory Gain (MG) is calculated as follows:

$$MG = 1 - \frac{\bar{M}_r}{\bar{M}_b}\,, \text{where}$$

  $\bar{M}_r$ and $\bar{M}_b$ are the time weighted means of the restructured system and the baseline systems, respectively. The values are interpreted as follows:

  - $MG > 0$: Effective utilization - reduced memory usage compared to baseline.
  - $MG = 0$: No impact - no effect on memory usage relative to baseline.
  - $MG < 0$: Ineffective utilization - increases memory usage compared to baseline.

2.6 Summary

The fundamental knowledge required for the contributions of this thesis is presented in this chapter. The evolution of software architecture from monolithic, service-oriented, to microservices architecture established the requirement for architectural migrations. The approaches used for legacy monolithic systems to microservices migration were examined, highlighting the limitations of static and artifact-driven approaches and the importance of dynamic data-driven identification of microservice boundaries. Dynamic software analysis and instrumentation were introduced, covering the principles of runtime observation, the instrumentation approaches supported by the Kieker monitoring framework, the structure of execution events and traces, the distinction between software logs and structured event

logs, and the gap between existing distributed tracing standards and the representational needs for software execution analysis. Process mining and sequential pattern mining were introduced as concepts for mining execution event data in software systems. The five parameters: support, confidence, pattern length, call stack depth, and execution time, that contribute to microservice identification from execution traces were defined.

The background on autonomous systems, self-adaptive systems, and the MAPE-K control loop was analyzed, while the background on multi-agent systems introduces the decentralized agent model that enables service-level adaptation without central coordination. Together, these foundations reveal limitations in existing approaches, including reliance on infrastructure metrics rather than rich runtime execution data, reactive adaptation, and a lack of decentralized service-level decision-making, establishing a requirement for a runtime data-driven, self-adaptive microservice-based system that can respond to dynamic operational conditions. Moreover, the performance metrics defined in the chapter: throughput, average response time, adaptation effectiveness, queue size, incoming request rate, memory consumption, and memory gain, form the complete evaluation framework.

Analyzing the existing work conducted in related areas is essential to validate the identified research gaps and questions. The next chapter systematically reviews the prior research on microservices migration, self-adaptation, and the utilization of software execution data.

Chapter 3

Related Work

This chapter reviews the existing research literature and related work of this thesis contributions: a performance-aware approach to microservice identification using historical execution traces; a self-adaptive microservice-based system with decentralized, proactive decision-making; and a standardized conceptual data model for software execution event data.

Section 3.1 comprehensively examines existing work on monolithic to microservices migration, covering static, dynamic, hybrid, and domain-driven identification approaches, evaluation criteria used in the literature, and a systematic review of reengineering techniques. Section 3.2 reviews existing work on self-adaptive microservices-based systems and self-adaptive approaches. Section 3.3 reviews existing work on formalizing software execution data, including studies that convert software logs to event logs. The chapter concludes by summarizing the gaps and positioning the contributions of this thesis relative to the existing work.

This chapter is derived from:

Thakshila Imiya Mohottige, Artem Polyvyanyy, Colin Fidge, Rajkumar Buyya, Alistair Barros, Reengineering software systems into microservices: State-of-the-art and future directions, *Information and Software Technology*, Volume 183, 2025, 107732, ISSN 0950-5849.

3.1 Reengineering Legacy Systems To Microservices

The reengineering of legacy monolithic systems into microservices has attracted significant research attention, as the microservice architecture has been widely adopted in industry. A substantial body of work has proposed methods, techniques, and tools for identifying microservice boundaries in legacy systems, with evaluation criteria to assess the quality of decompositions. Therefore, a systematic review is necessary to examine the concerns addressed in existing studies and to understand how individual contributions relate within the broader research landscape.

Several studies have reviewed the literature on microservice migration [75, 76, 77, 27, 78, 79, 80, 81]. Reviews of model-driven, domain-driven, and architectural refactoring approaches for microservice identification are presented in [75, 79], while the techniques applied during the requirements analysis and design phase, along with corresponding evaluation methods, are examined in [76]. Quality assessment criteria, quality-driven migration approaches, and quality attribute analysis for decomposed microservices are reviewed in [77, 27]. Migration techniques, their applicability to different system types, validation methods, and associated challenges are covered in [78, 80], where [80] additionally defines a roadmap for modernizing legacy systems encompassing decomposition, execution, validation, monitoring, and infrastructure concerns. A taxonomy of service identification approaches [81] combines inputs, processes, outputs, and usability criteria of monolithic to microservices migration.

Table 3.1 compares the existing reviews related to microservices migration studies. It confirms that existing reviews are limited both in scope and in the number of primary studies examined. Key aspects of reengineering legacy monolithic systems into microservices are inadequately covered, including the full spectrum of identification approaches and techniques, tools that support the reengineering process, evaluation methods for the resulting microservice systems, and the challenges and limitations of existing approaches. Hence, a systematic literature review was conducted to address these gaps by providing a comprehensive analysis of 117 primary studies, offering a structured understanding of microservice identification techniques with respect to software architectural properties, and identifying directions for future research on monolithic to microservices migration.

Table 3.1: A comparison of existing literature reviews on reengineering of software systems into microservices.

Literature review	[75]	[76]	[77]	[27]	[78]	[79]	[80]	[81]
Review period/year	2013–2019	2020	1998–2018	2016–2022	2019	2018	2020	2019
Number of reviewed papers	27	31	29	58	20	10	62	41
Comparison of research questions								
What are the techniques/approaches/patterns for legacy software reengineering?	◐	●	○	○	●	●	●	◐
What types of systems have the existing reengineering techniques been applied to?	○	○	○	○	●	●	○	○
What tools are used for reengineering monolithic systems into microservices?	○	○	○	○	○	○	○	○
What inputs/outputs are used by the existing reengineering techniques?	○	○	○	○	○	○	●	◐
What driving forces/evaluation criteria are used for the identified microservices?	○	●	○	○	○	○	●	◐
How reengineering processes/techniques are validated?	○	○	○	○	●	○	○	○
What quality-driven/assessment criteria are used for reengineering?	○	○	●	●	○	○	○	○
What quality attributes are analyzed, and how have they been implemented for reengineering?	○	○	●	●	○	○	○	◐
What are the challenges of reengineering legacy software systems into microservices?	○	○	○	○	●	○	○	○
What usability aspects, advantages, and disadvantages/limitations are highlighted?	○	○	○	○	○	○	○	◐
What are the roles and responsibilities involved in the identification of microservices?	●	○	○	○	○	○	○	○

● Addressed ◐ Partially addressed ○ Not addressed

3.1.1 Systematic Literature Review Process

A systematic literature review presented in this section follows the guidelines for performing systematic literature reviews in software engineering [82, 83]. The review examined existing methods, techniques, and tools for reengineering, to understand their limitations and identify directions for future research. The following research questions guided the review:

LRQ1 How did research on the reengineering of software systems into microservice-based systems develop over time?

LRQ2 What approaches are used to reengineer software systems into microservice-based systems, and how are reengineered systems evaluated?

LRQ2.1 What classes of approaches (e.g., static and dynamic) exist?

LRQ2.2 What tools exist, and which level of automation do they support?
LRQ2.3 Which techniques/algorithms are used?
LRQ2.4 How are the reengineered systems evaluated?

LRQ3 What are the challenges and limitations of existing methods to reengineer software systems into microservice-based systems?

3.1.1.1 Search Protocol and Selection Criteria

Primary studies were selected from five databases widely used to index publications: Web of Science,[1] Scopus,[2] ScienceDirect,[3] ACM Digital Library,[4] and IEEE Xplorer Digital Library.[5]. The search keywords were selected to maximize the chances of identifying relevant papers. The search query used for the Web of Science database was:

> (TS = (microservice* AND (reengineer* OR re-engineer* OR redesign* OR re-design* OR discover* OR identify* OR refactor* OR rearchitect* OR re-architect* OR migrate*))) AND (WC = (Computer Science)) AND (DT = (Article OR Book Chapter OR Proceedings Paper)) AND (LA = (English)).

3.1.1.2 Study Selection and Data Extraction

Figure 3.1 summarizes the study selection process. The initial search for relevant papers over the five databases was conducted on the 23rd of January 2023. It retrieved 4,843 references from the five databases. After multiple filtering steps, 107 primary studies were identified. A subsequent snowballing stage, involving both forward and backward referencing, identified a further 10 relevant works. The final set of primary studies comprises 117 articles.

To systematize the knowledge extracted from the primary studies, the taxonomy development method proposed in [84] was followed. This iterative approach identifies concepts and their characteristics and groups them into dimensions, guiding the evaluation of the resulting taxonomy for completeness and robustness. Following the definition

[1] https://clarivate.com/webofsciencegroup/solutions/web-of-science
[2] https://www.scopus.com/
[3] https://www.sciencedirect.com/
[4] http://portal.acm.org/
[5] https://ieeexplore.ieee.org/

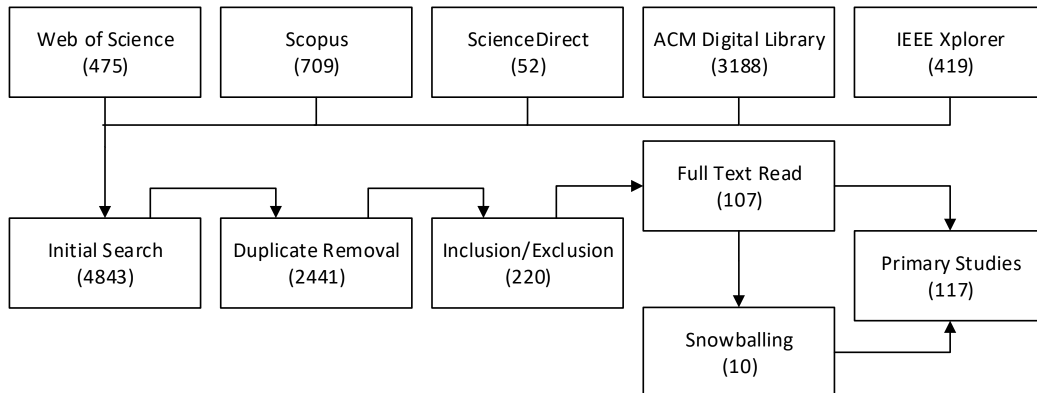


Figure 3.1: Literature review study selection process.

of classification criteria aligned with the research questions, the primary studies were analyzed in depth, and relevant findings were recorded for subsequent synthesis.

3.1.2 Results: Monolithic to Microservices Migration

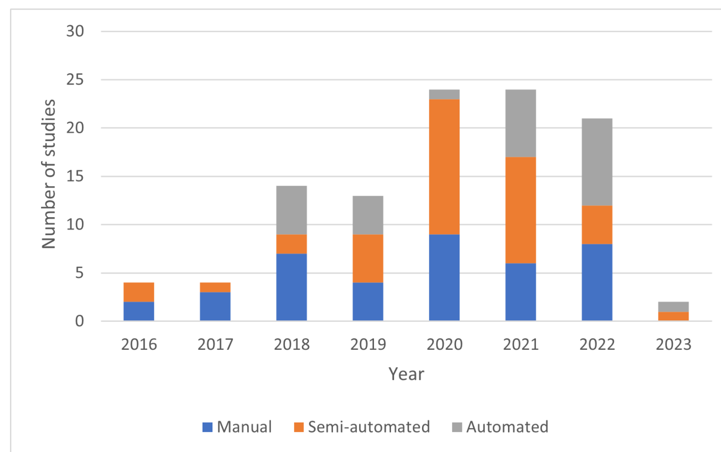
This section presents the results of the systematic literature review, based on the identified research questions. The primary study list is available in the Chapter A. Table 3.2 summarizes the classification of the selected 117 papers. The majority of the papers (71%) explain legacy system migration strategies, while most of the remaining papers (25%) focus on industry interviews and case studies. A small number of papers (4%) discuss greenfield development, where new system implementation in a microservice-based architecture is considered. Greenfield development was included in the analysis since it is applied in the context of artifact-based microservice extraction.

Table 3.2: SLR paper classification details.

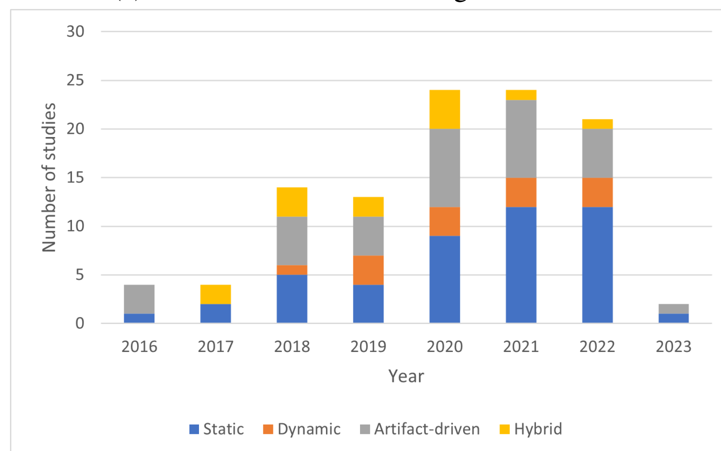
Type of study	Number of papers
Software system migration studies	83
Case studies and industry interviews	30
Greenfield development	4

3.1.2.1 LRQ1) How did research on the reengineering of software systems into microservice-based systems develop over time?

Research on reengineering software systems into microservices began in 2016 and has grown substantially since then, with the majority of studies published between 2020 and 2022. As illustrates in Figure 3.2, the identified approaches are classified into four categories: static, dynamic, artifact-driven, and hybrid analysis.



(a) Level of automation in migration studies.



(b) Study types in microservice migration.

Figure 3.2: Number of studies changes over time.

Static analysis uses source code, database schema, and version control history to identify microservice boundaries. *Dynamic analysis* uses runtime execution data to derive service boundaries. *Artifact-driven analysis* relies on system design artifacts, including

UML diagrams, use cases, user stories, and domain models, with domain-driven design (DDD) and functional-driven design being the most commonly adopted patterns within this category. *Hybrid approaches* combine two or more of the approaches discussed.

Static analysis is the most prevalent approach, accounting for 44% of the reviewed studies, followed by artifact-driven analysis at 32%. Dynamic and hybrid approaches are less common, each representing 12% of studies. In terms of automation, semi-automated techniques are the most frequently proposed(38%), followed closely by manual approaches (37%), while fully automated techniques account for the remaining 25%. However, a notable shift towards automated approaches has been observed in more recent studies, reflecting growing interest in reducing human effort in the reengineering process.

3.1.2.2 LRQ2.1) What classes of approaches exist?

The full taxonomy of the approaches identified in Section 3.1.2.1: static, dynamic, artifact-driven, and hybrid analysis, is presented under this research question. Figure 3.3 illustrates the full taxonomy of approaches and sub-categories.

Artifact-driven analysis examines high-level system artifacts such as domain models, business processes, data flow diagrams, system features, and domain semantics to support microservice identification. Domain models and languages(e.g., UML, ADL, use cases, and user stories) capture system structure and business terminology, enabling the identification of service boundaries [22]. The dependencies of business processes are analyzed (e.g., data, structural, semantic, and control), often represented as matrices to reveal interactions [85]. Data flow diagrams provide insights into relationships between processes, data stores, and external entities, supporting dependency-based grouping of components [86]. Furthermore, system features and functionalities are decomposed into smaller tasks to identify cohesive functional groups [87], while semantic analysis leverages domain-specific vocabulary and similarity measures to cluster related entities and operations into candidate microservices [24].

Static analysis is the most widely used approach for the identification of microservices, relying on artifacts derived from source code, databases, and version control systems.

- *Source code analysis* examines structure, interaction, and semantic aspects of the

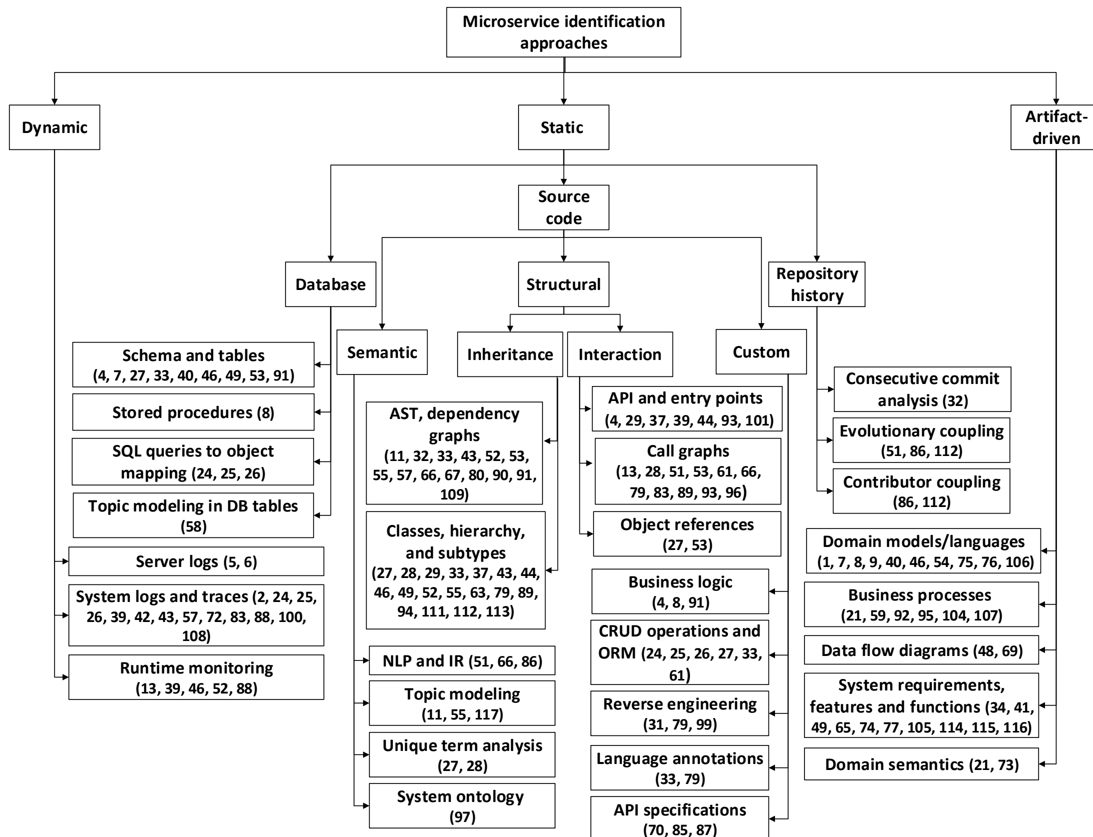


Figure 3.3: Classification of migration approaches.

system to group related functionalities into cohesive services. Structural analysis explores class hierarchies, dependencies, and abstract syntax trees (ASTs) to construct dependency graphs, while interaction analysis uses APIs, execution paths in the source code, and call graphs(both context-sensitive and insensitive) to identify interconnected components [15]. Semantic approaches apply natural language processing, information retrieval, and topic modeling techniques (e.g., LDA, SLDA) to cluster classes based on lexical similarity [23]. Additionally, custom analysis considers business logic, the persistence layer (e.g., CRUD operations), reverse engineering outputs, annotations, and API specifications to derive service boundaries.

- *Database analysis* focuses on database schema, structure, and relationships between data, which are critical to defining boundaries under the ”database per microservice”

principle [88, 15]. This includes analyzing schemas, tables, stored procedures, SQL queries, and applying topic modeling to group related data entities into cohesive units.

- *Version control analysis* leverages commit histories to identify relationships between system components through evolutionary and logical coupling. Techniques such as co-change analysis and coupling graphs group classes that frequently change together, while contributor-based analysis links system evolution to developer activities, supporting microservice decomposition [16].

Dynamic analysis considers the software system as a black box to identify microservice boundaries by analyzing runtime behavior, execution traces, and interaction patterns. This approach is commonly categorized into three types. *Server log analysis* examines web server logs (e.g., Apache Tomcat, WildFly) to identify frequently invoked URIs and request patterns based on access frequency, responses, and data sizes, enabling the grouping of related operations. *System log analysis* relies on instrumenting the source code, after using aspect-oriented programming, to capture execution logs, which are then analyzed or processed using process mining tools such as Disco to identify frequent execution paths and dependencies [9, 6, 18]. *Runtime monitoring* observes the system during execution using tools such as Elastic APM and Dynatrace, collecting runtime data to support the identification of service boundaries and system reengineering.

Hybrid approaches combine artifact-driven, static, and dynamic analysis techniques to improve microservice identification. In such approaches, a primary approach and a complementary approach were observed to support microservice identification. For example, static analysis may be applied initially, with its results further refined using dynamic analysis, a primary approach, and a complementary approach to support microservice identification [21, 89, 90]. Alternatively, artifact-driven analysis can act as the dominant approach, with static analysis contributing additional structural and dependency information [91].

3.1.2.3 LRQ2.2) What tools exist, and which level of automation do they support?

Multiple tool categories are available based on the approaches used to examine the monolithic system. There are tools for the static analysis of software systems, database adminis-

tration, runtime monitoring, visualization, architectural validation, and load simulations. These tools, technologies used, and respective study IDs are listed in Table 3.3. Other supporting tools used for testing, clustering, and other specific purposes are listed in Table 3.4.

In existing studies on microservice migration, automatic and semi-automatic tools were implemented to support the migration process, aligning with the concept in the underlying study. Such migration frameworks, their levels of support, and accessibility to source code/tools are listed in Table 3.5. Frameworks that provide microservice recommendations based on primary inputs, such as source code, log files, and system artifacts, are considered automated. Studies involving tools at different stages of migration, such as data extraction and system modeling, are categorized as partially automated.

3.1.2.4 LRQ2.3) Which techniques/algorithms are used?

Two techniques were observed in microservices migration: system modeling and microservice extraction. System modeling techniques are used to interpret or model software systems, creating their abstract representations, while microservice extraction techniques are applied to identify the microservices within the interpreted systems, thereby defining boundaries of potential microservices. Figure 3.4 and Fig. 3.5 illustrate the system modeling and extraction techniques. The leaf nodes refer to the relevant study IDs listed in Chapter A.
System modeling techniques provide a structured representation of legacy systems to support microservice identification, with several commonly used techniques as follows:

- *Graph-based modeling* is the most prominent technique, representing system elements (e.g., classes, methods, business processes, or database entities) as vertices and their relationships as weighted edges. These relationships are derived from structural dependencies, method calls, semantic similarity, runtime interactions, or evolutionary coupling. Weights are often computed using coupling metrics, tf-idf, cosine similarity, or topic modeling techniques such as LDA and SLDA [53, 50, 92].

- *Matrix/table-based modeling* represents system elements and their relationships in matrix form, where entries capture frequencies, dependencies, or similarity measures. Various matrices (e.g., class-to-class, use-case-to-entity, or business-process

Table 3.3: Tools for monolithic system analysis.

Purpose	Tool/Library	Details	Technology	Study IDs
Static analysis (source code)	Java Call Graph (open source)	Reads jar files to collect the method calling sequences. Dynamic Analysis is possible but is used only in static context. (`https://github.com/gousiosg/java-callgraph`)	Java	13, 111, 112
	Java Parser/Symbol Resolver (open source)	Constructs abstract syntax tree for structural dependency extraction. (`https://javaparser.org/`)	Java	11, 33
	Mondrian (open source)	Performs static source code analysis (`https://github.com/Trismegiste/Mondrian`)	PHP	27, 28
	WALA (open source)	Analyses project class hierarchies and generates call graphs. (`https://github.com/wala/WALA`)	Java, JavaScript	53, 57
	Soot (open source)	Models source code to analyse, instrument, optimize, and visualize applications. (`https://soot-oss.github.io/soot/`)	Java, Android	29, 53
	Doop & Datalog (open source)	Conducts static analysis of source code using Datalog engine. (`https://plast-lab.github.io/doop-pldi15-tutorial/`)	Java, Android	53
	JackEE (open source)	Provides static analysis of Java Web applications with enterprise framework support. Additional parameter is used for Doop framework to run JackEE. (`https://github.com/plast-lab/doop`)	JEE applications	53
	Spoon (open source)	Parses source code into an abstract syntax tree. (`https://spoon.gforge.inria.fr/`)	Java	61
	Structure 101 (commercial)	Validates software architectures by visualizing their structures from source code. (`https://www.sonarsource.com/structure101/`)	Java, .Net, C/C++	2, 46, 71, 111
	Sonargraph Architect (commercial)	Offers architecture checks, duplicate code detection, virtual refactorings, cyclic dependency resolution, and comparison with previous versions. Suppors Git repository mining. (`https://www.hello2morrow.com/products/`)	C#, C/C++, Java, Python 3	77
Semantic analysis (source code)	ANTLR (open source)	Parses the source code to generate grammar for language recongnition. (`https://www.antlr.org/`)	Java, C#, Python, Go, C++, Swift, JavaScript, TypeScript	97
Static analysis (database)	SchemaSpy (free software)	Generates Web-based visual representations by analysing database metadata. (`https://schemaspy.sourceforge.net/`)	Java-based tool	2, 71, 72
	DBeaver (free and commercial versions)	Provides tools for database adminstration and schema analysis. (`https://dbeaver.io/`)	MySQL, Maria DB, PostgreSQL, SQLite	46
	JSqlParser (open source)	Parses SQL statements and translates them into hierarchies of Java classes. `https://github.com/JSQLParser/JSqlParser`)	Java, SQL	61
Dynamic analysis	Kieker (open source)	Monitors and analyzes runtime behavior of software systems. (`https://kieker-monitoring.net/`)	Java, .Net, C, VB	13, 39, 88, 108
	Elastic APM (commercial)	Supports real-time monitoring, performance analysis of incoming requests/responses, database queries, cache invocations, and external calls. (`https://www.elastic.co/solutions/apm`)	Java-based Web, Data access frameworks, application servers, messaging frameworks, AWS	2, 72
	Disco (free and commercial versions)	Analyzes event logs to identify call graphs and enables automated process discovery. (`https://fluxicon.com/disco/`)	Log files of software systems	2, 24, 25, 26, 72
	ExplorViz (open source)	Provides runtime monitoring and visualization of software systems (`https://explorviz.dev/`)	Applied to Java-based systems	46
	django-silk (open source)	Profiles and inspects the django framework, analyzing HTTP requests and database queries. (`https://github.com/jazzband/django-silk`)	Python django framework-based tools	52

Table 3.4: Additional tools used for system analysis.

Purpose	Tool	Details	Technology	Study IDs
Testing	Jmeter (open source)	Provides load simulation (`https://jmeter.apache.org/`)	Java	6, 7, 108
	Gatling (commercial)	Provides stress testing (`https://gatling.io/`)	Java, Kotlin, Scala	16
Reverse engineering	MoDisco (open source)	Provides model-driven reverse engineering of the source code (`https://wiki.eclipse.org/MoDisco/`)	Java, JEE, XML	31, 99
Topic modeling	GuidedLDA (open source)	Provides topic modeling using latent Dirichlet allocation (`https://guidedlda.readthedocs.io/en/latest/`)	Python	55
Clustering	SciPy (open source)	Provides hierarchical clustering and generates dendograms (`https://www.scipy.org/`)	Python	61, 111
Optimization algorithm	Jmetal (open source)	Supports multi-objective optimization algorithms NSGA II and NSGA III (`https://jmetal.sourceforge.net/`)	Java	82, 96
Document enrichmnet	WordWeb, WordNet (public dictionary)	Lexical databases to identify synonyms for topic modeling (`https://wordnet.princeton.edu/`)	Word dictionary	58
Lines of code count	CLOC (open source)	Blank, comment, and physical lines counting (`https://github.com/AlDanial/cloc`)	Java, C, Python	12

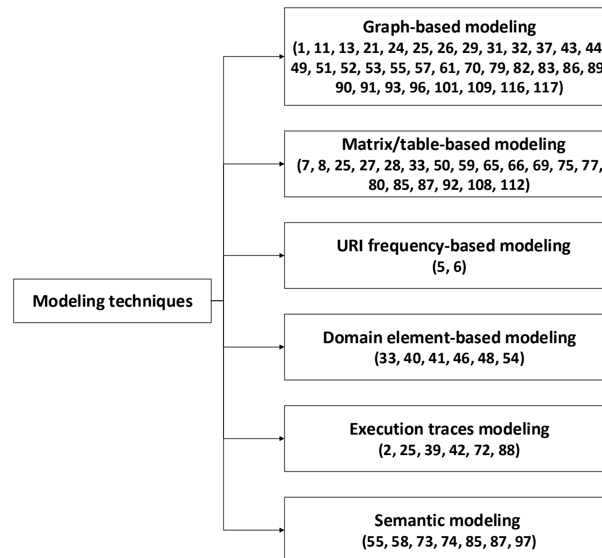


Figure 3.4: Classification of legacy system modeling techniques.

dependencies) are analyzed using similarity metrics, such as cosine similarity, to identify cohesive groups of components.

- *URI-based modeling* focuses on web applications by analyzing request/response interactions captured in server logs. Frequently invoked URIs, along with metrics

Table 3.5: Tools and levels of automation; automated (A) and partially automated (PA).

Study ID	Level of automation	Available artifacts
1	PA	https://github.com/ServiceCutter/ServiceCutter
11	A	https://github.com/miguelfbrito/microservice-identification
24	PA	https://github.com/AnuruddhaDeAlwis/NSGAII
25	PA	https://github.com/AnuruddhaDeAlwis/Subtype
26	PA	https://github.com/AnuruddhaDeAlwis/NSGAIIFOROptimization
29	PA	https://github.com/utkd/cogcn
37	PA	https://github.com/Rofiqul-Islam/logparser
39	PA	https://github.com/wj86/FoSCI
42	A	https://www.ibm.com/cloud/mono2micro
51	PA	https://github.com/loehnertz/Steinmetz https://github.com/loehnertz/semantic-coupling
52	PA	https://github.com/tiagoCMatias/monoBreaker
55	A	https://essere.disco.unimib.it/wiki/arcan
61	A	https://github.com/socialsoftware/mono2micro
70	A	https://github.com/HduDBSI/MsDecomposer
77	PA	https://github.com/RLLDLBF/FeatureTable
79	PA	https://gitlab.com/LeveragingInternalArchitecture/IdentificationApproach
86	PA	https://github.com/gmazlami/microserviceExtraction-backend https://github.com/gmazlami/microserviceExtraction-frontend
89	A	https://drive.google.com/drive/folders/1TQaS8etLr-32d0RXwC1Le-IOMVaDBcSS

such as response time and size, are used to infer service boundaries.

- *Domain-element-based modeling* relies on high-level system representations such as UML diagrams, data flow diagrams, and context maps. These models are typically analyzed manually to identify bounded contexts and service candidates.

- *Execution trace modeling* leverages runtime logs to capture actual system behavior, identifying frequent execution paths, call patterns, and functional units. Techniques such as process mining and execution graph analysis are used to derive microservice candidates [19, 9, 6, 90, 18].

- *Semantic-based modeling* uses domain-specific and linguistic information, such as system vocabularies, topics, and API terminology, to group related components based on conceptual similarity [93, 25, 94, 95].

Microservices extraction techniques are illustrated in Fig. 3.5. After modeling the system using the aforementioned techniques, the extraction process identifies potential microservice candidates. The main techniques are as follows:

- *Clustering* is the predominant approach for the extraction of microservices. Hierarchical clustering (agglomerative and divisive) is used when the number of clusters is

unknown, while K-means clustering requires a predefined number of clusters and allows exploration of different decomposition granularities [50]. Variants such as temporo-spatial and collaborative clustering extend hierarchical methods. Density-based clustering, such as DBSCAN, groups elements based on density, identifying clusters of closely related components while treating sparse elements as noise. It relies on the neighborhood distance and the minimum point threshold [96].

- *Community detection* techniques apply network analysis algorithms to graph-based models to identify groups of closely related components. Common methods include Girvan-Newman [97], Epidemic Label Propagation(ELP) [98], and modularity-based approaches such as Louvain, which does not require prior knowledge of the number of clusters and has shown strong performance [23, 99].

- *Multi-objective optimization techniques* uses algorithms such as NSGA-II and NSGA-III to optimize multiple conflicting objectives(e.g., coupling, cohesion, modularity) when identifying microservices. However, studies show that NSGA-III does not consistently outperform NSGA-II [89, 100, 101].

- *Graph-based extraction* is widely used due to graph-based system modeling. This technique extracts microservices by analyzing relationships between nodes (e.g., classes, components) and grouping them based on edge weights representing dependencies, similarity, or interaction strength.

- *Matrix-based extraction* applies clustering (commonly hierarchical agglomerative clustering) to matrix representations of system relationships, such as similarity of dependency matrices, to identify cohesive groups.

- *Manual and expert-driven techniques* often used in artifact-driven approaches, rely on domain expertise to interpret system artifacts and identify microservice boundaries, typically complementing automated techniques.

3.1.2.5 LRQ2.4) How are the reengineered systems evaluated?

After extracting microservices, the reengineered system must be evaluated to ensure that it preserves the functional behavior of the legacy system while meeting performance and

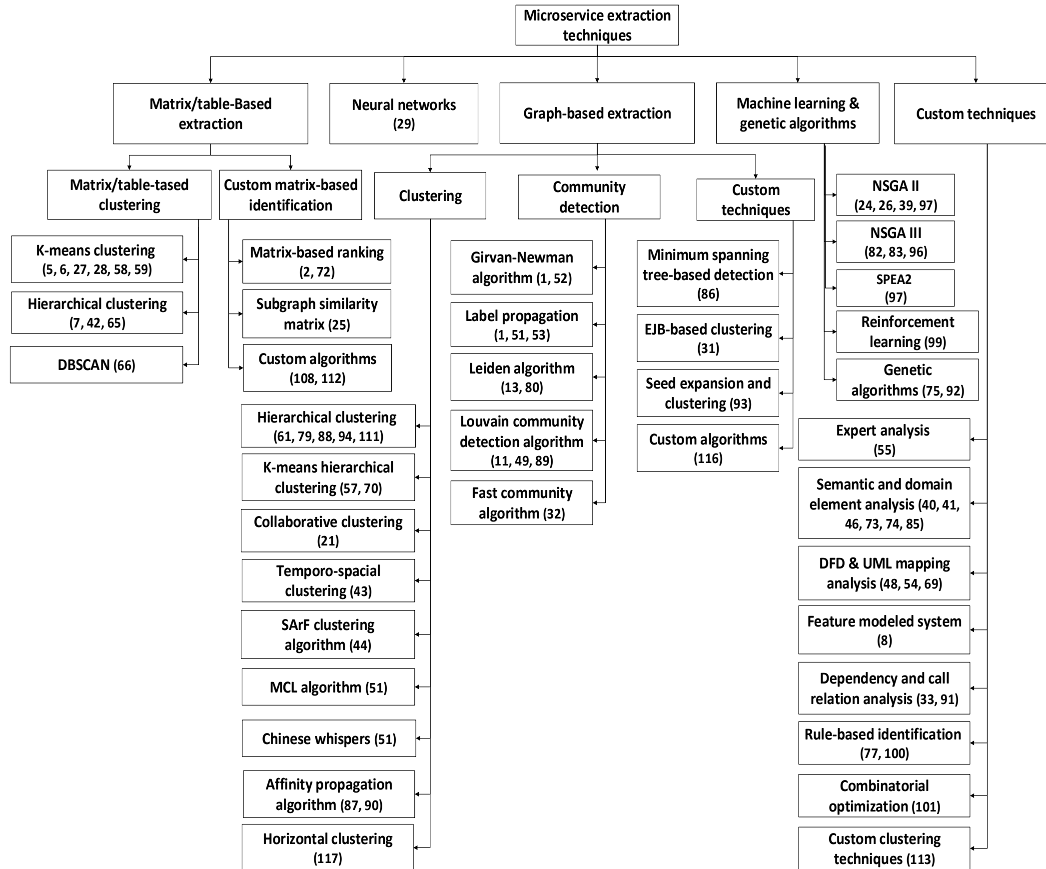


Figure 3.5: Classification of microservice extraction techniques.

quality requirements. In addition to maintaining functionality, the system should exhibit key quality attributes such as modularity, low coupling, high cohesion, and appropriate service granularity.

Evaluation approaches in the literature include expert-based assessments, prototype implementation, industrial case studies, and cross-system comparison with existing techniques. However, a widely used approach is property-based evaluation, which computes quantitative metrics. Some studies incorporate hyperparameter optimization to identify optimal decomposition configurations [96, 95].

Evaluation metrics categorization is illustrated in Fig. 3.6. *Runtime performance* assesses the efficiency of the reengineered systems. *Modularity* evaluates the quality

of decomposition and separation of concerns, often using clustering-based metrics [97]. *Coupling* assesses the interdependence between services. *Cohesion* measures the intra-dependencies within within a service. *Independence* captures the ability of services to evolve independently with minimal cross-service impact, often analyzed through change frequency and interaction metrics [6]. Finally, *quality of ecomposition* evaluates how effectively functionalities, data, and responsibilities are distributed across services, including measures such as business context purity and transaction distribution [15]. Additional evaluation measures include similarity-based comparisons with reference architecture(e.g., MoJo metrics), API identification accuracy, and cluster quality metrics such as the Silhouette coefficient [102, 103, 104, 105].

The applications used for evaluation are mainly open-source benchmark applications that have been used to implement and evaluate the proposed reengineering solutions. Most of the reengineered systems were Java-based, with limited PHP systems identified. Applications reengineered in at least two works are listed in Table 3.6.

Table 3.6: Evaluated applications.

Application	**Study IDs**	**Technology**
JPetStore	7, 11, 13, 39, 42, 53, 57, 66, 80, 88, 90, 108, 115, 116	Java
Acme Air	6, 7, 29, 53, 66, 80, 93	Java
Cargo Tracking System	1, 7, 48, 49, 77, 80, 115	Java
Daytrader	29, 43, 53, 55, 57, 66, 93	Java
Springblog	39, 80, 88, 90, 113	Java
Jforum	39, 88, 89, 90	Java
Apache Roller	39, 88, 90	Java
Spring boot pet clinic	44, 66, 89, 93	Java
E-commerce system	49, 58	Java
Microservices event sourcing	66, 70	Java
Kanban board	66, 70	Java
TFWA (Teachers Feedback Web Application)	5, 7	Java
Train Ticket Microservice Benchmark	12, 88	Java
Plants by WebSphere	29, 53	Java
SugarCRM	24, 25, 26, 27	PHP
ChurchCRM	24, 25, 26, 27	PHP

In several studies, extensive evaluations have been reported, in which the proposed

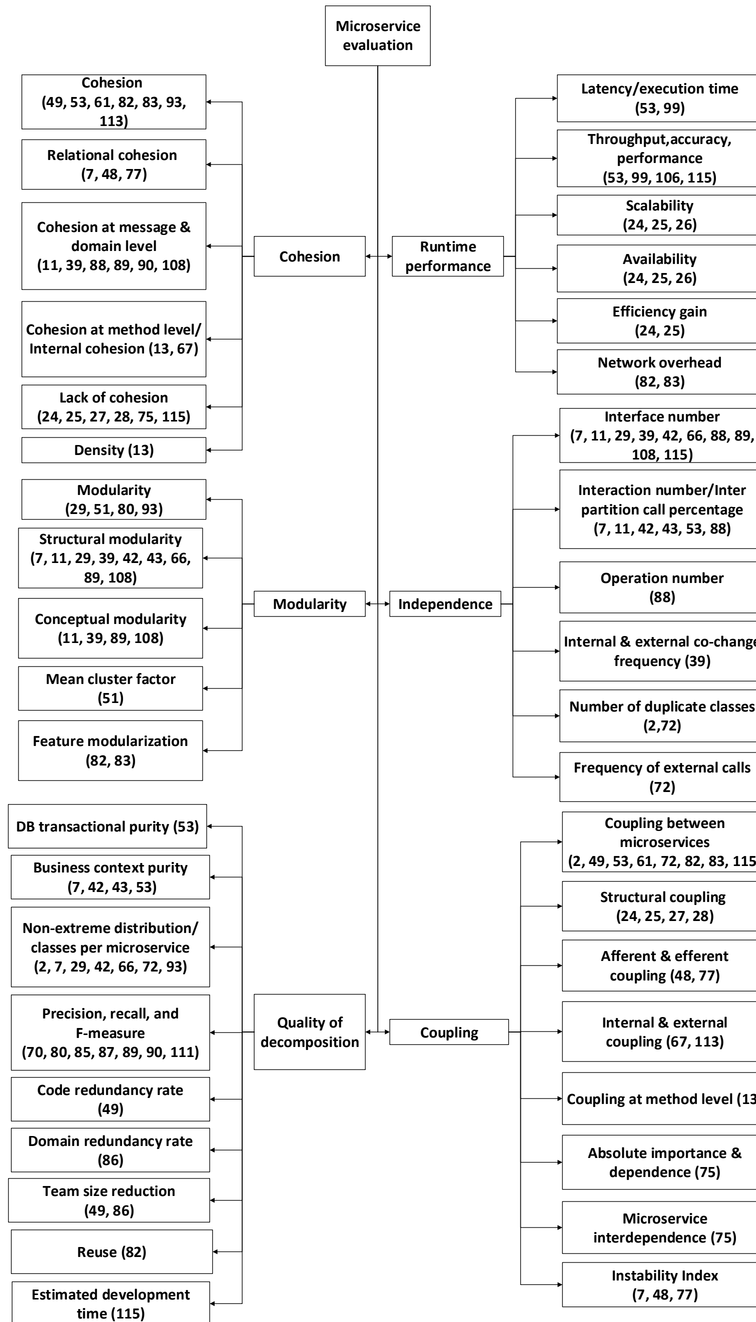


Figure 3.6: Classification of evaluation techniques.

approaches are compared with existing migration frameworks, validated using benchmark applications, and assessed through prototyping and quantitative property measurements.

The evaluation criteria adopted by key studies are summarized in Table 3.7.

Table 3.7: Cross-system evaluated frameworks.

ID	Name	Evaluation Type	Details
1	Service Cutter	Prototype and case study	Evaluated the approach with cargo tracking system and trading system.
6		Compared with legacy system	ACME air web application compared in monolith and microservices versions.
7	Green Micro	Cross comparison	Cross compared with FoSCI, CoGCN, Mono2Micro, MEM, Service Cutter, API , DFD , and Business process analysis.
11	Topic Modeling	Case study	Evaluated using 200 Java Spring applications selected from GitHub for property calculations.
13		Cross comparison	Evaluated against Fosci, DFD approach and distributed source code representation.
25, 26, 27		Compared with legacy system	Compared Sugar CRM and Church CRM legacy and microservice versions.
28		Compared with legacy system	Compared Dolibarr open-source enterprise management system legacy and microservice versions.
29	Co_GCN	Prototype	Evaluated using Daytrader, Plants by websphere , Acme-Air, and Diet App
39	FOSCI	Cross comparison	Compared with LIMBO, WCA, and MEM approaches.
42	Mono2Micro	Cross comparison	Compared with FOSCI, CO_GCN, and Munch approaches.
48	DFD	Cross comparison	Cross compared with Service cutter and API analysis approach.
49	Knowledge Graphs	Prototype	Evaluated the approach with E-commerce application and cargo tracking system.
51	Steinmetz	Case study	Evaluated properties using 14 applications.
53	CARGO	Cross comparison	Evaluated against Mono2Micro, CoCGN, MEM, and FOSCI.
61		Case study	Applied the approach to 121 monolith applications for comparison.
66	Hierarchical DBSCAN	Benchmark and cross comparison	Evaluated existing microservices projects - Spring PetClinic, Microservices Event Sourcing, and Kanban Board Cross compared with Bunch, CoGCN, FOSCI, MEM, and Mono2Micro frameworks.
70	API Graph	Benchmark and cross comparison	Evaluated existing microservices projects Kanban, Money Transfer, Piggy Metrics, Microservices Event Sourcing, and Sock Shop Cross compared with Service Cutter.
77	Feature Table	Cross comparison	Evaluated against DFD, Service Cutter, API analysis frameworks.
85	Interface Analysis	Prototype	Precision and recall properties evaluated using Cargo tracking system.
86	MEM	Case study	Evaluated 21 projects for logical, semantic, and contributor coupling.
87		Benchmark	Evaluated existing microservices projects Kanban Board, and Money Transfer app. Amazon Web Services and PayPal evaluated using OpenAPI specifications.
88	FOME	Cross comparison	Evaluated LIMBO, WCA, and MEM frameworks.
89		Case study and cross comparison	Evaluated against existing Service Cutter and topic modeling frameworks. Five applications including PetClinic JForum 3, and Compiere applications evaluated for accuracy.
90		Case study and cross comparison	Evaluated against FOME and multi-objective evlutionary search frameworks. Property evaluated in JPetStore, SpringBlog, Jforum, Roller applications.
108	Log2MS	Case study and cross comparison	Evaluated against FOSCI and Mono2Micro frameworks. Property evaluated in four applications including JPetStore.
115	Backlog	Case study and cross comparison	Evaluated against Domain-driven design, Interface analysis, and Service Cutter frameworks. JPetStore, Cargo Tracking System, and Foristom Conferences(real life system) used for evaluation.

3.1.2.6 Results Discussion

This systematic literature review analyzed 117 primary studies on microservices migration and reveals significant insights of the current state of legacy systems to microservices reengineering.

Dominance of Static and Artifact-driven Approaches: Static and artifact-driven analysis collectively accounts for 76% of the reviewed studies, with structural source code analysis being the most prominent technique. Prominent static analysis studies, including CoGCN [14], CARGO [15], and MEM [16], have demonstrated promising results in structural decomposition. However, static analysis faces a fundamental limitation in legacy systems: it cannot distinguish between actively executing code and deprecated or unreachable segments, leading to inaccurate identification of service boundaries [18, 6]. Although artifact-driven approaches are based on design documentation, the unavailability and incomplete documentation in legacy software systems are the primary limitations.

Underutilization of Dynamic Analysis: Despite its accuracy advantage, dynamic analysis accounts for only 12% of the reviewed studies, representing a significant gap in the literature. Existing dynamic approaches, including FoSCI [6], FoME [18], Mono2Micro [19], and the process mining-based approach [9], demonstrate that runtime execution data provide more accurate insights into legacy system behavior than static source code analysis. However, these approaches share two critical limitations. First, they rely on predefined test cases or a single execution session, which cannot capture the full range of system behavior across varying workloads and usage patterns. Second, none of these studies evaluate the performance implications of identified microservices for throughput and latency. Dynamic analysis, therefore, presents a significant opportunity for future research with advanced runtime data analysis techniques and performance-implication measures.

Limited Performance Evaluation: The evaluation of reengineered systems has primarily focused on structural quality. Although these metrics assess the structural soundness of the decomposition, they do not capture whether the migration delivers operational benefits. Microservices impact system performance due to factors including network communication overhead, data serialization cost, and distributed coordination complexity that are absent in monolithic systems. These factors can directly impact throughput and latency

due to the additional overhead introduced to the microservice system. A structurally sound decomposition that degrades system performance represents ineffective migration. The impact of parameters such as service granularity and service boundaries on the runtime performance of the resulting microservices remains largely uninvestigated. This gap is critical: evaluating reengineered systems solely on structural properties provides incomplete details on migration effectiveness.

Emerging Techniques: Machine learning and AI-based approaches, including Microminer [53] and distributed source code representation [106], have begun to appear in the literature, but remain underutilized and lack cross-comparison with established migration techniques. Genetic algorithm-based approaches, such as Microservice Backlog [107], incorporate granularity and performance objectives along with structural metrics, but the impact of microservice granularity on runtime performance remains to be investigated. Database migration also remains a challenging and underexplored area, particularly the performance implications of distributed transactions under the widely recommended 'database per microservice' pattern.

3.1.3 Summary of Study Gaps and Further Requirements

The analysis of existing microservice identification approaches reveals two interconnected limitations that remain unaddressed in the literature. First, despite its proven ability to provide accurate insights, dynamic analysis for microservice migration is an underutilized area. Second, the evaluation of reengineered systems has been predominantly limited to structural quality metrics. The performance implications of identified microservices, specifically throughput and latency, remain largely uninvestigated. These gaps collectively indicate a need for a dynamic, execution-data-driven approach to microservice identification that considers not only structural quality but also the runtime performance characteristics of resulting microservices.

3.2 Self-Adaptive Software Systems

The operating environment of a software system is highly dynamic and requires significant effort to monitor and maintain. However, the monitoring and maintenance operations are

largely manual activities [29]. Service-oriented architectures [39] and microservices [3, 2] are software architectural styles that describe a system as a group of communicating services. Often, such a system identifies a primary service responsible for communications within the system. When a system receives a request, it delegates the necessary functions to its constituent services. During the execution of a specific functionality, a service can communicate with other services in the system. The response to the user is provided by aggregating the results of potentially multiple service calls. As service-based systems scale and become more heterogeneous, monitoring and maintenance activities of these distributed systems become increasingly complex and costly, leading to the development of autonomous self-adaptive approaches for monitoring and maintaining software systems. This section reviews existing work on self-adaptive service-based systems and dominant self-adaptive approaches.

3.2.1 Review of Existing Self-Adaptive Service-based Systems

This section compares existing self-adaptive service-based systems. A systematic review has been conducted to compare existing distributed component/(micro)service-based systems. Scopus [6] and Web of Science [7] digital libraries were used with the keywords "self-adapt*", "software*", and "architecture" to retrieve relevant studies. The search query applied to the Scopus database was:

> TITLE-ABS-KEY (self-adapt*) AND TITLE-ABS-KEY (software*) AND TITLE-ABS-KEY (architecture) AND (LIMIT-TO (DOCTYPE ,"cp") OR LIMIT-TO (DOCTYPE ,"ar") OR LIMIT-TO (DOCTYPE ,"ch")) AND (LIMIT-TO (SUBJAREA,"COMP")) AND (LIMIT-TO (LANGUAGE ,"English"))

The initial search yielded 1,496 publications. After successive screening and filtering steps, including title and abstract screening against relevance to distributed component-based and (micro)service-based self-adaptive systems, and full-text analysis of candidate studies, 21 relevant studies were identified for the comparison.

[6] https://www.scopus.com/
[7] https://clarivate.com/webofsciencegroup/solutions/web-of-science

3.2.2 Results: Existing Self-Adaptive Service-Based Systems

A comparative analysis of reviewed studies is presented in Table 3.8. The characteristics presented in the table describe the key architectural and functional properties of the proposed system.

self-adaptive systems automatically adapt their behavior based on changing environmental conditions, workload variations, or internal performance metrics without requiring external intervention. *Decentralized decisions* imply that control and decision-making are distributed across multiple components without relying on a centralized controller. The *agent-based* property refers to the availability of autonomous agents that operate with local intelligence and collaborate to achieve global objectives. Operating at the *application layer* indicates that adaptation and coordination mechanisms are implemented above the network layer. A *learning-based* approach incorporates data-driven decisions to continuously improve the quality of decisions over time. Being *proactive* means that the system anticipates future conditions and takes preventive actions instead of reacting to events. *Split/merge actions* refer to the ability to dynamically divide and consolidate services to optimize performance and resource utilization. Finally, *generalizable* denotes that the proposed approach is domain and technology-independent.

The legend accompanying Table 3.8 clarifies the level of support each study provides for the evaluated characteristics. A property is *addressed* by the approach if it explicitly describes, proposes, or implements the corresponding characteristic. A property is *partially addressed* if it is considered or implemented to a limited extent. A property is *not addressed* if it is neither implemented nor discussed as part of the corresponding approach. Finally, the *information not available* label is used when insufficient details are provided about the property. This classification ensures a transparent and structured comparison, facilitating the identification of strengths, limitations, and research gaps across the reviewed studies.

Table 3.8 summarizes the comparison of 21 reviewed studies in the eight identified properties. All reviewed studies, except EPF4M [108], are self-adaptive systems, confirming the relevance of the selected studies. However, comparison reveals significant gaps across the remaining properties.

Decentralized decision-making is addressed in only seven of the studies, with two partially addressed studies. This confirms the centralized control of existing approaches.

Agent-based design is the least addressed property in the reviewed studies. Only

two studies fully adopt an agent-based model, with two further partially incorporating agent-based principles. The remaining 17 studies do not adopt an agent-based approach, indicating a significant gap in the existing literature for autonomous, agent-driven decision-making. Proactive behavior is addressed in only three studies, confirming that the predominant adaptation model across the reviewed studies is reactive, triggered only after a condition or threshold has been breached, rather than anticipating future states. Learning-based adaptation is addressed in only two studies, with one further partially incorporating a learning mechanism, indicating that the vast majority of existing approaches rely on static, pre-defined adaptation rules rather than data-driven learning.

Split-and-merge actions, which facilitate dynamically dividing or consolidating services to optimize performance, are addressed fully in only one study and partially in another. This property is the most consistently absent in the reviewed studies, reflecting a fundamental limitation in the scope of adaptation actions that existing approaches support.

Operating at the application layer is the property that has been the most consistently addressed among the reviewed studies, with the majority implementing adaptation mechanisms above the network layer. Generalizability is addressed in approximately half of the reviewed studies, suggesting that while many approaches are designed to be domain-independent, a substantial proportion remain specific to a particular application context.

In general, the comparison reveals a clear and consistent gap in the existing literature. No reviewed study fully addresses all eight properties simultaneously. The combination of decentralized decision-making, agent-based design, proactive behavior, learning-based adaptation, and split-and-merge actions are the properties that are individually rare and collectively absent in existing work.

3.2.3 State-of-the-Art Self-Adaptive Approaches

Several approaches have been established in the existing self-adaptive systems. This section examines the four most relevant classes: adaptive service mesh, gossip-based systems, MAPE-K-based frameworks, and multi-agent systems.

3.2.3.1 Adaptive Service Mesh

An adaptive service mesh is a self-managing communication layer that provides traffic management, observability, and security [128, 129, 130, 131, 132]. It typically collects traffic

Table 3.8: A comparison of existing self-adaptive software systems.

Study Ref	Self-adaptive system	Decentralized decisions	Agent-based	Application layer	Learning -based	Proactive	Split/ merge actions	Generalizable
[109]	●	●	○	○	○	●	○	●
[110]	●	○	○	●	○	○	○	○
MOSES [111]	●	○	○	○	○	○	○	●
SAFDIS [112]	●	●	◐	●	◐	●	○	○
S/T/A [113]	●	○	○	●	○	N/A	○	●
SAMSP [114]	●	○	○	●	○	○	○	●
SASSY [115]	●	○	○	●	○	○	○	●
[116]	●	●	●	●	●	●	○	○
[117]	●	○	○	○	○	○	○	○
[118]	●	○	○	●	○	○	○	●
[119]	●	○	○	●	○	○	○	○
[120]	●	◐	◐	●	●	○	○	○
DySOA [121]	●	○	○	●	○	○	○	○
[74]	●	●	●	N/A	●	○	○	●
EPF4M [108]	◐	○	○	●	○	○	●	●
[122]	●	○	○	●	○	○	◐	○
[123]	●	●	○	○	○	○	○	◐
DARE [124]	●	●	○	●	○	○	○	◐
[125]	●	●	◐	●	○	○	○	●
[126]	●	○	○	●	○	○	○	○
[127]	●	◐	○	●	○	○	○	●

● Addressed ◐ Partially addressed ○ Not addressed N/A Information not available

patterns, detects security violations and anomalies, and performs self-adaptive actions such as reroute, circuit breaker [133], service scaling, and security policy enforcement. It responds to runtime signals and policy-driven behaviors. For example, incoming message rate thresholds are used for adaptive decision-making. However, service mesh primarily adapts the network and data-layer behavior rather than application-level decomposition, and does not support structural adaptations such as service split or merging.

3.2.3.2 Gossip-Based Systems

Gossip-based systems use an epidemic protocol for data dissemination and failure detection between nodes in a distributed system [134, 135]. Self-adaptive systems based on gossip protocols monitor disseminated data and make adaptive decisions such as routing changes, resource scaling, workload rebalancing, and data replication [124]. Gossip-based

systems avoid centralized control and rely on local interactions between nodes. However, the objective of gossip-based is efficient data dissemination and eventual consistency through failure detection. It does not explicitly re-architect the system to ensure target performance guarantees based on execution behavior; instead, it uses network-level signals for communication.

3.2.3.3 MAPE-K-Based Frameworks

The MAPE-K control loop is the predominant self-adaptive software technique [136], organizing adaptation activities through a feedback cycle of monitoring, analysis, planning, and execution [137, 138, 139]. Centralized MAPE-K frameworks, including SASSY [115], MUSIC [140], and REACT [141], consolidate data collection, analysis, and decision-making in a single controller. Adaptation decisions in these systems are primarily driven by stakeholder-defined policies and pre-defined QoS models [115]. Decentralized MAPE-K [142, 143, 12] distributes the feedback loop between peer nodes and aligns more closely with the Service Colonies model in its use of local knowledge and distributed loops [144]. However, MAPKE-K is a general control model applied in diverse domains, not a prescriptive architectural style specifically designed for microservice systems. Furthermore, existing MAPE-K implementations rely on reactive, threshold-triggered adaptation; proactive structural decisions driven by execution-level behavioral data are not incorporated into MAPE-K-based studies.

3.2.3.4 Multi-Agent Systems

Multi-Agent Systems (MAS) [73] have several foundational principles, including agent autonomy, goal-directedness, and the ability to coordinate and negotiate. MAS is a broad engineering paradigm applied to complex distributed problem-solving across autonomous robotics, smart grid management, AI, and simulated systems [73]. Recent applications on MAS to self-adaptive service compositions [74] and LLM-based multi-agent systems [116, 145]. However, there are no studies on a specialized form of MAS integration in microservice-architecture-based self-adaptive systems, with concrete runtime operations including service splitting, merging, and deployment-aware behavior. Unlike general MAS frameworks, self-adaptive systems should be anchored to the specific operational context of microservice systems and driven by software execution data rather than abstract agent

communication protocols in multi-agent systems.

3.2.4 Summary of Study Gaps and Further Requirements

Across all four classes of approaches, a consistent pattern emerges: existing self-adaptive systems predominantly operate at the infrastructure or network layer, rely on centralized or reactive decision making, and use pre-defined rules or resource metrics rather than execution-level behavioral data as the primary input to adaptation decisions [32, 146]. Auto-scaling approaches optimize infrastructure resource utilization [147, 148, 103], but do not provide architectural adaptations. Architecture-based approaches such as Rainbow [149] and meta-manager model [150] provide global adaptation but rely on centralized control and constraint-based models rather than decentralized, execution-data-driven decisions. The introduction of self-management and self-architect principles grounded in runtime execution behavior, decentralized agent-based decision-making, and proactive structural adaptation remains an open challenge in the literature [5, 32]. Advancing microservice-based architectures towards autonomous self-management where individual services monitor their own execution behavior, make independent adaptation decisions without centralized coordination, and support structural operations such as service splitting and merging to maintain performance guaranties represents a promising direction for improving the resilience of microservice systems, but remains an open problem in the field.

3.3 Formalization of Software Execution Event Data

Runtime data generated during the execution of software systems serves as the foundation for a wide range of software system management activities, including microservice identification, runtime monitoring, performance analysis, and self-adaptive decision-making. As discussed in Chapter 2, capturing this data in a structured, interoperable, and reusable form across tools and domains requires a well-defined execution data model. This section reviews existing work on formalizing software execution data, examines the problem domains in which software logs are actively used for mining, and identifies gaps that motivate the development of a standardized conceptual data model.

3.3.1 Existing Execution Data Models and Standards

Existing work on formalizing execution data approaches the problem from two directions. Instrumentation-oriented studies propose domain models and meta-modes for collecting runtime traces, whereas business-process-oriented studies propose conceptual models for knowledge extraction from legacy systems.

Instrumentation-oriented studies include a formal definition and instrumentation strategy for collecting runtime traces based on the joint point-point cut model [63]. This work constructs a domain model that illustrates the key concepts of event logs, business transaction process mining, and software instruction, providing one of the earliest structured treatments of runtime trace collection. The process mining meta-model [62] defines the structural requirements for applying process mining techniques in software systems. However, neither of the studies formalizes a data model for software logs that directly facilitates event generation from software execution data.

Business-process-oriented studies propose conceptual models for knowledge extraction from legacy systems. The conceptual model for business processes [151] defines the process structure, business rules, and participants, aimed at supporting knowledge extraction from legacy software systems. A workflow metamodel [152] formalizes a model that is independent of workflow specification languages, enabling the recovery of business processes from source code using structural information. These studies are business-process-oriented and do not address the requirements for software execution data.

At the industry level, OpenTelemetry [34] has emerged as the dominant specification for distributed tracing, metrics, and logging in cloud-native and microservice-based systems. OpenTelemetry defines a data model for the distributed equivalent of a method-level execution record and specifies protocols for propagating trace context across service boundaries. However, its scope is limited to instrumentation and transport, but it does not define a conceptual data model for software execution events.

The requirements for the technology-independent conceptual data model for software execution data are not captured by the existing studies. Instrumentation studies are context-specific and do not generalize across all the tools and domains. Business-process models adopt a process perspective rather than a software execution context. Existing studies do not capture software-specific constructs such as method invocations, class instances, call stack depth, and thread identifiers. These limitations motivate a systematic analysis of how

software logs are actually used across analysis domains.

3.3.2 Review on Software Log Mining Applications

A systematic review of the problem domains in which software logs are used as a source to construct event logs for execution data mining is performed to address the gap in runtime execution data modeling. Process mining and specification mining were identified as the two dominant research areas that consume event logs derived from software systems. Consequently, a structured search query was designed to retrieve studies with the key words ”process mining”, ”specification mining”, and ”specification discovery”. The search was performed in two research databases, Web of Science [8] and Scopus [9], on April 15, 2025. The following is the search query used for Web of Science :

> (TS=(”process mining”) OR TS=(”specification mining”) OR TS=(”specification discovery”)) AND TS=(software) AND (TS=(”log”) or TS=”data”)).

An equivalent query was applied in Scopus. The initial search returned 200 results from Web of Science and 308 from Scopus, yielding 508 records in total.

3.3.2.1 Study Selection

The studies were selected using a multi-stage filtering criterion: duplicate records were removed, followed by title, abstract, and full-text read. Studies were incorporated only if they:

- Used software execution logs as input data.
- Reported attributes extracted or relationships relevant to event log construction.
- Applied process mining or specification mining techniques.

The exclusion criteria are as follows:

- Focused exclusively on infrastructure or system logs irrelevant to software behavior.

[8] https://www.webofscience.com/
[9] https://www.scopus.com/home.uri

- Lacked sufficient methodological details.
- Not in English.

Following this process, 47 studies were identified as relevant. An additional three studies were incorporated through snowballing, both backward-referenced examinees and forward-cited work, to ensure coverage of foundational and emerging work not captured by database queries, yielding a total of 50 studies.

3.3.3 Results: Problems Domains Addressed Using Software Logs

Software logs are widely used to support various mining tasks in multiple domains since runtime data provides better insights into actual behavior compared to static source code information [8]. These logs have served as a valuable source of behavioral and operational data, enabling the addressing of challenges spanning six broad categories: architecture reconstruction and design pattern detection, legacy system maintenance and modernization, runtime behavior analysis, system monitoring, anomaly and error detection, and specification mining. Across these domains, the nature and complexity of log properties vary considerably—from basic method-call information to rich temporal-constraint checks. Each problem category, the types of problems addressed, and the role of software logs are discussed below.

3.3.3.1 Architecture and Design Pattern Detection

Architecture provides an abstraction of the system, facilitating reuse, maintenance, and evolution. In addition, it supports more accurate impact and cost estimations during system change management [153]. However, software architecture erosion or drift from the original architecture is a common phenomenon [8], especially with legacy software systems. The unavailability of up-to-date architectural documentation is a primary concern, especially with legacy software systems [154]. Hence, reconstructing the system architecture is essential. Architecture mining applies reverse-engineering techniques by integrating static code analysis with execution data to reconstruct the underlying software architecture. It gathers method call data to capture interactions among system entities. In addition, process mining techniques are used to uncover hierarchical interaction models and process models,

which help identify the hierarchical structure of software systems and the relationships between their components [8].

The design pattern is a reusable solution to a recurring problem [155], which supports the development of well-structured and quality software [156], and provides helpful insights to understand and re-construct the system architecture [156]. Software logs have been utilized to discover design patterns, particularly when the source code is unavailable. Class and method call information derived from execution traces, along with known design pattern specifications, is used to detect recurring design structures and behavioral constraints in software systems.

3.3.3.2 Legacy System Maintenance and Modernization

Enterprise software systems require new features and functions to adapt to changing customer needs and corporate strategies[157]. This is adaptive maintenance, where applications are enhanced to adapt to business trends. With legacy systems, adaptive maintenance requires more human resources than developing the application itself [157]. Maintenance of aging software becomes difficult as updates gradually destroy the original structure [157]. Hence, it is challenging to accurately estimate the impact and cost on the source code when adding new business functionality. Software anti-aging methods, such as refactoring and restructuring, are essential to maintain software quality and ensure maintainability [157]. Due to the challenges involved in modifying legacy system source code, requirements are prioritized based on their impact on critical business operations. When prioritizing change requests, both the nature of business activities and the frequency of their execution are key considerations. This information can be gathered using process mining techniques that analyze runtime software data. Software logs serve as input for process discovery, helping to identify frequent work packages and their corresponding source code segments [157].

Legacy software systems support complex business operations that remain critical due to the high risk associated with system failures [151]. However, these systems often fail to meet current business expectations due to outdated or missing engineering documentation, obsolete technologies, inefficient and costly hardware usage, and challenges in change management [151]. Consequently, reengineering these legacy systems is necessary to improve their reliability and maintainability.

Microservices is a modern architectural style in which software systems are composed

of lightweight, loosely coupled, and highly cohesive services that communicate over well-defined interfaces. Software logs and process mining have been employed for microservices identification [49]. The dynamic aspects of software systems, like runtime dependencies and frequent method calls, are crucial for microservices identification, as they affect the performance and maintainability of microservices [49]. Extended log files capturing all execution paths, including UI calls, system use cases, and user simulations, are collected and used as input for process mining tools to identify highly correlated methods and classes that may serve as potential microservice candidates.

Restructuring architecture by discovering components and interfaces improves system understanding and facilitates maintenance [158]. Leveraging runtime data reveals functional interactions that support effective component decomposition. In component discovery, interaction methods were identified from the caller-callee relationships in execution data. The connector behavior model was then discovered from the interaction logs. Finally, the interface instances and cardinalities were identified based on the connector behaviors [158, 159].

Moreover, Process mining is applied to extract business processes for business-process-based restructuring and the migration of legacy systems. The logs collected by instrumenting the source code serve as input to an incremental process mining to identify business processes [151]. Subsequently, the restructuring is performed by grouping these identified business processes.

3.3.3.3 Runtime Execution and Behavior Analysis

Modern software systems often contain millions of lines of code and thousands of interdependencies, making them challenging to manage [160]. To address this complexity, execution data is leveraged to analyze the actual runtime behavior of software systems [161]. Behavioral models derived from this data provide insight into real system usage, support model-based testing, improve usability, and help localize performance issues and architectural bottlenecks.

In distributed component-based systems, runtime behavioral models can reveal both the internal behavior of individual components and their interactions [159]. Execution logs generated in real-time or collected from historical data are used to analyze user interactions, detect common usage patterns, and identify frequently accessed features [162]. These

insights contribute to better system optimization and customer satisfaction [163, 164]. Process mining plays a central role in discovering these workflows. Event logs for such analysis are generated via manual logging, source code instrumentation, or profiling tools—especially when source code is unavailable [163, 161]. The Kieker framework[10] discussed in Chapter 2 is the widely used instrumentation tool.

Behavior discovery differs from architecture reconstruction or reengineering approaches. While behavior discovery focuses on analyzing the internal dynamics of programs, architecture reconstruction typically emphasizes high-level component interactions, often treating internal component behavior as a black box [159]. Petri nets are frequently derived from execution traces to model system behavior [161]. However, concurrent execution of nearly independent software components often results in complex interleaved logs. Directly applying process mining techniques on such data often produces "spaghetti models", which are unstructured and hard-to-interpret visualizations. To overcome this, component-level behavioral models have been proposed to generate clearer and more structured representations [165]. Hence, the discovery of component models involves analyzing runtime execution data to construct models of individual software components [159]. First, the interface instances are identified from the event logs. Then, hierarchical event logs are constructed for each interface. Finally, hierarchical workflow nets are generated using these logs to represent the internal behavior of each component.

3.3.3.4 Monitoring and Observation

Microservices architectures inherently support scalability, maintainability, and deployability, making them well-suited to handle increasing system demands [166]. However, as the number of services grows, manual monitoring becomes infeasible and error-prone. To address this, distributed tracing combined with process mining techniques has been applied to effectively track microservice-based systems. A single user request in such systems can trigger interactions across multiple internal services. Without distributed tracing, correlating logs across services for troubleshooting requires significant manual effort. Distributed tracing addresses this by assigning a unique trace identifier to each request. It uses span and parent span IDs to capture the hierarchical relationships between remote method invocations [166]. These tracing tools detect inter-service calls, followed

[10] https://kieker-monitoring.net/framework/

by process mining tools to visualize system behavior through Petri net models.

The process models represent the dynamic behavior of software systems [63]. Such models are essential, particularly when the behavior of system components is not well documented or understood. Typically, process models are represented as directed graphs: nodes correspond to events recorded in logs, while edges represent transitions between these events [167]. Execution traces collected from the system are often converted into the XES (eXtensible Event Stream) format to facilitate process discovery. Once the process models are generated, they are validated against a predefined set of compliance properties to ensure correctness and conformity [168].

3.3.3.5 Anomaly and Error Detection

Security vulnerabilities and threats are unavoidable in software systems. The number of heterogeneous devices makes it even difficult to realize all possible threats[169]. Anti-virus and anti-malware tools are not effective against unknown attacks. Software logs are crucial for detecting vulnerabilities by capturing abnormal user and system behavior. However, vulnerability detection is still largely a manual process that relies on specialized approaches [169]. Similarly, this is applicable to error detection and failure analysis in computer applications [170]. Moreover, runtime testing, validation, and verification ensure the system reliability even after being deployed [171]

The volume of log data generated during the system execution makes it infeasible to perform manual analysis of security breaches [169]. Logs collected from software systems are passed to generate a more structured format(XES). Training logs are used to process discovery to infer the process model. Then, process discovery algorithms were applied to discover causal dependencies among the traces using algorithms like Inductive Miner. Then, an expert manual inspection is performed to evaluate the presence of suspicious activities. In an automated approach, conformance checking algorithms are used to calculate the alignment score and trace fitness metrics to measure the deviation from normal behavior [169, 170].

3.3.3.6 Specification Mining

The specification mining infers the software system specification from its execution traces [172]. The temporal specification mining extracts temporal properties from ex-

ecution traces.

A well-documented software system describes the features, modules, interactions, and the appropriate use of each module, which facilitates the implementation, refactoring, testing, and debugging of the system [173]. However, these documents are often poorly maintained and become outdated or incorrect due to infrequent updates and limited developer motivation, which increases the challenges in system maintenance [173]. In such cases, specification mining is performed to uncover hidden specifications of the software system. However, specification mining is complex and time-consuming and requires automated approaches [174]. Frequently occurring functions mean that users are more interested in those execution characteristics. Hence, by identifying frequent functions and timing information to mine the performance specifications. Thus, software execution data with frequency and timing information is used to mine system specifications.

3.3.4 Summary of Study Gaps and Further Requirements

The reviewed work collectively demonstrates that, while significant progress has been made in defining instrumentation strategies, event log formats, and business-process-oriented conceptual models, no existing work provides a general, standardized conceptual data model for software execution event data. Three specific requirements emerge from this analysis that existing approaches do not fulfill.

- Existing instrumentation strategies and domain models are designed for specific tools, frameworks, or application contexts, limiting their interoperability and reusability across different software systems and analysis domains.
- Existing event log formats and standards, including MXML, XES, and Open Telemetry, do not capture software-specific execution constructs at the level of abstraction required for software system analysis. Constructs such as method invocations, class instances, call stack depth, thread identifiers, and node-level contexts are absent from existing models, limiting their utility for tasks including microservice identification, performance analysis, and self-adaptive decision-making.
- None of the existing work defines a technology-independent conceptual data model that standardizes the representation of software execution events, traces, and the representation across the full range of entities involved in software execution. The

absence of such a model limits the systematic collection, integration, and reuse of execution data across software management activities.

These gaps collectively indicate that the formalization of software execution event data remains an open problem. A standardized conceptual data model that captures software-specific execution constructs, supports interoperability across tools and domains, and covers the full range of entities involved in software execution is required to maximize the utility of runtime data for microservice identification, self-adaptive decision-making, and broader software system management activities.

3.4 Summary

This chapter reviewed existing work across the three research areas addressed in this thesis: reengineering legacy monolithic systems to microservices, self-adaptive microservice-based systems, and the formalization of software execution event data.

The systematic review of 117 primary studies on legacy systems to microservice reengineering revealed that static and artifact-driven approaches dominate the field, while dynamic analysis is underutilized. Furthermore, advanced techniques like pattern mining have not been incorporated into dynamic-analysis-based migration approaches. Furthermore, the evaluation of reengineered systems is primarily based on structural quality metrics. The implications of Runtime performance remain uninvestigated in this area. These gaps collectively indicate the need for a dynamic, execution-data-driven approach to microservices migration, with structural and runtime performance measures.

The systematic review of 21 self-adaptive service-based systems revealed that no existing study fully addresses the architectural properties of agent-based design, decentralized decision-making, application-layer operation, learning-based adaptation, proactive behavior, split-and-merge action support, and generalizability. Existing systems primarily rely on centralized, reactive, and rule-based decision-making driven by infrastructure-level resource metrics. Furthermore, a comparison of existing self-adaptive approaches, including adaptive service mesh, gossip-based systems, MAPE-K-based frameworks, and multi-agent systems was conducted. Decentralized, proactive adaptation at the application layer driven by runtime execution data remains an open problem in service-based self-adaptive systems and across existing classes of approaches.

The review of the formalization of the execution data identified that existing instrumentation strategies, event log formats, and business-process-oriented conceptual models do not provide a general and independent conceptual data model for software execution event data. A systematic review of 50 studies in process mining and specification mining, where execution event logs are widely used, confirmed that software execution logs are actively used across six problem domains: architecture reconstruction, legacy system maintenance, runtime behavior analysis, monitoring, anomaly detection, and specification mining. However, existing models provide limited support for the constructs required in these domains. Hence, the construction of a unified conceptual model to capture execution data collection and representation remains an open problem in this area.

The three contributions presented in the following chapters, performance-aware microservice migration using execution data Chapter 4, runtime data-driven adaptation for microservice systems Chapter 5, and a unified model of software execution event data Chapter 6, address these identified gaps.

Chapter 4

Performance-aware Microservice Migration Using Execution Data

Microservice systems comprise small, loosely coupled, and highly cohesive services that interact with each other to provide system functionalities, offering advantages in scalability and flexibility through the use of multiple distributed services. However, migrating a legacy monolithic system to microservices architecture is challenging due to the complexities of tightly coupled code, managing data used by multiple services, and ensuring efficient inter-service communication. This chapter presents the ***MIST*** Framework, a sequential pattern-mining-based approach to identify microservices from legacy monolithic systems. *MIST* analyzes execution traces of a monolithic system to detect repetitive, coherent, low-level method invocation sequences and extracts them as independent, executable microservices. These microservices can be deployed independently while maintaining communication with other services in the system for operational continuity. Furthermore, the impact of different attributes and sequential patterns on microservice identification and performance is investigated. The results of open-source case studies show that *MIST* maintains high structural quality. Although relying on support can degrade performance, factors such as average call stack depth, confidence, pattern length, and execution time can lead to performance improvements.

This chapter is derived from:

Thakshila Imiya Mohottige, Artem Polyvyanyy, Colin Fidge, Rajkumar Buyya, Alistair Barros, MIST: Microservices Identification from Sequential Traces, *International Conference On Web Services, July 2026.*

4.1 Microservice Migration Using Sequential Traces

This chapter addresses RQ1: *How can historical executions of a software system be used to help migrate it to a microservice system that exhibits better performance?*. The question is motivated by three interconnected limitations in the current state of microservice migration research as identified in Chapter 2 and Chapter 3: the dominance of static analysis approaches that operate on source code irrespective of runtime behavior; the limited exploration of dynamic analysis techniques, particularly sequential pattern mining; and the lack of empirical performance validation in the evaluation of migrated systems.

This chapter proposes the *MIST* framework, a microservices identification framework grounded in sequential pattern mining of execution traces. *MIST* addresses the limitations in existing microservices migration by employing dynamic software analysis and empirically evaluating the performance implications of identified microservices. *MIST* analyses execution traces collected during the operational period of a monolithic system to detect repetitive, coherent sequences of low-level method invocations that recur consistently across diverse use cases, collectively covering the overall functionality of the system. These recurring sequences represent stable, self-contained units of system behavior that reflect how the system actually operates under real conditions, rather than how it is structurally organized in source code. The *MIST* framework follows the general methodology used in prior dynamic analysis studies [9, 18, 6]. However, it differs from existing work in that none of the previous studies have examined how sequential patterns and their associated attributes influence the identification of microservices or their performance implications.

MIST extracts these sequences as candidate microservices, each encapsulating a cohesive cluster of method invocations that co-execute regularly and can therefore be separated from the monolith without disrupting its functional continuity. To support the deployment of these candidates as services, *MIST* incorporates a code rewriting component that automates the structural transformation of the identified method clusters. This semi-automated process extracts the constituent methods from the monolithic codebase and restructures them into microservice stubs, self-contained, independently deployable units with well-defined interfaces. The extracted microservices maintain communication with the remainder of the system through these interfaces, ensuring that the operational behavior of the original monolith is preserved throughout the incremental migration process.

Sequential pattern mining is well-suited for execution traces, which consist of an

ordered sequence of discrete events [69, 65]. The structural quality of microservices migration is assessed using coupling, cohesion, and modularity metrics. Furthermore, this chapter empirically investigates the performance implications of migration by examining the influence of sequential pattern mining and program parameters: support, confidence, pattern length, call stack depth, and execution time, on the throughput and latency of the resulting system.

The remainder of this chapter is structured as follows: *MIST* framework(Section 4.2), evaluation(Section 4.3), implementation and reproducibility(Section 4.4), and the summary of the chapter(Section 4.5).

4.2 MIST Framework Overview

MIST is a semi-automated framework for identifying and extracting microservices from monolithic software systems using historical execution traces and sequential pattern mining. The framework operates in three phases: the automated pattern mining phase, which transforms execution traces into a ranked list of candidate microservice patterns; the manual microservice selection phase; and the microservices extraction phase, which translates selected microservices into deployable source code. Figure 4.1 provides an overview of the *MIST* pipeline, including the high-level steps in each phase of the framework.

4.2.1 Execution Trace Analysis and Pattern Mining

MIST framework is based on dynamic software analysis; the availability of a log file across the full range of operational conditions is essential. Enterprise software systems typically generate raw log files that contain developer-inserted logs, which often lack the necessary information for comprehensive pattern mining. To enable detailed analysis, additional runtime data, such as method signatures, parameter values, method entry and exit timestamps, call stack depth, and session identifiers, must be collected. However, as discussed in Section 2.3, manually updating log statements or instrumenting communication interfaces to incorporate this information is labor-intensive and error-prone, especially in large codebases. A commonly used solution is aspect-oriented programming, particularly the joinpoint-pointcut model, which supports the collection of execution data without modifying the original code [72]. As highlighted in Section 2.3, this allows unobtrusive

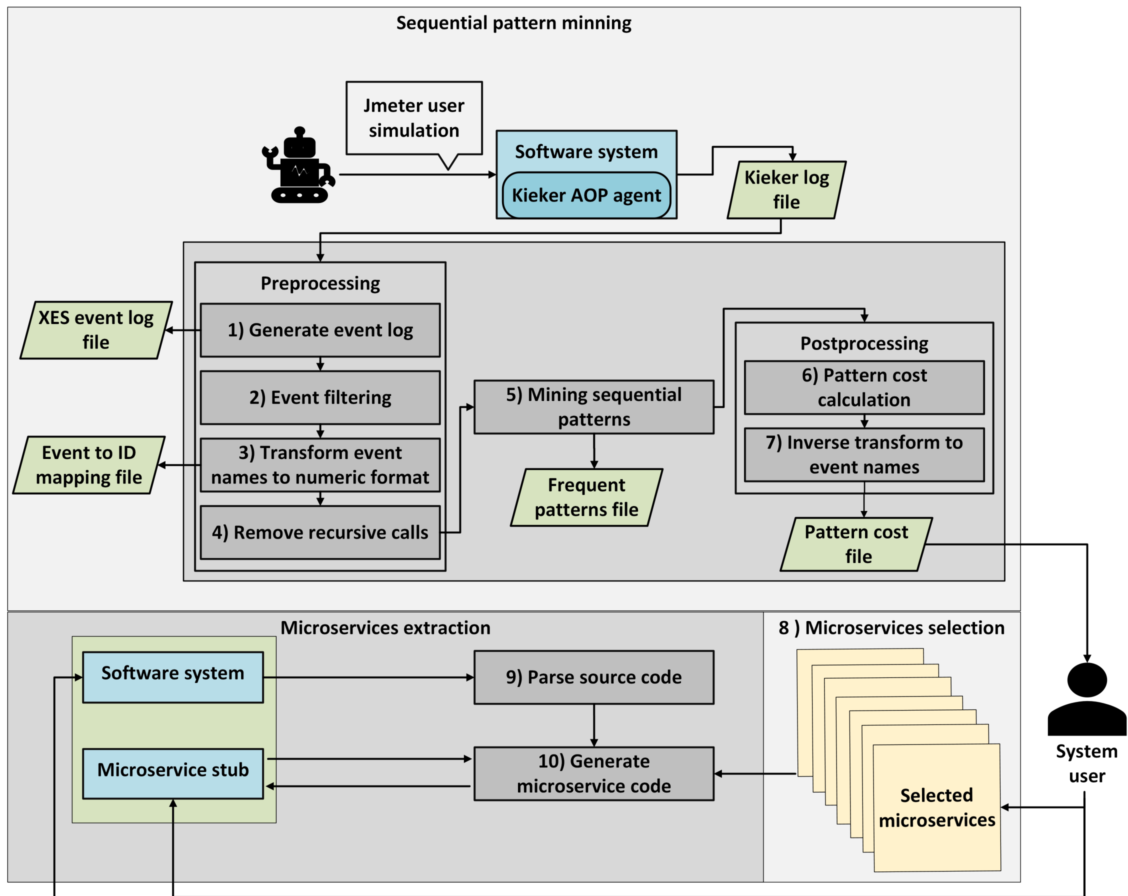


Figure 4.1: Overview of the *MIST* framework.

monitoring by integrating a third-party dependency into the system to capture detailed execution information. Accordingly, the *MIST* framework begins by instrumenting the target software system using the Kieker monitoring tool[1], which has been widely adopted in dynamic software analysis [9, 6, 18](refer Section 2.3).

The effectiveness of dynamic analysis depends on capturing a comprehensive range of user behaviors. To improve reproducibility, particularly during evaluation, it is important to replay the same user behavior consistently. Therefore, automated user simulation becomes essential. JMeter[2], an open-source load-testing tool, is used to simulate user behavior. Predefined test cases simulate user behavior, covering all users and system functions. As illustrated in Fig. 4.1, the execution of these use cases on the instrumented system produces

[1] `https://kieker-monitoring.net/`
[2] `https://jmeter.apache.org/`

a Kieker log file, which serves as the input for subsequent processing in the pattern mining and microservice identification pipeline.

Listing 4.1 presents an excerpt of a Kieker log file capturing information on three method invocations, one per log line. Using the semicolumn symbol as a delimiter, each line captures Kieker log type, unique entry ID, invoked method, execution trace ID, start timestamp of method execution, end timestamp of method execution, computation node ID, order of the method invocation in the system execution, and depth of the method invocation in the call stack. This detailed, method-level execution data is essential for the operation of the *MIST* framework. The generated Kieker log file is then fed into the *MIST* tool, which implements the framework. The tool executes in three stages: sequential pattern mining, microservices selection, and microservice extraction. Sequential pattern mining is a seven-step pipeline to identify frequent patterns with associated costs, as depicted in Fig. 4.1.

```
1 $1;1733116555379401700;public static java.lang.String net.jforum.util.preferences.
      SystemGlobals.getValue(java.lang.String);2140194985419472908;1733116555379026700;
      1733116555379396200;4180L-212561-W;57;3
2 $1;1733116555379053100;public java.lang.String net.jforum.util.preferences.SystemGlobals
      .getVariableValue(java.lang.String);2140194985419472908;1733116555379029900;
      1733116555379052400;4180L-212561-W;58;4
3 $1;1733116555379039200;public java.lang.String net.jforum.util.preferences.
      VariableExpander.expandVariables(java.lang.String)
      ;2140194985419472908;1733116555379034100; 1733116555379036300;4180L-212561-W;59;5
```

Listing 4.1: Excerpt of a Kieker log.

4.2.1.1 Preprocessing

The Kieker log file is fed into the *MIST* tool, which begins processing with a four-step preprocessing pipeline that transforms the raw log into a filtered, structured, and optimized event log suitable for sequential pattern mining. *MIST* tool is the software implementation of the *MIST* framework.

Step 1 - Generate event log: The programmatic implementation of the *MIST* framework starts by converting the integrated Kieker logs to event tuples defined in Section 2.4. Each method invocation in the Kieker log is converted to an event. An event log file is generated as an intermediate output, in the XES[3] format, the de facto standard for event data in process

[3] `https://www.xes-standard.org/`

mining. As discussed in Section 2.4, these XES logs typically serve as primary inputs for a wide range of mining applications. This supports interoperability, allowing seamless integration with established tools such as ProM[4] and Disco[5], both of which natively support the XES format. Although the generated events are used in subsequent steps, the XES file itself is not used in the *MIST* framework; it is produced solely to support future research on applying process mining techniques to microservice identification. Listing 4.2 illustrates an example event in XES format, generated by *MIST* from the first line of the Kieker log shown in Listing 4.1.

```
1  <trace>
2  <string key='traceID' value='2140194985419472908'/>
3  ...
4  <event>
5  <string key='concept:name' value='net.jforum.util.preferences.
       SystemGlobals.getValue(java.lang.String)'/>
6  <string key='lifecycle:transition' value='complete'/>
7  <date key='time:timestamp' value='2024-12-02T17:15:55.26700'/>
8  <string key='eventId' value='1733116555379401700'/>
9  <int key='duration' value='369500.0'/>
10 <int key='callingOrder' value='57'/>
11 <int key='depth' value='3'/>
12 </event>
13 ...
14 </trace>
```

Listing 4.2: Example event (XES format).

Step 2 - Event Filtering: Software systems often include frequently executed methods that are not meaningful for microservice identification, including initialization routines, routing mechanisms, and communication handlers. These methods introduce noise into the pattern mining process, potentially leading to spurious patterns. Event filtering removes these methods from the event log using a configurable exclusion list, allowing to specify packages, classes, or individual methods to be excluded from the analysis.

Step 3 - Transform event names to numeric format: Method signatures in the Kieker and event tuples are stored as strings. For large systems and extensive test suites, the volume and variety of distinct string entries create a significant performance impact on pattern mining, as repeated string comparisons are computationally expensive. To mitigate

[4] https://promtools.org/
[5] https://fluxicon.com/disco/

this overhead, each unique method signature is mapped to a distinct integer identifier. All subsequent processing operates on compact numerical arrays rather than string data, reducing memory consumption and accelerating look-ups, joins, and aggregations. The mapping from string method signature to numerical value is stored in a temporary data store, which is used in the postprocessing phase to restore the original method names.

Step 4 - Remove recursive calls: Execution traces often contain repeated occurrences of the same method due to the recursive calls. Although recursion increases the trace length, each method counts only once per trace; thus, it does not affect the method call frequency (support). However, retaining recursive calls adds unnecessary complexity and reduces mining efficiency. Therefore, this step removes recursive method calls from all the traces to enable more effective sequential pattern mining, which focuses on non-recursive execution behavior. The ability to invoke methods recursively is preserved in the extracted microservices and can still be executed at runtime.

4.2.2 Sequential Pattern Mining

Step 5 - Mining sequential patterns: The preprocessed event log is passed to Step 5, which applies sequential pattern mining to discover frequent method invocation sequences across all traces. *MIST* uses the PrefixSpan algorithm from the open-source SPMF library[6] [175], a pattern growth-based approach that avoids explicit candidate generation by recursively projecting the FP-tree structure as discussed in Section 2.4. Hence, it is efficient for long sequences and large event logs [69]. The SPMF library provides over 250 data mining algorithms, and *MIST* adopts a pluggable architecture that enables substitution of alternative mining algorithms as needed. SPMF library requires a specific input data file. Hence, this step includes preparing input files for the SPMF library, executing the mining process, and producing the output. The mining process is configured with a minimum support threshold σ_{min}, which controls the minimum amount of traces in which a pattern should appear in order to be a frequent pattern. A frequent pattern, together with its support values in the numerical encoding established in Step 3, is produced as the output of this step. Listing 4.3 illustrates a sample of the pattern mining output. -1 value in the sample output acts as the delimiter between pattern and support values.

[6] `https://www.philippe-fournier-viger.com/spmf/`

```
1 6 -1 #SUP: 500
2 6 132 -1 #SUP: 150
3 6 132 216 -1 #SUP: 145
4 6 40 -1 #SUP: 165
5 6 40 41 -1 #SUP: 165
6 6 41 -1 #SUP: 178
7 6 41 42 -1 #SUP: 178
8 6 42 -1 #SUP: 196
9 6 42 43 -1 #SUP: 173
10 6 42 59 -1 #SUP: 151
11 6 42 44 -1 #SUP: 173
12 6 42 79 -1 #SUP: 147
```

Listing 4.3: Pattern mining output.

4.2.3 Pattern Cost Calculation and Postprocessing

Step 6 - Pattern cost calculation: Sequential pattern mining (step 5) produces a large volume of frequent patterns. However, all patterns are equally qualified for extraction as microservices. For example, a pattern with high support but very short length may not have a meaningful functional scope. Hence, a ranking method is required to prioritize patterns that are structurally coherent and operationally significant. *MIST* evaluates each mined pattern against five properties: support(S), confidence(C), pattern length(P), average depth(D), and average execution time(E), combining them into a unified pattern cost score. Each property is discussed below, followed by the normalization procedure and the cost formula.

Support: Support measures the frequency of a pattern across the event log and is formally defined in Section 2.4. It is computed directly by the SPMF library during the execution of the PrefixSpan algorithm and is available in the pattern mining output. A pattern with high support appears consistently across a large portion of system executions, indicating that the corresponding method-invocation sequence is a stable and recurring unit of system behavior. Hence, patterns with higher support values are given greater priority in the pattern cost calculation, as they are more likely to represent generalizable microservice candidates that operate reliably across the full range of system use cases.

Confidence: Confidence measures the strength of association among the method calls within a pattern, representing the conditional probability that the complete sequence occurs

without interruption given that its prefix has been observed. It is formally defined in the Section 2.4 and computed using the support values available in the pattern mining library output. For example, the confidence of pattern (a, b) is computed as:

$$Conf((a, b) = \frac{Sup(a, b)}{Sup(a)}.$$

Since (a, b) is a frequent pattern, its prefix (a) must be frequent, and it appears in the mining output, as illustrated in Listing 4.3. Higher confidence indicates that the methods in the pattern are tightly coupled in execution and consistently co-occur in the same order, which is a desirable property for the microservices boundary. Patterns with higher confidence are assigned a greater cost.

Pattern Length: Pattern length corresponds to the number of method invocations in the sequence, as defined in Section 2.4. Longer patterns encapsulate more method invocations within a single cohesive execution sequence, indicating a richer and more functionally complete unit of behavior. A very short pattern, such as a one method sequence, may represent an overly fine-grained decomposition that introduces excessive inter-service communication overhead when extracted as a microservice. Longer patterns are prioritized in the cost calculation, as they are more likely to yield a microservice with sufficient functional scope and internal cohesion for independent deployment.

Average Depth: Average depth captures the mean call-stack depth of the patterns across all its occurrences in the event log, formally defined in Section 2.4. The call stack depth is recorded in the Kieker log and mapped to the depth field δ of each event during preprocessing. Methods invoked at deeper levels of the call stack tend to represent finer-grained, core implementations of functionalities that are encapsulated and less dependent on the broader system context. Patterns whose constituent methods consistently appear at greater depths are therefore considered more suitable microservice candidates and assigned a higher cost.

Average Execution Time: Average execution time reflects the mean total runtime of all method invocations within a pattern across all its occurrences, as formally defined in Section 2.4. The duration of each method invocation is derived from the entry and exit

timestamps recorded in the Kieker log, as shown in Listing 4.1, and mapped to the duration field d of each event. Patterns with higher average execution time represent performance-critical behavior where a sequence of method invocations consumes a significant portion of process time. Extracting such patterns as microservices has a higher potential to improve the overall throughput and resource utilization of the migrated system. Accordingly, patterns with higher execution time are prioritized in cost calculation.

Normalization and Cost Formula: The five properties are measured on different scales. Confidence is in the range [0.1], while support, pattern length, average depth, and average execution time are positive values where the magnitude of the values can vary depending on the system under analysis. To ensure fair assessment and prevent a single property from dominating due to scale differences, all five properties are normalized to the range [0,1] prior to cost calculation using min-max normalization. Since confidence is already in this range, normalization does not apply to confidence.

The overall cost of each pattern is then computed as the product of its five normalized property values:

$$\text{PatternCost} = \text{S} \times \text{C} \times \text{P} \times \text{D} \times \text{E} \tag{4.1}$$

The multiplication of properties is chosen to provide equal prioritization. Unlike a weighted sum, which allows a pattern scoring zero on one property to compensate through high scores on other properties, a product-based formula ensures that a pattern scoring zero or near zero on any single property receives a correspondingly low value. This ensures that microservices satisfy all five criteria: frequent, tightly associated, sufficiently long, appropriately deep, and computationally intensive. A pattern that fails on any one dimension is unlikely to yield a high-quality microservice regardless of its performance on the others. The resulting pattern cost values are used to rank all discovered patterns in descending order, producing the ranked pattern cost table as shown in Table 4.1, which serves as the input to the microservice extraction stage.

Step 7 - Inverse transform to event names: The numerical patterns are translated back to their original method signatures using the event-to-ID mapping store produced in Step 3. The final output of phase one is a pattern cost file containing each discovered pattern expressed as an ordered sequence of fully qualified method names, ranked in descending

Table 4.1: Sample output of pattern and cost.

Pattern	Cost
public static java.lang.String net.jforum.util.preferences.SystemGlobals.getValue(java.lang.String) , public java.lang.String net.jforum.util.preferences.SystemGlobals.getVariableValue(java.lang.String) , public java.lang.String net.jforum.util.preferences.VariableExpander.expandVariables(java.lang.String) ,	1
public static boolean net.jforum.repository.ForumRepository.isCategoryAccessible(int) , public static boolean net.jforum.repository.ForumRepository.isCategoryAccessible(int, int) ,	1
public java.lang.String net.jforum.util.preferences.SystemGlobals.getVariableValue(java.lang.String) , public void net.jforum.csrf.CsrfLogger.log(org.owasp.csrfguard.log.LogLevel, java.lang.String) ,	0.989
public static int net.jforum.util.preferences.SystemGlobals.getIntValue(java.lang.String) , public static int net.jforum.repository.ForumRepository.getTotalMessages() , public static java.util.List net.jforum.repository.ForumRepository.getAllCategories(int) ,	0.988

order of pattern cost. A sample output table is provided in Table 4.1. This file serves as the primary input to the microservice extraction stage described in Section 4.2.4.

4.2.4 Microservice Selection and Code Generation

The *MIST* framework spans the identification, development, deployment, and performance measurement phases. Hence, the microservices selection and implementation procedure is required. This stage translates the ranked patterns produced by the automated *MIST* pipeline into deployable microservice code. This stage requires manual operations: pattern selection, stub creation, and code verification, making microservice extraction a semi-automated process. Hence, expert system user operations are required for this stage as illustrated in Figure 4.1.

The pattern cost file produced in Step 7 contains the patterns ranked in descending order of cost, along with their fully qualified method names. The system user inspects this table to select the pattern to be extracted as microservices.

Step 8 - Microservices Selection: This step remains a manual task due to the infeasibility of automating domain-driven design boundaries selection, the feasibility of extraction, avoiding functional overlaps, and operational level conditions. The system user examines the highest cost patterns in the cost table, assessing each candidate against these selection criteria and selecting those that represent meaningful, self-contained functional units that can be clearly separated from the monolithic source code. Overlapping patterns and non-consecutive patterns are excluded from selection. For instance, methods A, B, C, and E generate two frequent sequences: A, B, C with a cost of 0.75, and A, C, E with a cost of

0.85. Despite A, C, E having the highest cost, the sequence A, B, C was chosen for the experiment due to its consecutive nature. The output of this step is selected patterns, each defining the method-invocation sequence to be extracted as an independent microservice. The manual selection of patterns is a current limitation of *MIST* and represents a direction for future work, where machine learning-based recommendations or a cost thresholds-based approach could replace the manual inspection.

Step 9 - Parsing Source Code: This step facilitates the extraction of microservices by analyzing and parsing the monolithic source code. Since *MIST* provides a sequence of method invocations, locating the method-level details of the monolithic source is essential. This is achieved by using JavaParser [7], an open-source Java library that parses Java source code into an Abstract Syntax Tree(AST). It allows programmatic analysis, access, and manipulation of code structure in plain source code rather than compiled code, making it well-suited for method-level extraction required by the *MIST* framework. Hence, this step generates the AST of the source code by parsing all the Java source files as compilation units, which contain the package declaration of the class, import statements, class declarations, field declarations, and method declarations.

Step 10 - Microservice Code Generation: This step is responsible for generating the microservice code. A microservice stub containing the initial code structure for dependency management and deployment is provided as input to this step, along with the user-identified microservices. This step iterates through the methods in the microservices, locates the source code from the parsed AST in step 9, and writes the method to the microservice stub, including package declarations, class imports, and the complete method body.

The final outputs of this stage are the microservice stub with relevant microservices methods and the monolithic software system. This step requires system user intervention, as the communication between monolithic system and microservice stub have not been automated due to the high complexity of automatically updating source files in the monolithic system. To preserve the operational continuity of the original system, request processing is intercepted and forwarded to the microservices. Once the operation is completed, the response is routed back to the monolithic system. The microservice stub is packaged for independent deployment and can be scaled, updated, and managed separately from

[7] https://javaparser.org/

the monolithic system. Hence, it is realizing the operational benefits of microservices architecture.

The monolithic and microservice sample implementations are illustrated in Listing 4.4 and Listing 4.5, respectively, where Listing 4.4 shows the request sent from the monolithic system. Listing 4.5 illustrates the capture, processing, and return of the response to the monolithic system using RESTful web services. This is a sample implementation for the first pattern in Table 4.1

```
1
2 public class SystemGlobals{
3     ....
4     public static String getValue(String field) {
5             MSSenderService msSenderService = new MSSenderService();
6             String response = msSenderService.sendSystemGlobalMessage(field);
7             return response;
8         }
9     ....
10    }
11
12 public class MSSenderService {
13   public String sendSystemGlobalMessage(String field) {
14     String url=String.join(ip, port, '/service/api/receiveMessage');
15     try {
16       RestTemplate rest = new RestTemplate();
17       Gson gson = new Gson();
18       ServiceMessageEntity msg = new ServiceMessageEntity();
19       msg.setMessageType(MSMessageTypes.SYSTEM_GLOBAL);
20       msg.setField(field);
21       URI uri = new URI(url);
22       return rest.postForObject(uri, gson.toJson(msg), String.class);
23     } catch (URISyntaxException e) {
24       System.out.println('Error: ' + e.getMessage());
25     }
26     return '';
27   }
28 }
```

Listing 4.4: Sample request sending from monolithic system.

4.3 Evaluation of Microservices

This section evaluates *MIST* the two aspects that together determine the quality of a microservice migration: performance and structural quality. Performance evaluation ex-

```
1  @RestController
2  @RequestMapping(/api)
3  public class MSReceiverService {
4
5    @RequestMapping(/receiveMessage)
6    public String receiveMessage(@RequestBody String payload) {
7      Gson gson = new Gson();
8      ServiceMessageEntity serviceMessageEntity = gson.fromJson(postPayload, ServiceMessageEntity.class)
           ;
9      switch (serviceMessageEntity.getMessageType()) {
10         ....
11          case SYSTEM_GLOBAL:
12          String val = SystemGlobals.getValue(serviceMessageEntity.getField());
13          return val;
14         ....
15     }
16   }
17 }
18
19 public class SystemGlobals {
20   public static String getValue(String field) {
21     {
22         return globals.getVariableValue(field);
23     }
24   public static String getVariableValueServiceCall(String field)
25     {
26         return globals.getVariableValue(field);
27     }
28   public String getVariableValue(String field)
29     {
30         String preExpansion = globals.installation.getProperty(field);
31         if (preExpansion == null) {
32             preExpansion = this.defaults.getProperty(field);
33             if (preExpansion == null) {
34                 if (LOGGER.isEnabledFor(Level.INFO)) {
35                         LOGGER.info(Key ' + field + ' is not found in  + globals.defaultConfig +  and  +
                              globals.installationConfig);
36                 }
37                 return null;
38             }
39         }
40         return expander.expandVariables(preExpansion);
41     }
42 }
```

Listing 4.5: Sample microservice code.

amines whether the migration produced by *MIST* preserves or improves the operational characteristics of the system, specifically throughput and latency, relative to the monolithic

Table 4.2: Summary of evaluated applications.

Application	Description	Version	LoC	Classes	Test cases	Requests	Events
JForum	Online discussion forum	2.8.3	32,171	353	48	635	24,999
Apache Roller	Multi user blog server	5.2.0	53,889	615	92	512	485,379

baseline. Structural quality assesses whether the identified microservices exhibit the architectural properties considered desirable in the microservices literature, such as modularity, cohesion, and coupling. Evaluating both dimensions is essential because structural soundness alone does not guarantee operational quality, and performance improvement alone does not guarantee operational viability.

Two open-source Java applications were selected for evaluation to represent a range of system sizes, domain types, and code complexity, as summarized in Table 4.2. The 'Requests' column reports the number of simulated user requests used to generate the execution traces, while the 'Events' column shows the number of events produced by the *MIST* framework from these simulated requests. All tested applications are Java-based implementations and provide sufficient scale for pattern mining. As illustrated in Section 3.1, these applications are widely used in existing dynamic-analysis-based microservice migration studies. The applications were selected to represent a range of system sizes in Lines of Code (LoC), ranging from medium (32,171 LoC), to large (53,889 LoC), to reflect realistic legacy system scales.

4.3.1 Performance Impact of Sequential Patterns on Microservices

The performance evaluation is guided by the following questions:

- EQ1) Do microservices identified by *MIST* achieve comparable or higher throughput and latency relative to the monolithic baseline?
- EQ2) Which attributes individually and collectively produce the greatest performance improvement?
- eQ3) Is there a consistent relationship between specific pattern properties and performance outcomes?

To address these questions, we enhanced the output of the first stage of the *MIST* framework to provide sequential patterns for all non-empty subsets of five properties:

support (S), confidence (C), pattern length (P), average depth (D), and average execution time (E). This yields $2^5 - 1 = 31$ unique combinations. For each subset of attributes, the *MIST* framework generates a corresponding list of frequent patterns, where the cost of each pattern is calculated as the product of the normalized values of the selected attributes. For instance, the “S” list ranks patterns solely by normalized support values, while the “C” list uses confidence values. The “S,C” list, in turn, ranks patterns using the product of support and confidence. The final list, labeled ”S,C,P,D,E”, uses all five properties and computes the cost as defined in Equation (4.1). As shown in Table 4.1, 31 similar lists with associated patterns for each list are provided as the pattern output for this evaluation. The three highest-cost patterns with consecutive methods were selected as microservices for each testing scenario as discussed in Section 4.2. Three microservices were selected for each category to reflect the structure of real-world applications, which often consist of multiple microservices.

The test setup consists of four servers: one dedicated to the monolithic portion of the system and the remaining three serving the microservices. The testing was conducted in a cloud environment with four Ubuntu servers and one MySQL database instance. Each server contains 4 VCPUs, 16GB of RAM, and 30GB of hard drive. The database instance has 4GB RAM and 4GB of hard drive. Cloud deployment is used over local servers to provide the standard microservices deployment environment.

The cost of cloud utilization depends on resource utilization, and *MIST* is designed to identify microservices suitable for edge and IOT environments. Evaluating under stressed CPU conditions provides a realistic simulation of resource-constrained deployment conditions. depends on resource utilization, and *MIST* is designed to identify microservices suitable for edge and IOT environments. Hence, to simulate resource-constrained environments of edge and IOT devices, the CPU is stressed to 100% using the Ubuntu stress tool.

JMeter was employed to simulate the system load during the experiments. The JForum test suite comprised six user simulations: two admin users and four application users. By manipulating the thread count and loop properties in JMeter, the desired load was replicated across 25 threads per user during the evaluation, resulting in a total of 150 concurrent threads. The Apache Roller test suite includes one admin user and one general user, each with 100 threads, for a total of 200 simultaneous threads. These values were derived from empirical evaluations of these two systems in resource-constrained cloud

environments where systems are overloaded.

Two performance metrics were evaluated: throughput, the number of requests processed per second, and latency, the time interval between sending the request to the server and receiving the response. Hence, higher throughput and lower latencies indicate better performance. To enhance the reliability of the results, the experiments were repeated three times, and the average value for each metric was calculated. In the experiment results, the monolith system is represented by 'M', the attributes S, C, P, D, E, and their combinations represent the properties used in the cost formula for each test scenario.

The throughput of the monolithic JForum and Apache Roller applications, together with the throughput of the 31 corresponding microservice configurations, is presented in Figures 4.2 and 4.3. Higher throughput indicates better results. The results show that microservices identified using the higher-average-depth attribute alone achieve higher throughput than the monolithic system. Furthermore, combining confidence with average depth yields performance improvements. The use of pattern length and average execution time has the potential to enhance system performance. In contrast, selecting microservices solely based on the Support property consistently leads to a decline in throughput, indicating reduced system performance. In addition, the results indicate that increasing the number of properties used for microservice identification can reduce system throughput.

Figures 4.4 and 4.5 present the latency of the monolithic JForum and Apache Roller applications relative to the 31 test scenarios. Lower latency indicates better results. The results show that the average depth attribute consistently reduces latency compared to the monolithic system, thereby enhancing performance. In contrast, confidence, pattern length, and average execution time exhibit inconsistent effects on latency, while the support consistently increases latency, resulting in reduced performance.

Overall, the latency and throughput results confirm that the average depth property consistently improves system performance, whereas support and its combinations lead to a substantial performance decline. The pattern length, average execution time, and confidence properties exhibit variable behavior, with the potential to improve performance.

These results can be interpreted in terms of the operational characteristics. The average depth property consistently improves performance because deep call-stack patterns are infrequent compared to high-level method invocations, resulting in low execution overhead as a microservice. In contrast, support-based selections identify patterns that appear frequently across traces, caused by frequent remote invocations. Thereby, increasing the

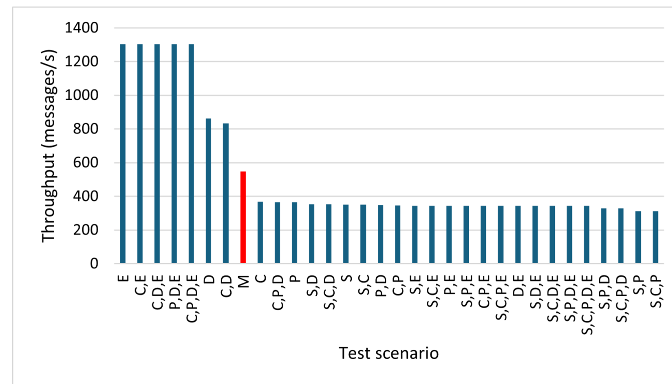


Figure 4.2: JForum throughput analysis.

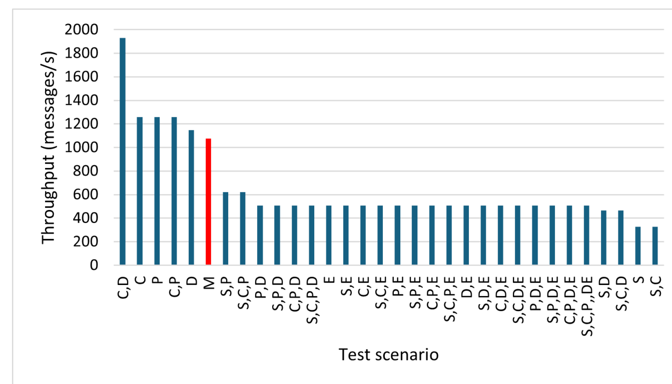


Figure 4.3: Apache Roller throughput analysis.

communication overhead. The variable behavior of pattern length, confidence, and average execution time suggests that these properties influence the performance indirectly.

In answering the first evaluation question, EQ1, the results demonstrate that microservices identified by *MIST* can achieve superior throughput and lower latency than the monolithic baseline, provided that appropriate cost properties are selected. Configurations with average depth consistently outperform the monolith across both JForum and Apache Roller applications, confirming that *MIST* can produce microservices that improve operational performance.

In response to EQ2, average depth is identified as the single most effective individual property for performance improvement, consistently improving both throughput and latency across both systems. Support is identified as negatively impacting property when used in isolation or in combination with other properties. Among the combined configurations, confidence and depth yield the most reliable performance improvements.

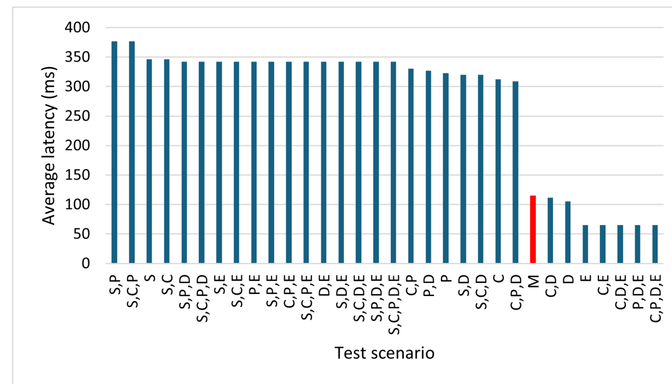


Figure 4.4: JForum latency analysis.

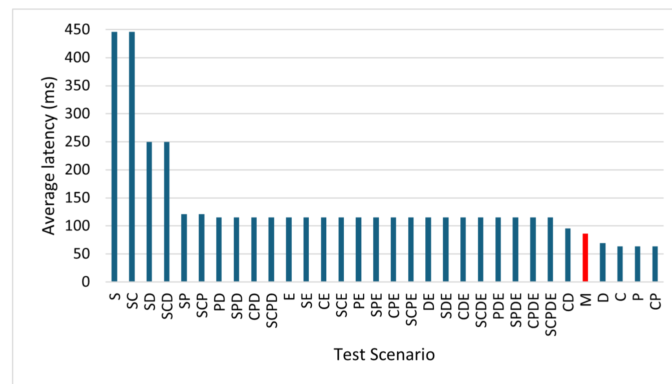


Figure 4.5: Apache Roller latency analysis.

In response to EQ3, the relationship between average depth and performance improvement is consistent across both JForum and ApacheRoller, suggesting that this finding generalizes beyond a single system. The negative effects of support and combinations are similarly consistent across both systems. In contrast, the effects of pattern length, confidence, and average execution time vary across systems, indicating that these properties depend on system-specific characteristics.

4.3.2 Structural Quality Evaluation

Assessing structural properties is the general approach to evaluating the quality of microservices identification studies, focusing on the architectural soundness of the identified microservices and their coupling, cohesion, and modularity properties. Four metrics drawn from established microservice evaluation benchmarks are used as measures: Struc-

Table 4.3: Structural property evaluation; **bold** - best value, underline - second best value.

System	Property	*MIST*	FoSCI [6]	FOME [18]	MEM [16]	Micro Miner [53]	DRS [106]
Apache Roller	SM(+)	**0.644**	0.319		0.056		
	CHM(+)	**1.000**	0.833	0.700	0.779		0.760
	CHD(+)	**1.000**	0.682	0.900	0.385		0.53
	IFN(-)	**1.000**	1.288	1.750	15.000		
JForum	SM(+)	0.187	**0.435**		0.0359	0.040	
	CHM(+)	**1.000**	0.771	0.800	0.508	0.670	0.730
	CHD(+)	**1.000**	0.486	0.800	0.152	0.666	0.520
	IFN(-)	**1.000**	2.816	2.555	27.000	2.800	

tural Modularity (SM), Interface Number (IFN), Cohesion at Message Level (CHM), and Cohesion at Domain Level (CHD) [6, 51, 19, 53, 18].

For structural quality validation, microservices are identified using the cost formula by combining *S, C, P, D, E* as given in Eq. (4.1), rather than the 31 combinations used for performance evaluation. The structural quality evaluation assesses the microservices produced by the *MIST* framework and determines whether they exhibit higher values. The full cost formula, including all five properties, represents the complete configuration of the *MIST* framework. Evaluating structural properties across 31 combinations assesses partial and incomplete configurations. Hence, the cost formula with all five attributes is used for the validation. Table 4.3 illustrates the structural properties of candidate microservices. The best-case results reported in existing benchmark approaches were used for this comparison. In general, *MIST* achieves the best or second-best metric values, demonstrating that the microservices it identifies exhibit higher cohesion, lower coupling, and stronger structural modularity. Since the microservices are in a consecutive-sequential pattern, each microservice exposes only a single interface, resulting in an IFN value of one. Moreover, these independent sequences show no interaction, resulting in CHM and CHD values of one.

4.4 Implementation and Reproducibility

The *MIST* framework is implemented as a fully configurable Java Spring Boot application and is publicly available at `https://github.com/thakshilad/MIST`. The framework implementation contains the pipeline described in Section 4.2. The Spring Boot framework was chosen deliberately due to its convention-over-configuration approach, which enables all the pipeline parameters to be configured through an external *application.properties* file. Regarding the system requirements, the framework is compatible with Java 17 or higher versions and requires the SPMF pattern mining library. The *MIST* framework contains two components: the sequential pattern mining and code rewriting.

4.4.1 Instrumentation and Input Configuration

The sequential pattern mining component is responsible for mining sequential patterns from execution traces. It is structured around the seven stages that correspond directly to the pipeline steps described in Section 4.2. Each stage reads its input from the outputs of the preceding stage, producing multiple intermediate output files during the pipeline execution. The pattern mining component accepts Kieker log files as its primary input. The location for the Kiker log files requires specifying through the *app.kiekerLogFileLocation* property in the configuration file, which locates the folder containing all the Kieker log files generated by the instrumented application. The Kieker log files collected from JPetStore, JForum, and Apache Roller are provided in the public repository. Event filtering is implemented as a configurable exclusion mechanism specified through *app.StringToAvoidFromPatternMatching* property, which accepts a comma-separated list of string patterns. This design reflects the configurable nature of Step 2 described in Section 4.2, allowing domain-specific filtering decisions to be captured as explicit, reproducible configuration rather than hard-coded logic.

4.4.2 Plugable Pattern Mining Algorithms

The *MIST* framework implements a pluggable architecture in which the pattern mining algorithm is specified as a configuration parameter through the *app.PatternMiningAlgorithm* property. The default and recommended algorithm is PrefixSpan, provided through the SPMF library [175]. The SPMF library provides over 250 data mining algorithms, and

any sequential pattern mining algorithm available within the library can be substituted by updating this single configuration parameter without modifying the source code.

The minimum support threshold is specified as a percentage of the total number of traces through the *app.patternMiningMinimumSupport* property. This parameter controls the minimum support of the traces in which the pattern must appear to be considered frequent, with the default value of 25%. The SPMF JAR library is provided using the *app.patternMinningLibrary* property, which specifies the reference to the folder location.

4.4.3 Pattern Mining Output Files

The primary output of the sequential pattern mining component of the *MIST* framework is *patternOutput.xlsx* Excel workbook, which constitutes the complete dataset for the performance evaluation reported in Section 4.3. This workbook contains 31 worksheets – one for each non-empty subset of the five cost properties, corresponding to the $2^5 - 1 = 31$ attribute combinations evaluated in the performance analysis. Each worksheet presents all discovered patterns for the corresponding attribute combination, expressed as an ordered sequence of fully qualified method names, and ranked in descending order of their PatternCost value under that combination. Furthermore, intermediate output files of the pipeline, including event XES event logs, event to ID mapping, and sequential pattern mining library outputs, are generated to enhance the verification of the pipeline. All output file locations are configurable through the application properties file.

4.4.4 Code Rewriting Component

The code rewriting component is implemented separately to enhance interoperability, without depending on the pipeline steps in the sequential pattern mining component. It is based on JavaParser [176], which parses the source code into an abstract syntax tree. Everything in the source code is considered a node in the abstract syntax tree. A node is represented as a compilation unit. The respective methods of a particular microservice were identified by traversing the tree and extracting them in the microservice stub using lexical preservation printing in the JavaParser library. All configurations, including the extraction method list, the monolithic source code location, and the microservice stub location, are configurable via the application properties file.

4.4.5 Reproducibility

The experiments reported in this chapter are fully replicable, and all resources required to reproduce them are publicly accessible at `https://github.com/thakshilad/MIST`. The repository is organized into six folders whose content collectively covers all stages of the experimental pipeline. *MIST tool* contains the source code implementation, *JmeterTestSuits* contain the JMeter test scripts used to simulate user load during both execution trace collection and performance evaluation. *PatternMiningLibrary* contains the SPMF library JAR used for sequential pattern mining. *Inputs* folder contains the Kieker log files collected from JPetStore, JForum, and Apache Roller during the JMeter simulated test executions. *Outputs* contains the intermediate output files at each pipeline stage for each experimental scenario. *Results* provides the complete experimental results used to produce the evaluation figures and tables reported in this chapter.

4.5 Summary

This chapter proposed *MIST*, a semi-automated framework for identifying microservices and evaluating how sequential patterns and related attributes affect their performance. By examining low-level, repetitive, and consistent method invocation sequences, *MIST* allows for the systematic breakdown of monolithic applications into independent executable microservices. These services can be deployed separately while maintaining communication with other system components, thereby ensuring operational continuity during migration.

The experimental evaluation confirms that sequential patterns and related attributes can effectively support the identification of microservices from legacy systems. Among the five analyzed factors (support, confidence, pattern length, average depth, and average execution time), microservice identification based on the support attribute reduces system performance, while average depth can improve it. Confidence, pattern length, and average execution time have the potential to improve the system performance. *MIST* exhibits high structural quality, achieving high cohesion and low coupling. This indicates that *MIST* effectively produces small, highly cohesive, and loosely coupled service boundaries, essential to enhance system functional independence, scalability, maintainability, and flexibility. Our evaluation results demonstrate that complex, tightly coupled monolithic systems can successfully migrate to microservices-based architectures, achieving performance

improvements by leveraging sequential execution traces to identify meaningful, highly cohesive, and loosely coupled services.

Microservices operate in highly dynamic environments where continuous monitoring, failure detection, and recovery mechanisms are essential. Enhancing the microservice architecture to respond effectively to changing environmental conditions is crucial to reduce the complexities in monitoring and maintenance. Self-adaptive systems address these challenges in the operational environment by incorporating capabilities such as self-configuration, self-healing, self-optimization, and self-protection into software systems. Enabling self-adaptivity in microservice architecture is explored in the next chapter.

Chapter 5

Runtime Data Driven Adaptation for Microservice Systems

Modern distributed and microservice-based systems operate in highly dynamic environments, making monitoring, troubleshooting, and maintenance complex, time-consuming, and error-prone. To address this, **Service Colony** is proposed as a novel architectural style for developing software systems as a self-adaptive microservice-based system. A service colony organizes a system as a group of autonomous software services that cooperate to achieve its objectives. Each service in the colony is responsible for implementing a specific functionality, interacting with other services and system users, and making proactive decisions that influence its and the system's performance, as well as the system's topology and interaction patterns. By enhancing the self-awareness and autonomy of individual components, this architecture fosters greater decentralization, flexibility, modularity, and robustness. Experiments involving the reengineering of an open-source system into a service colony demonstrate its ability to dynamically rearchitect and adjust performance in response to changing user demands and usage intensity.

This chapter is derived from:

Thakshila Imiya Mohottige, Artem Polyvyanyy, Colin Fidge, Rajkumar Buyya, Alistair Barros, Service Colonies: A Novel Architectural Style for Developing Software Systems with Autonomous and Cooperative Services, *ACM TOSEM Frontiers of Software Engineering(Second Review), April 2026.*

5.1 Self-Adaptive Software Systems

The software system architecture defines its structure and behavior within its operating environment. Software system architectures evolve to meet changing customer demands. Enterprise application architectures have evolved from legacy mainframe-based software systems and SOA to microservice architectures (MSA) [2, 3, 39]. Cloud-based technologies accelerate this transition by capitalizing on their scalability, flexibility, security, cost-effectiveness, and monitoring capabilities. However, the challenges of modern software systems concern the increasing complexity of managing distributed architectures in highly dynamic environments. Despite changes occurring in its environment, a software system must continue to operate reliably and efficiently, delivering its functionalities to its users. Therefore, it is desirable to allow systems to adjust their structures and behaviors at runtime based on environmental changes to enhance transparency, elasticity, resilience, and deployment management [32].

Due to substantial investments already made in software systems, manually migrating them to new architectural styles is often infeasible. Automated enhancement of existing systems to meet current and future requirements is a more reasonable and cost-effective alternative. However, ensuring continuous operation under varying conditions often requires extensive human testing and supervision, leading to high costs for application configuration, troubleshooting, and maintenance [177]. Hence, there is an increased demand to automate the monitoring and management of distributed systems to reduce costs while ensuring their robustness and quality [178].

Self-adaptive software systems emerged to reduce uncertainty and complexity in dynamic operating environments [177], with the requirements to enhance the reliability, availability, flexibility, and performance of the system [5]. As discussed in Section 2.5, the self-adaptive autonomous software system monitors its internal and external states, identifying when and how to reconfigure to manage new conditions and dynamically adapt the software architecture at runtime. Developing a self-adaptive software system is challenging due to conflicting system goals, the need to monitor different quality of service (QoS) properties, and handling complex adaptation mechanisms and their effects [136]. Once developed, self-adaptive systems can perform self-configuring, self-healing, self-optimizing, and self-protecting functions [29, 149].

With the increasing demand for (micro)service-based systems, the introduction of

self-adaptive and autonomous properties has gained attention. Multiple studies have addressed self-adaptivity in service-based systems. However, as discussed in Section 3.2, there is a gap in existing studies: none address decentralized decision-making, proactive, learning-based, application-layer operations, agent-based split-merge operations, and a generalizable framework. The majority of studies focus on optimized auto-scaling in cloud environments, particularly CPU and memory scaling, instead of self-adaptation. These approaches do not address issues of architectural flexibility. Performance degradation may stem from overloaded responsibilities or inappropriate service decomposition rather than from a lack of computational resources [122, 108]. Adapting the architectural structure is necessary when services exhibit heterogeneous scaling behavior. Components within a system may exhibit different demand growth rates, making uniform scaling inefficient. For example, a validation service may encompass both functional and security validation. Security validation is often more computationally expensive and subject to greater variability. Decoupling such concerns and optimizing them independently reduces resource waste and isolates performance variability. In addition, system workloads evolve dynamically due to changes in user behavior, feature additions, and seasonal demand patterns. Structural adaptation enables systems to split services when increased concurrency or specialization is required, and to merge services when traffic decreases, and inter-service communication overhead dominates. Although microservice architecture improves modularity and scalability, it introduces additional latency, coordination complexity, and operational costs. Under low-load conditions, service consolidation can reduce network overhead, deployment complexity, and monitoring costs. By enabling runtime adaptation of the architectural structure, systems achieve elasticity at the architectural level, rather than solely at the resource level. This approach improves fault isolation, mitigates cascading failures, supports continuous evolution, and improves overall cost efficiency [122, 108].

Accordingly, this chapter proposes the notion of a *service colony*, a software architectural style for developing systems as groups of autonomous software services that cooperate to fulfill the global objectives of the overall system, increasing the level of autonomy and intelligence of distributed (micro)services-based systems. Each inhabitant service of a colony fulfills a specific part of the functionality of the overall system that the colony implements. By following rules, either prescribed or acquired through learning, and interacting with other inhabitants, the services demonstrate a swarm-like global intelligence in adapting the individual services, their deployment in the environment, and the config-

uration of communication links between the services and the environment that ensures continuous, effective, and efficient performance of the system. Though individual services may have different roles, no system components are envisioned to implement centralized control over the overall system's behavior and configuration. An inhabitant service of a colony can interact with other services and the environment. Through interactions with individual services, the system interacts with its users to receive tasks and return results, as well as with the hardware, software, and network infrastructure, to optimize resource utilization. The interactions between the services ensure that complex functions composed of the constituent services' functionalities can be realized.

By monitoring both their behavior and environment, the inhabitants of a colony identify their own and the overall system's performance bottlenecks and resource constraints, and automatically adapt themselves and their communication links to overcome these issues. Therefore, service colonies follow a proactive adaptation strategy, taking corrective actions before problems occur. Consequently, a service colony monitors, adapts, and optimizes its behavior through self-diagnosis and adaptations of individual components based on environmental conditions. Thus, it self-manages and self-architects in a dynamic environment. Moreover, it is an autonomous, goal-oriented, and goal-directed system that aims to enhance the quality of service. This supports service colonies in extending self-adapting software systems, reducing the need for costly human interventions typically required for continuous monitoring, troubleshooting, and application maintenance.

The effectiveness of service colonies was validated using a proof-of-concept. The evaluation focuses on split, merge, and multiple split/merge scenarios, as well as the behavior of service colonies in resource-constrained environments. Experimental results indicate that service colonies improve response time and memory utilization of the system. The benefits are particularly pronounced in resource-constrained environments, manifesting itself as improvements in throughput, average response time, adaptation efficiency, and memory consumption. These results suggest that service colonies are well-suited for IoT and edge computing environments, where computational and memory resources are limited.

The remainder of the chapter is organized as follows: the concept of the service colony(Section 5.2), proof-of-concept and evaluation(Section 5.3), benefits and challenges of the service colony(Section 5.4), implementation and reproducibility(Section 5.5), and summary(Section 5.6).

5.2 Service Colonies

This section introduces the components and interactions that define service colonies. It provides a detailed discussion of the structural elements of inhabitants, focusing on their adaptive mechanisms and how these mechanisms support the dynamic evolution of the colony. Furthermore, the service colony concept is framed as a self-adaptive software system.

5.2.1 Components and Interactions

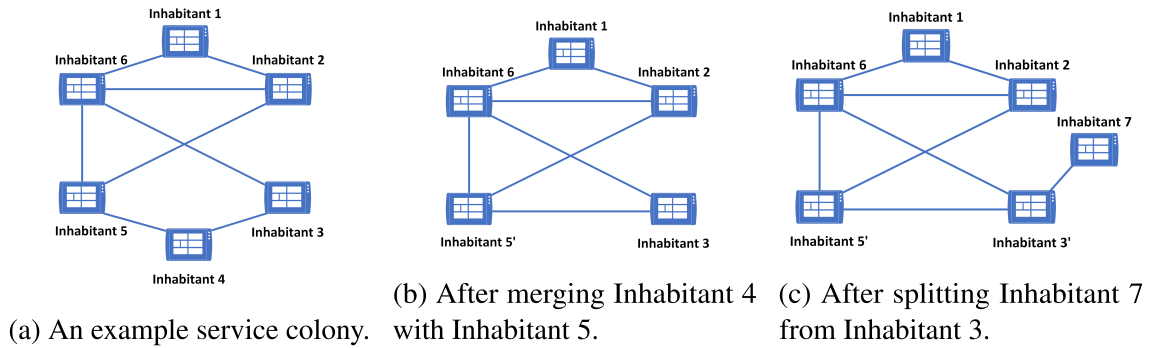
A *service colony* is a system composed of interacting software services, referred to as *inhabitants*, that collaborate to perform specific functions. These inhabitants are deployed in a distributed cloud platform, edge computing infrastructure, or Internet-of-Things devices, enabling the system to operate across diverse and decentralized resources. Depending on its design, a service colony can exhibit varying degrees of decentralization in its control and decision-making processes. These range from fully decentralized configurations, where individual inhabitants make autonomous decisions, to divisional or hierarchical structures, and even fully centralized models. This flexibility allows service colonies to adapt to different operational contexts, balancing scalability, responsiveness, and the complexity of coordination.

An *inhabitant* of a colony is an autonomous entity responsible for performing a specific functionality, or *service*, within the system. Each inhabitant has distinct roles and capabilities, reflecting its responsibilities and goals within the colony. Inhabitants aim to optimize their individual performance while contributing to the overall system's efficiency and quality of service. As self- and situationally-aware entities, they continuously monitor their performance relative to their assigned responsibilities, service obligations, and environmental conditions. Based on this awareness, inhabitants plan and execute actions to maintain or improve service quality. These actions may include replicating themselves, migrating to a more performant compute node, splitting into smaller entities, or integrating with inhabitants from another colony. Additionally, inhabitants can leverage learning mechanisms, such as reinforcement learning or rule-based adaptation, to refine their operations and decision-making based on prior experiences, enabling continuous improvement and adaptability.

An inhabitant interacts with other inhabitants and the environment by requesting services, providing services, or engaging in collaborative activities. Interaction patterns in a service colony can range from simple, direct message exchanges between two inhabitants to complex collaboration protocols involving multiple inhabitants and environmental inputs. These protocols may include synchronous or asynchronous communication, negotiation mechanisms, or coordinated workflows. Through these interactions, inhabitants coordinate their activities, exchange information, and interface with the system's users by processing inputs and delivering outputs, ensuring seamless operation and functionality within the colony.

The environment of a service colony comprises its inhabitants, the computing infrastructure supporting the colony, and the communication channels that enable interactions among them. Special components of the environment are the users, who interact with the colony to accomplish tasks and functions. Users achieve their goals by engaging with the colony's interface entities and receiving the results of their requests. This environment is inherently dynamic, as the state of its components and interactions evolves, and stochastic, due to the unpredictable nature of user requests, network conditions, and resource availability.

The service colony has decentralized monitoring, with inhabitants acting as autonomous agents. In general, there is no special inhabitant responsible for overall communication, monitoring, or dynamic adaptations. Instead, each inhabitant can learn from its behavior and automatically initiate actions to adjust the system. Each individual in a service colony can be (sub-)optimal, but collectively they aim to optimize the overall system performance. Figure 5.1a sketches an example composition of service inhabitants in a service colony. Arcs between inhabitants indicate the links used to support communications between the individuals. Inhabitants can communicate with other inhabitants, for example, via messages passing through communication links. Analytical messages and behavioral messages are the two types of special messages sent to the environment. Each inhabitant can initiate a change in the colony based on its analysis of itself and the environment using analytical messages. These changes can involve an inhabitant splitting, merging with another inhabitant, or adding or removing communication links between inhabitants. Furthermore, an inhabitant can send behavioral messages to the environment, indicating its capabilities or limitations. Examples of behavioral messages include the availability of more resources to be occupied by another inhabitant, delays in requests or responses, or



(a) An example service colony.

(b) After merging Inhabitant 4 with Inhabitant 5.

(c) After splitting Inhabitant 7 from Inhabitant 3.

Figure 5.1: Service colony adaptation process.

the volume of data in requests or responses.

5.2.2 Inhabitants

An inhabitant is the core building element of a service colony system. It encapsulates a delegated service, part of the functionality of the overall system, and monitors and proactively optimizes its performance. It is a self-contained entity with a designated functionality that can be clearly distinguished from other inhabitants. Inhabitants are autonomous agents that can act and react independently. Inhabitants have a dynamic state that can change over time. The future actions taken by an inhabitant depend on its current state. Inhabitants can share their state with other inhabitants in the colony. It is intended that an inhabitant frequently communicates and delivers services to other inhabitants. Thus, the current state of an inhabitant can be influenced by the state of another inhabitant in the colony and its behavior. Inhabitants are exploratory, and they can learn from the environment and adapt their behavior based on their experience. Therefore, inhabitants can proactively adjust the system based on their behavior. Each inhabitant has goals, making it goal-directed and goal-oriented. An inhabitant attempts to accomplish these goals through its behavior and to optimize the individual and overall system objectives via adaptive learning.

Figure 5.2 depicts an example architecture of an inhabitant. Each inhabitant has a boundary and can send messages to and receive messages from the external environment. The sub-modules of an inhabitant include system functionality, communication, knowledge, and decision-making. These sub-modules communicate based on their responsibilities to

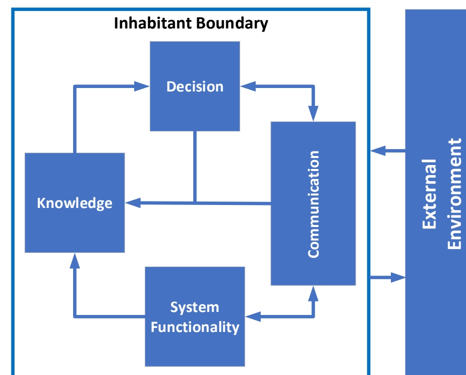


Figure 5.2: An example architecture of a service colony inhabitant.

provide the functionalities of the inhabitant.

The *communication* sub-module manages all interactions of the inhabitant and serves as its entry and exit point. Messages received from the environment, including those from other inhabitants, are captured by the communication module and forwarded to the relevant sub-modules. Messages initiated by the inhabitant and addressed to the environment are also routed through the communication sub-module for dissemination to their recipients. Upon receiving a message, the communication module validates its content to determine which sub-module is responsible for further processing. For example, functionality-related messages, such as requests to perform a function, are directed to the system functionality sub-module. Messages concerning environmental behavior are forwarded to the knowledge sub-module, while those pertaining to configurations and adaptations of inhabitants are redirected to the decision sub-module. Messages sent to the environment are routed to their intended recipients through the established communication links. Analytical messages can target individual inhabitants, subsets of inhabitants, or all inhabitants within the service colony. Behavioral messages are delivered to colony inhabitants based on the communication links within the topology of the colony.

The service colony has obligatory system functions distributed among its inhabitants. The *system functionality* sub-module is responsible for executing the inhabitant's delegated functionalities. Each inhabitant collaborates with other inhabitants in the service colony to fulfill these obligations. After completing a delegated function, an inhabitant responds to other inhabitants or users in the environment who have requested the result of the function via the communication module. Additionally, it sends log data to the knowledge sub-module to collect internal behavioral information.

Inhabitants monitor themselves and their environment. The *knowledge* sub-module builds the inhabitant's intelligence that aggregates the experiences of its monitoring processes. During initialization, an inhabitant collects data related to the service colony's structure and behavior. This information includes the total number of inhabitants, their configurations, and initial behavioral data of inhabitants. During execution, data and experiences stem from two sources. Functional data is collected from the system functionality sub-module. Additionally, an inhabitant gathers behavioral and analytical data of the service colony via the communication sub-module. The inhabitant does not accept all the data received from the environment. It filters data based on quality and relevance. Furthermore, it tracks the data origin for the decision-making process executed by the decision sub-module.

Inhabitants can react to changes in their environment. They learn and take action based on their behavior and the evolution of the environment. The *decision* sub-module handles these learning and decision-making functions. This sub-module depends on data provided by the knowledge sub-module. Learning is driven by data available in the knowledge sub-module and by previous decisions made by the decision sub-module. An inhabitant can use adaptive learning to dynamically adjust to the environment based on the current and past behavior of the system. Data collected after implementing a decision is used as feedback for learning. The decision sub-module handles two functions. First, if data from the inhabitant indicates a relevant behavior, it disseminates that message to the environment via the communication sub-module, using behavioral messages. Second, it analyzes inhabitant and environmental data to identify system operations, such as those relevant to rearchitecting the system, and actions to improve performance. Such information is shared with other inhabitants in the colony via analytical messages. Each inhabitant has objectives captured as collections of rules within the decision sub-module. These rules can be updated dynamically via the interface provided to the decision sub-module. Therefore, during execution, new rules can be added, while existing rules can be modified or deleted.

5.2.3 Adaptations

An inhabitant can replicate itself. For example, if an inhabitant experiences a high volume of requests and no further optimizations are possible, it can decide to replicate itself to accommodate the increased workload.

A service colony can adapt its structure to the environment and optimize its communication patterns. Four basic operations are needed to support such adaptations: merging two or more inhabitants, splitting an inhabitant into two or more inhabitants, and adding and removing communication links between inhabitants. Two or more inhabitants can *merge* to form a single inhabitant, for example, to simplify the communication structure between themselves and the rest of the colony during a low system load. Alternatively, such joining can be triggered to maximize resource utilization in the system, thereby reducing operational costs. A complex merge of multiple inhabitants can be implemented as a sequence of atomic joins between pairs of inhabitants. That is, two inhabitants merge in the first step. Then, another inhabitant joins with the previously merged inhabitant. An inhabitant can *split* into multiple inhabitants via a sequence of atomic splits of one inhabitant into two inhabitants to distribute its functionality and responsibilities among multiple system elements. Such splittings can be initiated based on identified performance bottlenecks and resource-intensive functionalities that degrade the quality of the service provided. Hence, it can split its functionalities to maximize performance, for instance, by replicating those decomposed services that receive high request loads. Finally, an inhabitant can establish direct communication links with other inhabitants, for example, to reduce current communication latency or to identify peer inhabitants from which to request required services. Moreover, merging and splitting operations in the system can add or remove communication links in the colony.

Figure 5.1b depicts the re-architectured system from Figure 5.1a after merging Inhabitant 4 with Inhabitant 5. This operation leads to redirecting the communication links of Inhabitant 4 to and from the resulting Inhabitant 5′. An example splitting of Inhabitant 3 is depicted in Figure 5.1c. In this case, Inhabitant 7 is split out of the Inhabitant 3 in Figure 5.1b, creating Inhabitant 7 and Inhabitant 3′. Hence, a communication link is established between the resulting inhabitants. Inhabitants can use this link to request services from one another.

One can develop different strategies for deciding which adaptations to the system structure to implement. Such strategies can result in the authoritative implementation of the intended adaptation or an adaptation confirmed by peers, a group of inhabitants, or the entire service colony. Inhabitants execute authoritative actions without requesting confirmation from other members of the colony. Alternatively, an inhabitant can negotiate the intended splittings and joins with other inhabitants to ensure mutual agreement and

maximal benefit for all the negotiators.

Inhabitants can request confirmation from their peers in case the change affects them. For example, this communication can follow an approach similar to the two-phase commit protocol [179]. If the change affects a group of colony inhabitants, permissions for the change can be requested from the affected inhabitants. For instance, Inhabitant 5 in Figure 5.1a decides to merge with Inhabitant 4, and requires obtaining permission from Inhabitants 3 since Inhabitant 4 has a direct communication link with Inhabitant 3. The execution of an adaptation decision is initiated by sending a message to the relevant inhabitants requesting permission. Once these requests arrive at the recipients, they validate the requested change. The validation is based on the knowledge stored within the decision sub-modules. Once validated, the inhabitants reply with approvals or rejections of the change. If the change is approved, the requesting inhabitant executes the change process. At the beginning of this process, the inhabitant confirms the start of the change to the environment. After the inhabitant receives a confirmation to start the process, it should not accept further adaptation requests until the current execution process is completed and confirmed. Then, the inhabitant executes the change. After completing the change, it informs the environment by sending the execution completion message. Then, other inhabitants update their configurations based on the change that was executed. These updates can involve adding, removing, or updating links between inhabitants.

Figure 5.3 illustrates the scenario of merging two inhabitants 4 and 5. In this case, Inhabitant 5 identifies the need to merge and acts as the leading inhabitant of the change process. First, it seeks to get confirmation from Inhabitant 4, the inhabitant it intends to merge with. Once Inhabitant 4 accepts the merging request, the next confirmation request is sent to Inhabitant 3 to get approval to merge. If the inhabitants respond positively, the merging process commences. After the merge process completes, the execution confirmation message is sent to the Inhabitant 3 to update relevant configurations. Figure 5.4 illustrates the sample communication sequence for splitting Inhabitant 3 illustrated in Figure 5.1c. The Inhabitant 3 requests to split from Inhabitant 5 and Inhabitant 6. Upon the acceptance of both requests, the splitting task is confirmed, and the splitting process is executed. After successful splitting, a confirmation is sent to the inhabitants. Then, the relevant inhabitants of the colony update their knowledge and links. If the inhabitant's request to split is rejected, no further actions are taken, and the communication ends.

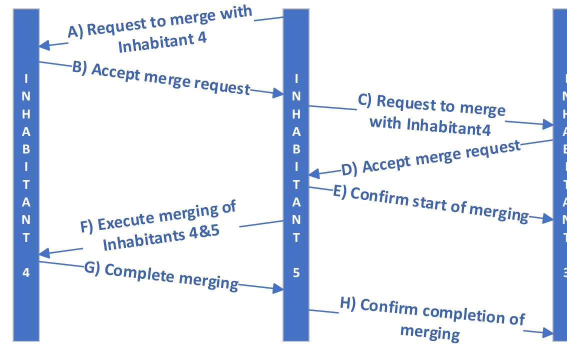


Figure 5.3: Agreement for merging Inhabitants 4 and 5.

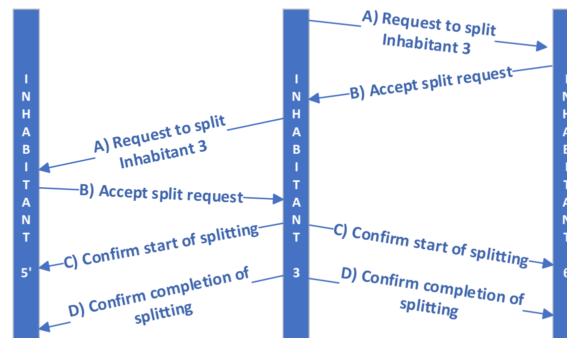


Figure 5.4: Agreement for splitting between Inhabitants 3, 5′, and 6.

5.3 Proof of Concept and Evaluation

A “Todo” item management application has been developed as a proof-of-concept prototype to validate the effectiveness of service colonies.[1] The prototype supports basic splitting and merging operations of inhabitants. While a service colony, in general, is envisaged to support proactive decision-making by its inhabitants, for simplicity, the prototype implements rule-based, reactive decision-making. A service colony can identify services requiring splitting through comprehensive system data analysis. Once a split is confirmed,

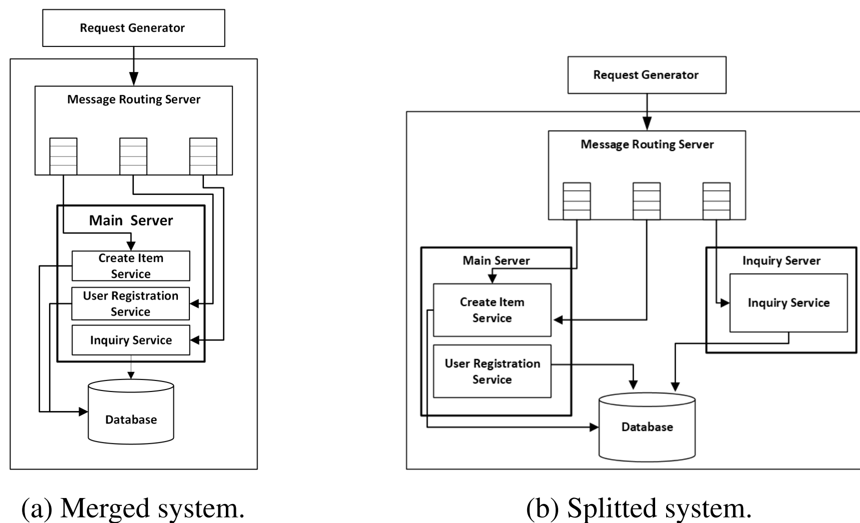
[1] https://github.com/thakshilad/ServiceColonyResources

the service will be automatically deployed on the appropriate server(s). During startup, the service would communicate with other colony services, update the colony configurations, and begin handling service requests. To test the effectiveness of the service colony concept, pre-deployed services (split and merged) were used instead of automated extraction and deployment.

The Todo item management application consists of three services: Create Todo Item, User Registration, and Inquiry. The Create Todo Item service creates and persists new todo entries, the User Registration service creates and persists system users, and the Inquiry service performs a database search to retrieve a given todo item. A message-routing service directs incoming requests to the appropriate service, maintaining a dedicated queue of requests for each service. A separate tool, Request Generator, was developed to simulate user behavior by concurrently sending messages to each service at controllable rates. During testing, the Create Todo Item service and the User Registration service were always deployed as a single inhabitant of the colony on the main server. The Inquiry service was designed to support merging with or splitting from this single inhabitant. Initially, all three services were deployed on the main server as a single inhabitant, as shown in Fig. 5.5a. This illustrates a scenario in which an inhabitant implements multiple services. If this inhabitant decides to split, the inquiry requests are handled by a dedicated inquiry inhabitant deployed on a separate server, as depicted in Fig. 5.5b. If the two inhabitants decide to merge, the system re-architects itself back into the configuration shown in Fig. 5.5a. The Inquiry service was chosen for split-off and merge-back operations because it involves querying data from the database (retrieving stored items), whereas the Create Todo Item and User Registration services perform only direct, lightweight database insertions. The split and merge operations were simulated by routing messages between the main and inquiry servers, both of which were pre-deployed to test the service colony operations.

5.3.1 Self-Adaptation Description Language for Service Colony

Modeling self-adaptive systems is inherently challenging due to uncertainty in their operational environments. Self-adaptive capabilities further increase this difficulty because adaptation-related features are distributed across multiple parts of the system. As a result, models often become tightly interwoven, making the system harder to understand [180].



(a) Merged system. (b) Splitted system.

Figure 5.5: Two service colony deployments.

To address this complexity, several Domain-Specific Modeling Languages (DSMLs) have been proposed [181]. The Adapt Case Modeling Language (ACML) is one such DSML. It follows the Concern-Specific Modeling Language (CSML) approach, which is built on top of the Unified Modeling Language (UML). ACML separates concerns explicitly and provides dedicated constructs for specifying self-adaptive behavior [180]. Therefore, ACML is adopted as the description language for service colonies.

ACML comprises two major components: the adaptation view, which is the structural model, and adapt cases, which are behavioral models of the system. The adaptation view illustrates core components of the system with sensors, effectors, adaptation properties, and knowledge, as a UML-like component view. A sensor is an interface/class that enables the system to observe or measure relevant properties. An effector is an interface/class used by the system to perform actions that change its state or the environment.

Figure 5.6 shows the sample low-level adaptation view of the service colonies, including sensors and effectors. It contains three inhabitants, X, Y, and Z. The internal module structure is depicted only for the inhabitant X. Both inhabitant Y and Z have a similar structure. Thus, for simplicity, only their interaction with the colony is depicted in the figure.

The knowledge module is responsible for acquiring and maintaining information relevant to the colony and its inhabitants. To support this role, sensing and context-

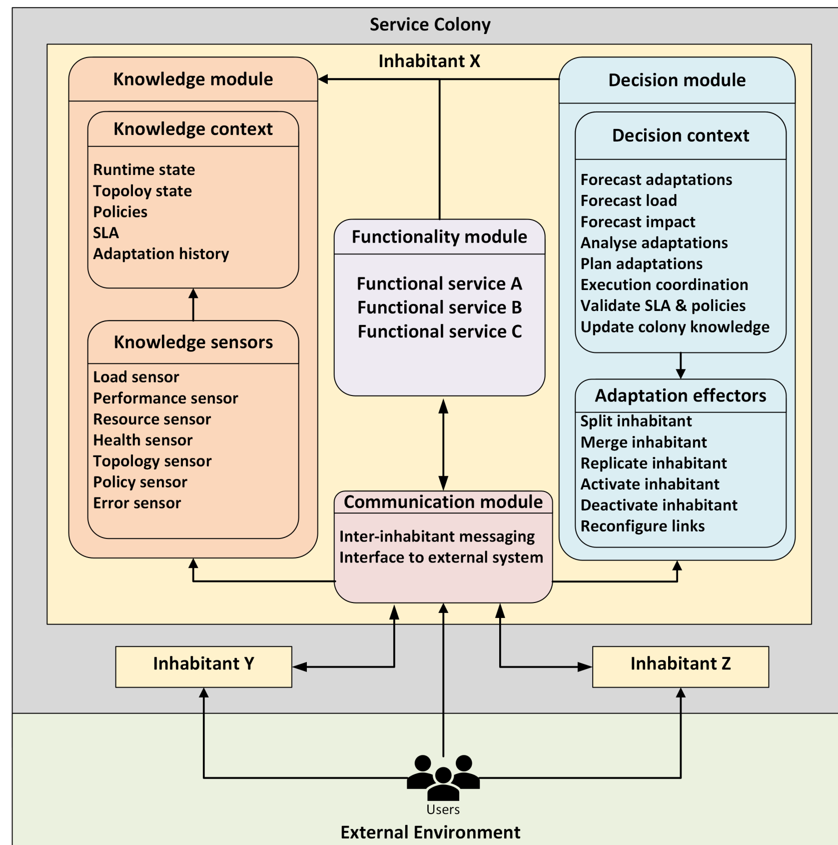


Figure 5.6: ACML adaptation view: An example model for service colonies.

awareness capabilities are embedded in the module, enabling it to maintain an up-to-date view of both the inhabitant's state and its operating environment. The knowledge module maintains several categories of context information, including the runtime status of the colony, the current topology of interconnected inhabitants, adaptation policies, Service Level Agreements (SLAs), and adaptation history. This contextual knowledge is derived from data captured through a set of sensors embedded in the knowledge module. Service colonies incorporate multiple sensor types to observe both operational and structural conditions. The most fundamental sensor is the service load sensor, which monitors the incoming request rate per inhabitant. Another key sensor is the performance sensor, which, in the implemented prototype, measures queue size, i.e., the number of pending requests awaiting processing by an inhabitant. In addition, each inhabitant continuously monitors its own operational status and contributes information about overall resource utilization

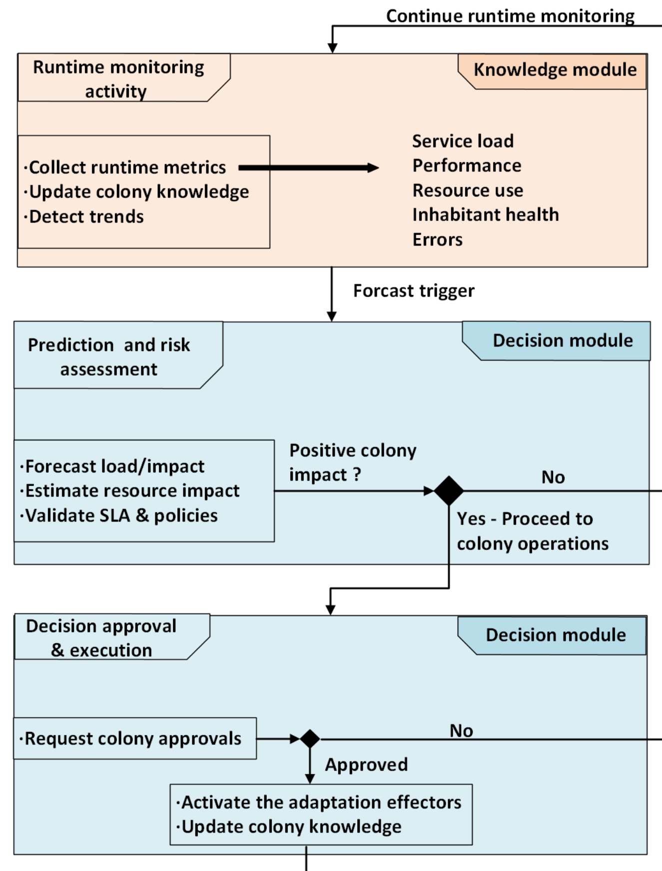


Figure 5.7: Example ACML adaptation cases for service colonies.

within the colony, thereby acting as a distributed sensing entity. Moreover, the knowledge module captures the changes in colony topology, policy updates, and inhabitant-level errors or failures.

The decision module functions as the adaptation effector and control unit, performing reasoning and executing adaptive decisions. It is responsible for analyzing contextual information captured by the knowledge module and triggering appropriate adaptation actions when required. Figure 5.6 illustrates sample actions performed by the decision module during runtime adaptation. The decision module performs a range of operations, including identifying the need for adaptation, analyzing and forecasting load variations, assessing the impact of potential changes, evaluating adaptation risk, planning the adaptation process, and requesting coordination or approval from relevant fellow inhabitants when required. In addition, it is responsible for executing adaptation decisions, validating against applicable

SLAs and policies, and updating the knowledge module once the adaptation process has been completed. These controlling functions lead to the selection and execution of adaptive decisions, which are carried out through a set of adaptation effectors. The service colony architecture provides effectors to support key structural and behavioral adaptation actions, including splitting, merging, replicating, activating/deactivating inhabitants, and runtime reconfiguration of colony relationships and interactions.

The system functionality module is responsible for performing domain-specific business functionalities of the inhabitant. Through this module, an inhabitant can provide one or more services that are exposed to external users or other inhabitants within the colony.

The communication module manages the communication and message exchange between inhabitants and the external environment. It acts as the interaction layer through which functional service requests and inhabitant-related messages are transmitted. These may include business requests, adaptation-related notifications, and other coordination information required for colony operation.

The adapt cases of the ACML model define how the system monitors its environment and the type of actions it should perform in response. Examples of adapt cases are shown in Figure 5.7. It captures the logic for monitoring and knowledge capture, predicting possible adaptive operations, performing impact analysis, and executing adaptive operations via effectors.

5.3.2 Experimental Results

Four test scenarios were conducted to evaluate the performance of the service colony with respect to the metrics defined in Section 2.5. The incoming request rate and queue size were selected as key parameters for adaptive decision-making due to their ease of measurement and real-time availability.

The testing setup contains five Ubuntu servers and one MySQL database instance. Each server has 4 VCPUs, 16GB RAM, and a 30GB hard drive. The database instance has 4GB RAM and a 1GB hard drive. The service colony logged its operations under different workloads, which were subsequently analyzed. The testing scenarios and the results are discussed below.

5.3.2.1 Scenario 1: Splitting

This scenario evaluated the performance of the split function of the service colony. The following three cases are evaluated in this scenario:

Case 1: No splitting. All the requests are served from the main server(Fig. 5.5a).

Case 2: Splitting the Inquiry service at the IRR reaches or exceeds 1,000 requests per second(Fig. 5.5b).

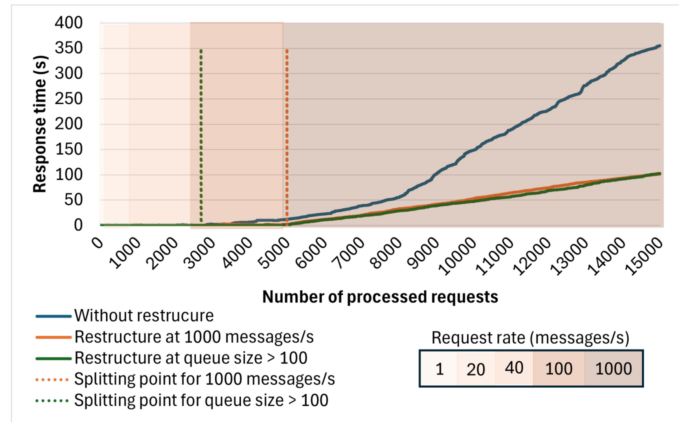
Case 3: Splitting the Inquiry service when the number of requests to be processed in the Inquiry requests queue size exceeds 100(Fig. 5.5b).

Case 1 represents the behavior of the original software system, while Cases 2 and 3 simulate an inhabitant splitting in a service colony under overload conditions.

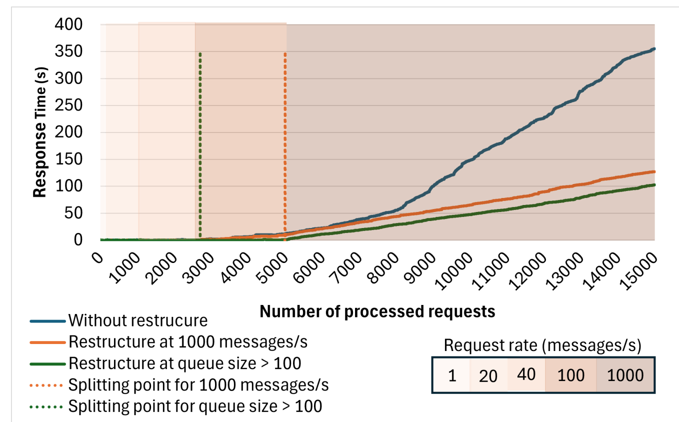
Initially, all three services were deployed on the main server, as depicted in Fig. 5.5a. To simulate overload conditions, the request generator progressively increased the IRR from 1 request per second to 1,000 requests per second for each service for all the cases. The selected IRR range and corresponding thresholds were determined through empirical evaluation of the system. Within this interval, response time exhibited significant variation as the workload increased. Beyond 1,000 requests per second, the system reached a saturation state in which response time remained relatively stable despite further increases in IRR. Therefore, 1,000 requests per second was chosen as the upper bound.

All services were subject to gradually increasing workloads to simulate server overload conditions. The Inquiry service is latency-sensitive due to database lookups. Therefore, the service colony activates the splitting function for the Inquiry service when either IRR or QS exceeds a predefined threshold. In scenario 2, the IRR threshold was set to 1,000 requests per second, as this value corresponds to the saturation point of the system. In scenario 3, the QS threshold was set to 100, based on empirical observations indicating this value as the onset of response time degradation with increasing queue size. After the IRR reached 1,000 requests per second, the load was maintained to keep the system saturated and prevent the services from merging onto a single server during the experiment. The performance of the splitting function was measured using the response time metrics. In addition, memory consumption was analyzed to assess resource utilization.

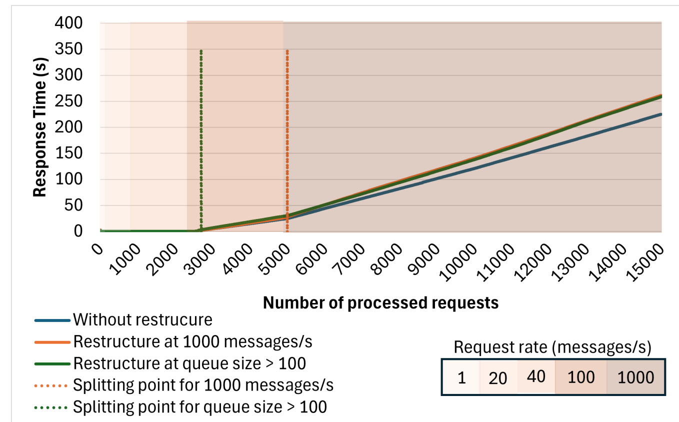
Figure 5.8 shows the response time changes of the three services over time. While the Inquiry service does not display a significant improvement following the splitting, both



(a) Create Todo Item service.

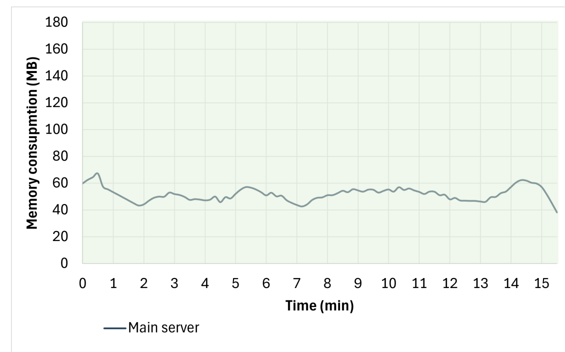


(b) User Registration service.

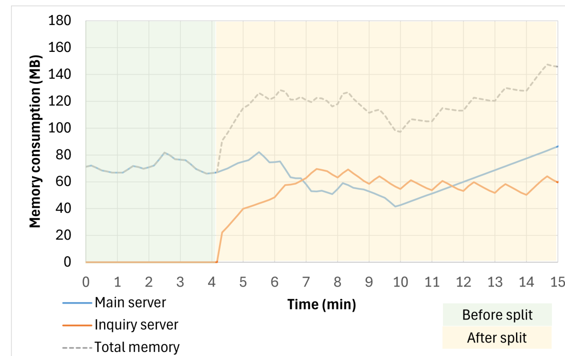


(c) Inquiry service.

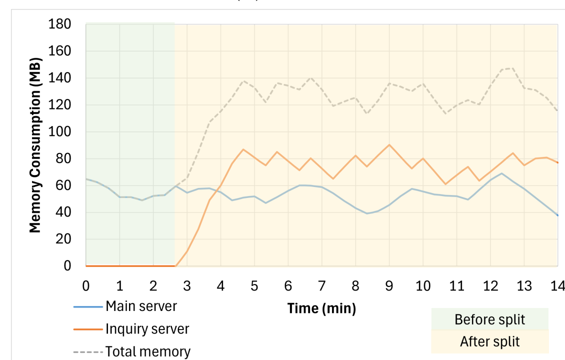
Figure 5.8: Scenario 1: Response time analysis (splitting).



(a) Case 1.



(b) Case 2.



(c) Case 3.

Figure 5.9: Scenario 1: Memory consumption analysis (splitting).

the Create Todo Item and User Registration services experience a notable reduction in response time. This reduction is attributed to enhanced resource availability after offloading the resource-intensive inquiry request processing. Furthermore, splitting based on queue size marginally outperforms splitting based on response rate. Figure 5.9 depicts the total memory consumption for the three cases. As expected, the total memory consumption

by the system is negatively impacted since splitting introduced an additional server to the system. In summary, the results reveal a seamless transition between the two architectural configurations, both before and after splitting, accompanied by a considerable improvement in response time.

Furthermore, Table 5.1 depicts the statistical analysis of the results, based on the defined metrics: throughput, average response time (ART), adaptation efficiency (AE), and memory gain (MG). According to the metrics' definitions, higher throughput indicates better performance, whereas lower ART reflects improved system responsiveness. Additionally, positive values of AE and MG represent performance improvements in the service colony compared to the baseline configuration. The comparative analysis confirms that, in the splitting scenario (Scenario 1), the service colony architecture improves throughput, reduces average response time, and yields positive adaptation efficiency for create-todo and register services. However, this configuration adversely affects memory utilization, and the split service's (Inquiry) performance may degrade.

5.3.2.2 Scenario 2: Merging

This scenario tested the merging of inhabitants. Initially, the system operates in the deployment mode shown in Fig. 5.5b. After observing a low inquiry request rate, it merges its two inhabitants into one as depicted in Fig. 5.5a.

Throughout the experiment, the Create Todo Items and User Registration services maintained a constant IRR of 20 requests per second. This controlled workload ensured that the server hosting these two services was not overloaded when the inquiry service was merged back. A minimum stable workload was required to preserve consistent system behavior. Based on the empirical evaluation, an IRR of 20 requests per second provided stable performance. Therefore, this rate was maintained for both services during the experiment.

In contrast, the Inquiry service was used to simulate varying load conditions. Its request rate was gradually reduced to emulate a decreasing IRR, starting from 1,000 requests per second and decreasing to 1 request per second. An IRR of 1,000 requests per second was sufficient to overload the system. Accordingly, the Inquiry service simulated a transition from high-load to low-load conditions by decrementing the IRR from 1,000 to 1 request per second. The queue size was not used in the merging decision in this experiment to

Table 5.1: Adaptation scenario evaluation results.

Scenario	Case	Throughput			ART			AE			MG
		Todo	Inquiry	Register	Todo	Inquiry	Register	Todo	Inquiry	Register	
1	1	0.021	0.039	0.021	307870.760	83832.268	309059.331				
	2	0.053	0.024	0.053	45662.401	185200.552	45722.105	0.851	-1.209	0.852	-0.997
	3	0.042	0.036	0.042	77873.044	96548.193	78243.879	0.747	-0.152	0.747	-1.155
2	1	0.019	0.020	0.019	35.614	112308.498	36.336				
	2	0.019	0.019	0.019	152.200	132198.792	155.248	-2.763	0.825	-2.774	0.352
3	1	0.019	0.007	0.019	16.791	141326.264	16.798				
	2	0.019	0.008	0.019	19.865	114015.614	20.028	-0.183	0.193	-0.192	-0.119
4	1	0.029	0.026	0.028	220557.237	273584.176	219514.829				
	2	0.052	0.029	0.052	141223.403	244031.893	140818.619	0.359	0.108	0.358	0.256

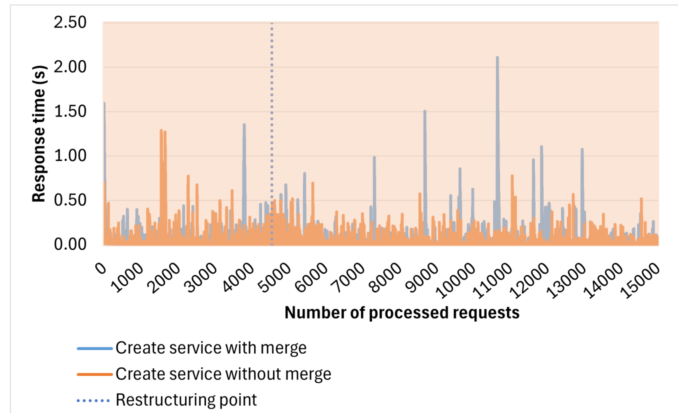
avoid responding to temporary queue fluctuations caused by decreasing IRR values. Two cases were analyzed in this scenario:

Case 1: No merging. The main server runs one inhabitant that contains the Create Todo Item and User Registration services. The Inquiry service is deployed on a dedicated server (Fig. 5.5b).

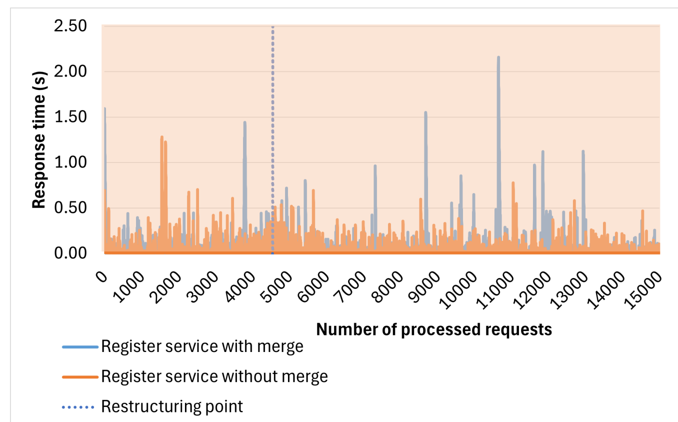
Case 2: The Inquiry service merges with the Create Todo Item and User Registration services and is deployed on the main server (Fig. 5.5a) when the request rate decreases to one message per second.

Case 1 simulates a software system with two services that do not support merging operations, while Case 2 validates merging functionality within the service colony. This scenario analyzes both response time and resource utilization, comparing the performance of a system employing the service colony with one that does not, thereby assessing the effectiveness of the merging function.

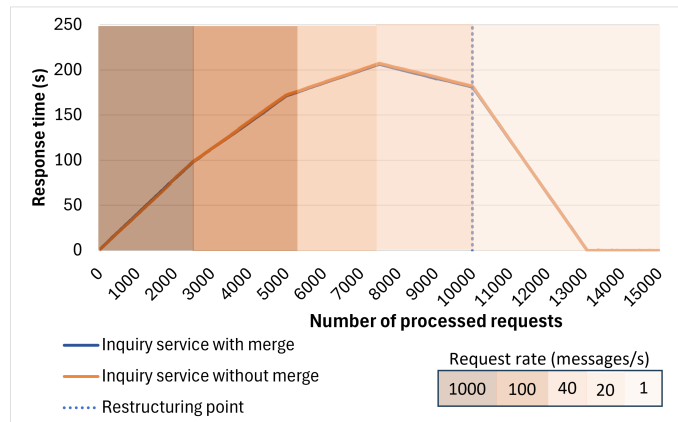
Figure 5.10 presents the results of the response time analysis. No significant impact on response time was observed in this scenario. However, a substantial reduction in total memory utilization was noted, as illustrated in Figure 5.11. The discontinuation of the inquiry server following the merging operation led to a decrease in overall memory consumption by approximately 50%, as illustrated in Fig. 5.11b. Furthermore, Table 5.1 yields positive memory gains but negatively affects throughput, average response time, and adaptation efficiency relative to the baseline service. Therefore, this experimental scenario demonstrates that resource utilization can be reduced without adversely affecting system performance.



(a) Create Todo Item service.

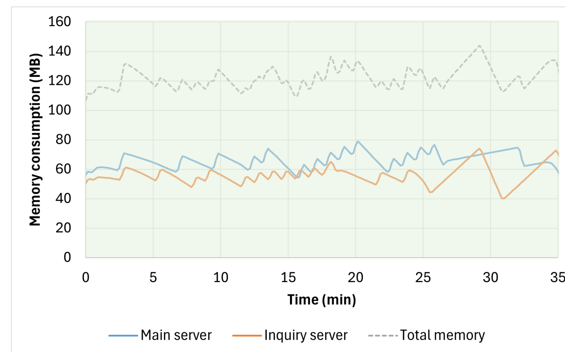


(b) User Registration service.

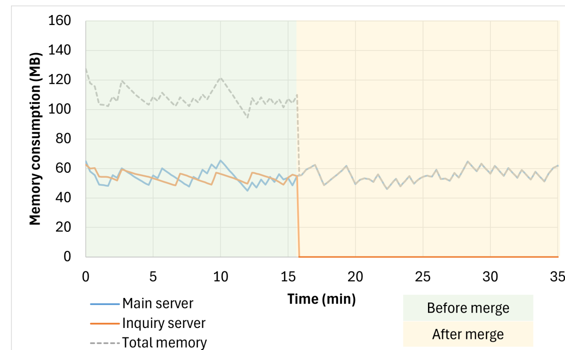


(c) Inquiry service.

Figure 5.10: Scenario 2: Response time analysis (merging).



(a) Case 1.



(b) Case 2.

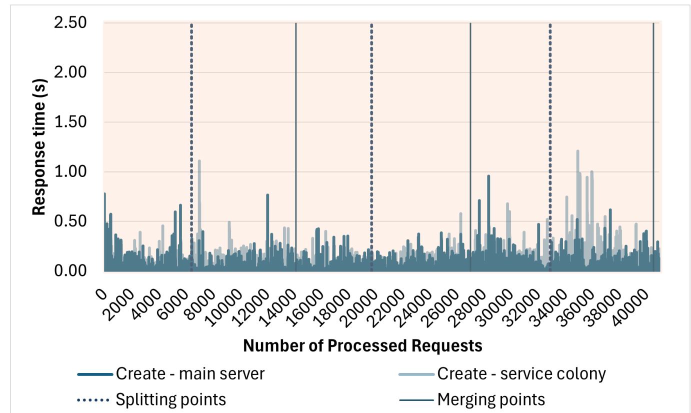
Figure 5.11: Scenario 2: Memory consumption analysis (merging).

5.3.2.3 Scenario 3: Multiple Splitting and Merging

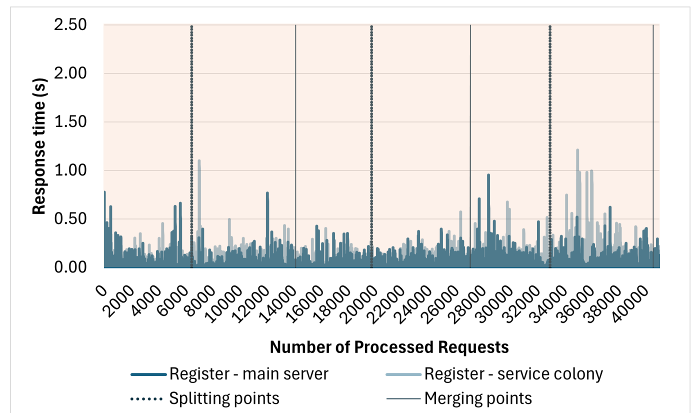
This scenario assesses the system performance under varying inquiry request rates to simulate the dynamic runtime operation of a service colony. As a result of these fluctuations, multiple merge and split adaptations occur during system operation.

As in Scenario 2, the Create Todo Items and User Registration services were maintained at a constant IRR of 20 requests per second to ensure stable behavior. In contrast, the Inquiry service IRR follows a wave-like pattern to emulate real-world workload variations. Specifically, the rate begins at 2 requests per second, gradually increases to 200 requests per second, then progressively decreases back to 2 requests per second, after which the cycle repeats. Although empirical evaluation indicates that the system becomes overloaded at an IRR of 1,000 requests per second, the load variation range in this experiment was limited to 2–200 requests per second to reflect the typical operating conditions rather than extreme cases.

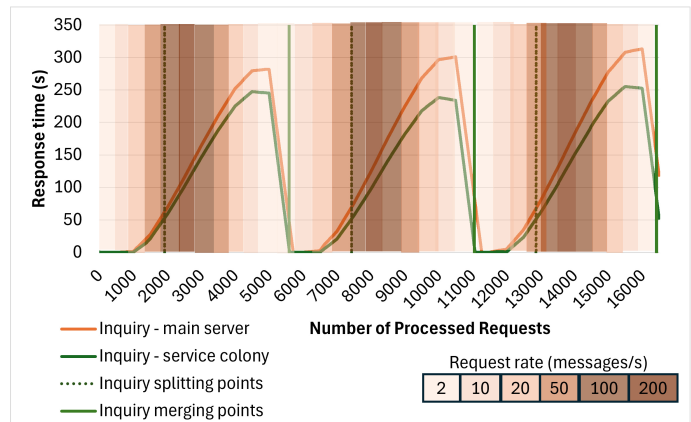
The split adaptation dividing the inhabitant executing all three services into two separate



(a) Create Todo Item service.



(b) User Registration service.



(c) Inquiry service.

Figure 5.12: Scenario 3: Response time analysis (splitting and merging).

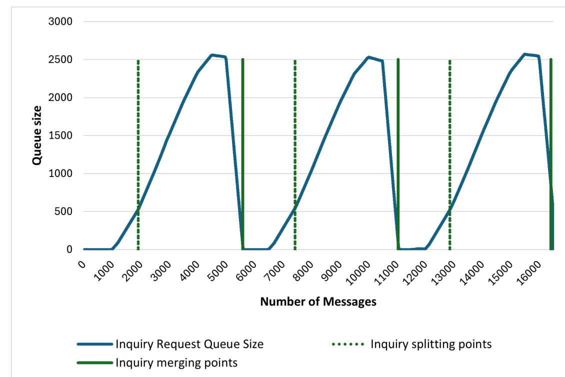


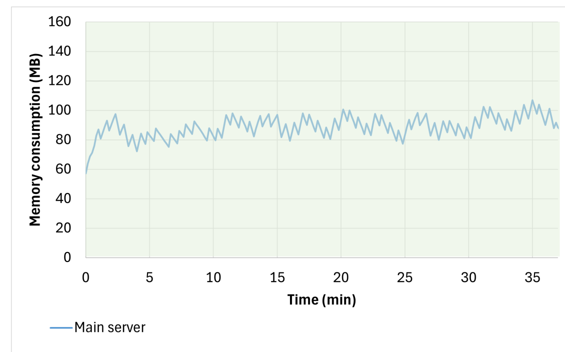
Figure 5.13: Scenario 3: Inquiry queue size fluctuation.

services, as illustrated in Fig. 5.5b was triggered when the queue size exceeded 500. Conversely, a merge adaptation consolidating the system into a single server, as shown in Fig. 5.5a was triggered when the inquiry queue size dropped below 50 requests. Since the request rate in this scenario was moderate, the queue size was selected as the primary adaptation metric, as it directly reflects system congestion. The threshold values were determined based on experimental observations of queue size fluctuations (see Fig. 5.13). Specifically, the split threshold of 500 requests was chosen because queue growth becomes sharp beyond this point, indicating rapid load escalation. The merge threshold of 50 requests was selected because the request rate effectively declined to near zero below this level. During the experiment, merging and splitting were observed three times each. Response time and memory consumption were analyzed in these two cases:

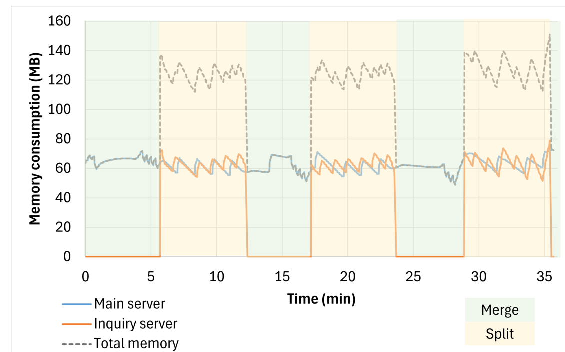
Case 1: The system is composed of a single inhabitant that implements all three services and executes on the main server without performing merge and split adaptations (Fig. 5.5a).

Case 2: The system alternates between one and two inhabitant configurations shown in Fig. 5.5a and Fig. 5.5b, respectively, as required, according to the described adaptation rules.

In this scenario, Case 1 represents the original software system, while Case 2 illustrates the behavior of the service colony system. The response times and memory consumption were analyzed and are presented in Fig. 5.12 and Fig. 5.14, respectively. Although the Create Todo Item and User Registration services are not significantly affected, the Inquiry service



(a) Case 1.



(b) Case 2.

Figure 5.14: Scenario 3: Memory consumption analysis (splitting and merging).

shows an improvement in response time, as demonstrated in Fig. 5.12c. Furthermore, the improvement in inquiry response time becomes more pronounced with each re-architecture cycle. The average total memory consumption of the two-inhabitant system, shown in Figure 5.14b, is comparable to that of the single-inhabitant system deployed on the main server, as depicted in Figure 5.14a. Therefore, this scenario highlights that, during the continuous operation of a service colony involving multiple merging and splitting operations, response time can improve while maintaining a similar overall average resource consumption.

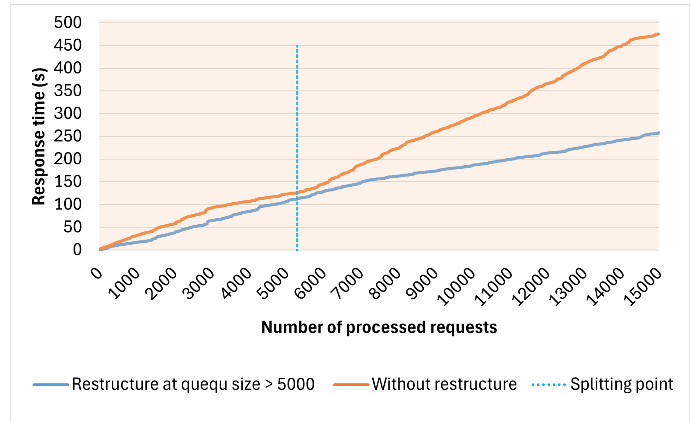
5.3.2.4 Scenario 4: Limited Resources

Scenarios 1 to 3 assume unlimited resources. In many real-world scenarios, resources are costly and, thus, constrained. In Scenario 4, system performance was evaluated under limited resource conditions. The main server CPU was stressed using the Ubuntu stress

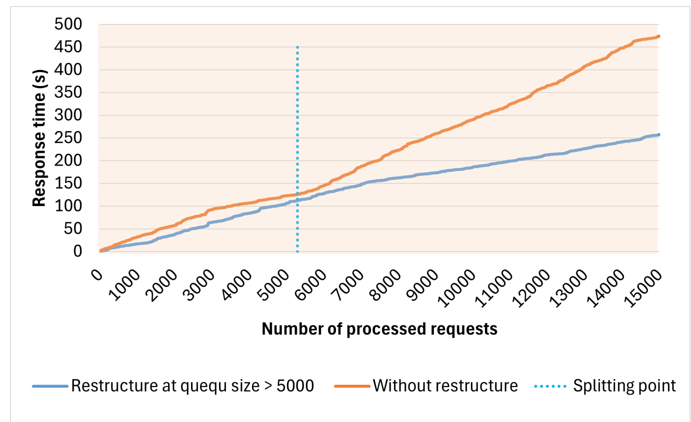
tool to simulate resource contention for the system. In addition, each inhabitant of the Todo application, implemented as a service colony, was started by limiting the Java Virtual Machine (JVM) memory allocation. Furthermore, memory constraints were simulated by allocating memory to arrays with 80 million entries. The response time of the three services and the memory consumption by the system's inhabitants were analyzed by allocating 32MB of memory per service. The memory allocation was chosen as 32 MB due to the JVM memory requirements for starting the system. Hence, during the execution, a total memory of 96MB was maintained across the system. These configurations were imposed to simulate the limited CPU and memory, thus simulating resource-constrained environments such as IoT and edge devices.

In this scenario, the service colony system was evaluated both with and without the splitting adaptation. Initially, the system was deployed with a single instance on the main server, hosting all three services within a single JVM, with appropriate memory allocations. Since JVM does not support dynamic memory allocation, upon reaching the splitting condition, the two inhabitants, one that implements the Create Todo Item and User Registration services, and the other that implements the Inquiry service, were deployed on fresh servers with the respective JVM memory allocations that maintain the necessary memory levels across the system. During the experiment, a constant IRR of 500 requests per second was applied to each service. This rate was selected based on empirical evaluations conducted under resource-constrained conditions, where an IRR of 500 requests per second was observed to produce a saturated system state. The splitting adaptation was triggered when the inquiry request queue size exceeded 5,000 requests. This threshold is higher than in other test scenarios because the system operated under resource constraints, leading to longer response times and consequently faster queue growth.

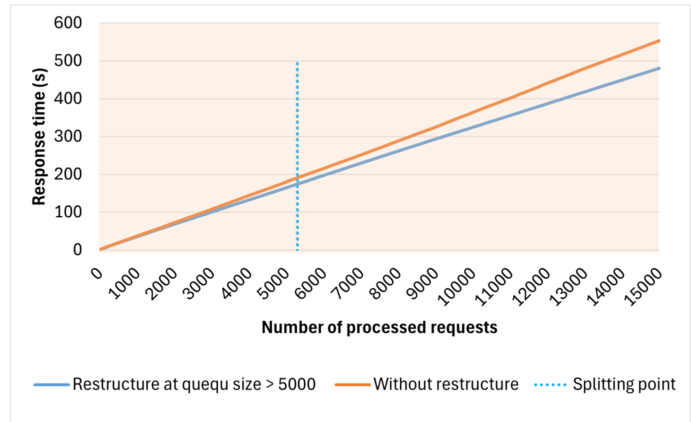
As illustrated in Figs. 5.15a and 5.15b, in the resource-constrained environments, the Create Todo Item and User Registration services exhibit better response time when the system is split into two inhabitants. When split, the response time of these two services increases linearly as more requests arrive, while it slows down substantially when a single inhabitant supports all three services. In turn, the response time of the Inquiry service shows a minor improvement when the system splits into two inhabitants, as depicted in Fig. 5.15c. The processing of an inquiry request, on average, is more resource-demanding than the processing of other services. Hence, when three services are implemented as single inhabitant and are executed by single JVM, most of the allocated memory is eventually



(a) Create Todo Item service.



(b) User Registration service.



(c) Inquiry service.

Figure 5.15: Scenario 4: Response time analysis (limited resources).

consumed by processing the inquiry requests, decreasing the overall performance of the system. When the Inquiry service is deployed on a separate JVM, the processing of the Create Todo Item and User Registration services accelerates substantially. In general, the system performs better when more memory is allocated. However, when the number of requests is overwhelming, this advantage gradually diminishes.

The memory consumption for the scenario is presented in Fig. 5.16. Case 1 represents the original software system, while Case 2 illustrates the behavior of the service colony system in a resource-constrained environment. Figure 5.16a illustrates the memory utilization of the main server. The main server is almost saturated and utilizes approximately 90MB of the total memory allocation, which is 96MB. In contrast, the implementation of service colonies exhibits lower overall memory consumption despite the system being distributed across two servers, as depicted in Fig. 5.16b.

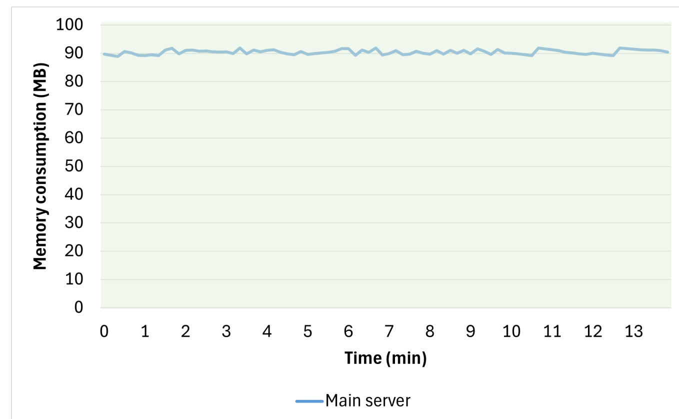
Furthermore, Table 5.1 confirms that under resource-constrained conditions (Scenario 4), the service colony architecture exhibits the most favorable overall performance, achieving higher throughput, lower average response time, positive adaptation efficiency, and memory gain. These findings indicate that the service colony architectural style is particularly well-suited for resource-constrained environments.

5.4 Benefits and Challenges

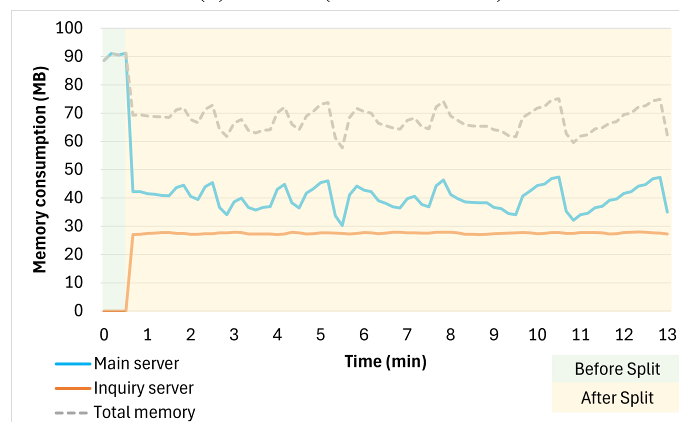
This section examines the benefits of implementing software systems as service colonies, as well as the challenges that may arise from such implementations.

5.4.1 Benefits

Both service-oriented and microservices architectures enable the identification of loosely coupled services that can be deployed independently. These services can be designed, developed, and tested autonomously. Additionally, containerization techniques allow these services to auto-scale in cloud-based environments based on performance demands. Generally, self-adapting systems address specific domain requirements, resource optimizations, and infrastructure adaptations by employing reactive error-handling functionality in pre-modeled scenarios [146, 182]. In contrast, service colonies are not constrained by pre-identified or pre-modeled scenarios. Instead, a colony can proactively re-architect



(a) Case 1 (one inhabitant)



(b) Case 2 (one inhabitant splits into two)

Figure 5.16: Scenario 4: Memory consumption (limited resources).

itself in a bottom-up manner through decentralized system analysis.

Consider an online shopping system developed as a service-based application. Assume that user registration and shopping cart actions are developed as two separate services within the system. In December, due to the festive season, the system receives a 100% surge in the volume of requests. During this high workload period, shopping cart services can experience performance bottlenecks, for example, due to latency in the payment handling process. A standard way to address this scenario is to increase resources for the entire shopping cart actions service. If this system is implemented as a service colony, it can identify that the bottleneck is caused by the payment transactions and split out this functionality into a new service. Subsequently, the colony can aim to utilize available resources to scale the payment transactions, for instance, by replicating the new payment

service while keeping other shopping cart actions within the original service. These adaptations can drastically reduce the operating costs of the system. Moreover, when the system experiences a low volume of requests, the shopping cart actions and payment handling services can merge back to achieve a smaller system footprint. Note that within a service colony, such splitting and merging happen automatically, reducing the costs of maintaining the system.
The benefits of a service colony to be at least the following:

- **Extension of microservices** Service colonies extend the capabilities of microservice-based architectures by inheriting their flexibility, resilience, modularity, optimized resource usage, and reduced operational overhead. As the industry increasingly adopts microservice-based systems, service colonies align with this trend, offering innovative options for engineering future software systems.

- **Enhanced flexibility** By fostering interactions among individual inhabitants, a service colony ensures that desired functionalities are delivered flexibly. For example, multiple inhabitants can deliver the same functionality based on different service agreement levels or with improved performance by proactively scaling or migrating the functionality to more productive compute nodes. In addition, each inhabitant can be developed, deployed, tested, and updated independently.

- **Proactive fault tolerance** Each inhabitant in the service colony generates its own analytics and shares them throughout the colony. The individual and collaborative analysis of this data supports effective predictions of the system's behavior, including potential future faults. Since the analysis is based on the runtime information of individual services, it can be used to effectively monitor, predict, and proactively respond to environmental changes and system faults. For example, in the proof-of-concept Scenario 1, each inhabitant independently monitors its own request rate and queue size, predicts overload conditions, and initiates adaptation decisions accordingly, without depending on a central monitoring agent to detect performance degradation and determine appropriate adaptation actions, thereby enabling proactive fault tolerance.

- **Continuous optimization** Through continuous learning from operational data at the individual inhabitant level, and consequently the entire service colony, progressively evolves into a self-optimizing system. This process enables ongoing performance

refinement, analogous to experimental scenario 3, in which inquiry response time continuously improves across successive adaptive cycles.

- **Goal orientation** Both the service colony as a whole and its individual inhabitants have objectives and strive to achieve them. Properties and threshold values at the System-wide and individual-inhabitant levels can be defined within the system, as in the proof of concept. Hence, the system's collaborative nature fosters iterative improvements toward shared goals.

- **Dynamic service composition** The dynamic introduction and integration of services within a service colony optimize system performance and resource utilization, seamlessly adapting to evolving demands and environmental changes. In particular, as demonstrated in Scenario 4 in the prototype, where the system operates in resource-constrained environments, the service colony can achieve positive adaptation efficiency and memory gain with improved throughput and lower average response time.

- **Heterogeneity** A service colony hosts diverse inhabitants, each potentially utilizing different technologies and implementations. However, adherence to standard communication protocols ensures effective data dissemination throughout the ecosystem.

  Consistent with the experimental scenarios, the splitting function operates independently because it represents either the full or a subset of the original technical stack and, therefore, introduces no additional dependencies. Similarly, the merging function can operate effectively within heterogeneous environments, provided that the deployment server supports the technical stack required by the merging service.

- **Scalability** By integrating dynamic splitting and merging, the service colony enables individual scalability based on performance metrics. It supports proactive decision-making and predictive scaling initiated by its inhabitants, reducing maintenance costs and ensuring that resource allocation remains aligned with actual system demands.

5.4.2 Challenges

The challenges associated with the design, implementation, and maintenance of service colonies are yet to be fully understood and studied, but the following challenges are identified while implementing and validating our service colony prototype (Section 5.3):

- **System complexity** A service colony is a dynamic, complex system composed of interacting components. Without hierarchical control or global coordination, even slight modifications to components and interaction patterns can significantly impact the system's overall behavior. This intricate relationship between low-level component behaviors and high-level system behavior complicates error identification and troubleshooting. Additionally, if the splitting of inhabitants is not controlled, the colony can proliferate excessively, increasing system latency. Although the system aims to optimize for latency, unnecessary adjustments can result in the opposite effect.

- **Verification and validation** To verify the correctness of a system, it is essential to test it under different conditions, for example, by simulating environmental changes. The distributed nature of a service colony complicates this process due to the vast number of possible scenarios the system can execute. New tools are required to simulate different workloads and environmental changes to test service colonies properly. Additionally, validating the correctness of system decisions before re-architecting is challenging due to the system's dynamic nature, which makes it difficult to predict outcomes and ensure stability in advance. Leveraging the expertise of system specialists, the training of the decision-making process through the integration of machine learning, evolutionary algorithms, and neural networks could enhance the verification and validation processes.

- **Dynamic updates** A service colony is a dynamic system. Adjustments within such a system can lead to temporary unavailability of certain functionalities. Therefore, sophisticated mechanisms are needed to manage these dynamic adjustments effectively, ensuring smooth operations during system re-architecting.

- **Heterogeneity** A service colony can host diverse inhabitants, each implemented using different technologies. To support this heterogeneity, standard interfaces and communication protocols must be established. Moreover, the integration of components implemented using different technologies complicates system development, testing, and monitoring. Furthermore, such a heterogeneous system is more vulnerable to security threats.

- **Persistence** A service-based system relies on databases and data caching layers for information storage. These components require non-trivial adaptations during the re-engineering of the system. Dynamic repartitioning and redesigning of databases can

lead to data replication, which may result in data inconsistencies. In contrast, inadequate data partitioning may lead to performance bottlenecks during periods of high system load, ultimately diminishing the overall performance of the service colony.

- **System updates and change requests** System updates are mandatory to comply with industry standards, and customer change requests are inevitable. Implementing these changes in a distributed system is a complex task. A robust system state management process must be in place to handle updates effectively. Automatically persisting the system state before and after adjustments is essential for maintaining system operations. Additionally, a service colony must be equipped with comprehensive mechanisms for the automatic deployment of services resulting from the splitting of colony inhabitants and for managing continuous delivery and integration pipelines.
- **Re-engineering** Existing systems that wish to benefit from the advantages of service colony architecture need to be re-engineered accordingly. A systematic process for re-engineering software systems into service colonies must be defined to facilitate the migration of legacy systems or existing service-based systems to this new architectural style.

5.5 Implementation and Reproducibility

The Service Colony prototype is implemented as a distributed Java Spring Boot application and is publicly available at `https://github.com/thakshilad/ServiceColonyResources`. The implementation realizes the subsystem architecture described in Section 5.3. The system requirements for running the Service Colony prototype require Java 17 or higher and a MySQL database instance. The Service Colony implementation is organized into three components: TodoApp, LoadBalancer, and RequestGenerator. All three components are built using Maven with embedded application servers, which does not require separate server installation for deployment.

5.5.1 System Components

The *TodoApp* is the representative microservice application comprising three services: a todo item creation service, a user registration service, and a todo item inquiry service.

Each service exposes the RESTful API endpoint and is deployed as two microservices, as illustrated in Fig. 5.5. The TodoApp connects to a shared MySQL database instance; the schema is provided as a SQL script *TodoAPPDBScript.sql* in the public repository. The *LoadBalance* is the managing subsystem that implements the request dispatching of the TodoApp. Furthermore, it monitors the queue size fluctuations and distributes the knowledge to TodoApp. The *RequestGenerator* serves as the user simulator, generating the workloads by sending requests to the LoadBalancer at a configurable rate and capturing the request and response timestamps for all three service types. The request/response time statistics produced by the RequestGenerator are stored in Excel files and used for the performance evaluation reported in Section 5.3.

The LoadBalancer implements four operating modes used for the experiment scenarios described in Section 5.3. The configurable property *operatingMode* defined in the *application.properties* file is used to distinguish the operational scenarios. Moreover, LoadBalancer configurations specify the primary and secondary IP addresses that are used for the TodoApp deployment. The RequestGenerator configuration specifies the LoadBalancer URL.

5.5.2 Reproducibility

All resources required to reproduce the experiments reported in this chapter are publicly available at `https://github.com/thakshilad/ServiceColonyResources`. The repository is organized into four folders whose contents collectively cover all stages of the experiments. TodoApp, LoadBalancer, and RequestGenerator contain the source code for each component. The *Results* folder contains the complete experimental results, including response time statistics and memory consumption logs produced during each experimental scenario.

5.6 Summary

This chapter proposed the concept of a *service colony*, an architectural style for developing software systems as groups of autonomous, interacting software services. Each inhabitant service in a colony is driven by its aim to deliver services to its users, either external users of the system or other inhabitants of the colony. Based on their past performance and proactive

observation of the environment, inhabitants can proactively decide to self-replicate, split into multiple services, or join with other inhabitants to ensure high-quality service delivery. In this way, the overall service colony system can adapt to changing workloads by either shrinking its footprint during periods of low workload or scaling specific high-demand functionality during workload bursts. By performing such adaptations, the system aims to minimize resource utilization while maximizing the quality of the delivered services over time. A service colony is a bottom-up complex system characterized by numerous interacting components that induce system-level behavior. Consequently, service colonies aim to inherit the benefits of complex system architectures, including resilience, robustness, adaptability, scalability, and distributed control. The performance of a service colony is validated using a proof of concept. The experimental scenarios demonstrated that service colonies produce improved results, regardless of the effort required for system restructuring. Furthermore, it is feasible to transition from one steady state to another without restarting the services while maintaining the system's performance.

Both Chapter 4 and Chapter 5 rely on runtime execution data as the primary input to their respective decision-making processes. Chapter 4, *MIST* framework, relies on runtime execution traces to identify microservice boundaries, which require reliable records of method invocations. Service colonies monitor per-service execution behavior to discover autonomous adaptation decisions, which require continuous and structured observation of runtime events. In both cases, the utility of the execution data depends on how it is collected, structured, and represented. Inconsistent and non-standard log formats increase the complexity of data processing and limit interoperability and reusability. As established in the background (Chapter 2) and the literature review (Chapter 3), no existing standard or conceptual model provides a unified representation of software execution event data. The following chapter addresses this limitation by presenting a unified conceptual data model for software execution event data.

Chapter 6

A Unified Model of Software Execution Event Data

Software logs capture rich information about the behavior of software systems. However, these logs remain largely incompatible with analytical techniques such as process mining and specification mining, which depend on structured event logs. Instead, these domains typically rely on event logs, which are commonly obtained through recording the dynamic behavior of software systems. While process-aware information systems inherently produce event logs, non-process-aware systems lack this capability, thereby constraining their potential to leverage event log-based analysis techniques. To bridge this gap, we introduce the Software Execution Event Data (SEED) model, a conceptual model that formalizes event logs generated by software systems. We examine the dimensional data modeling features of SEED and demonstrate that explicitly structured event representations yield significant advantages in execution data analysis. Finally, we validate the expressiveness and practical applicability of SEED by showing its ability to represent common compliance specifications used in software execution data mining.

This chapter is derived from:
Thakshila Imiya Mohottige, Artem Polyvyanyy, Colin Fidge, Rajkumar Buyya, Alistair Barros, A Unified Model of Software Execution Event Data, *Submitted to Journal of Systems and Software (JSS), May 2026.*

6.1 Software to Event Logs for Execution Data Mining

Software systems are primarily composed of static source code. However, the runtime behavior of the software systems cannot be fully understood through source code alone, particularly in the presence of advanced features such as polymorphism and runtime data binding [63]. To capture dynamic behavioral aspects, software systems generate logs during regular operations. These runtime logs provide valuable insights into operational characteristics, uncover system behavior, facilitate real-time performance monitoring, and are essential for the success of a software system [63, 159]. Despite the richness of behavioral information contained in log files, their direct use in analysis is often hindered by the absence of standardized formatting, making automated processing and interpretation challenging [159].

Software runtime data analysis can be classified as active learning and passive learning. Active learning focuses on an interactive approach based on queries to the system, whereas passive learning uses execution logs to generate behavioral models [159]. Passive learning approaches use K-Tail algorithms, first defined by Biermann and Feldman [165, 183]. However, these techniques are limited in their ability to explicitly capture concurrency, often resulting in overly complex models that are difficult to interpret and analyze.

As discussed in Section 2.4.5, process mining is an active discipline used for process discovery, conformance checking, and process enhancement. It connects data science and data mining [70], typically starting with an event log [71] as its primary input. When applied to software systems, this approach is known as software process mining, which leverages software execution data to uncover and analyze system behavior [161]. Process models of software systems can be discovered by employing process mining in software systems. It can produce flat, hierarchical, and behavioral process models, offering a more structured and comprehensive representation of software behavior. This is particularly useful, as event logs enable the discovery of such models without requiring prior knowledge of the system [165]. Furthermore, conformance checks are used to compare event logs of the same process and extend or improve an existing process [71]. Execution data typically consists of execution traces, where each trace is mapped to a use case (e.g., a user session or request), and method calls are treated as events for building the traces. Unlike traditional process mining that often results in flat models, software process mining aims to uncover interactions among components and capture hierarchical behaviors through

nested method calls. Process mining can be used both backward-looking, to identify the root causes of bottlenecks, and forward-looking, to predict process execution and provide recommendations to reduce the risk of failures [184]. By applying process mining techniques, software systems can improve performance and reliability through enhanced operational transparency, the identification of inefficiencies, data-driven decision-making, and the evaluation and audit of the software processes [71, 185, 62].

Specification mining is used to discover formal or semi-formal specifications from existing software artifacts. It discovers the temporal, performance, and data dependencies that the program adheres to during execution [186]. By analyzing logs generated from instrumented software systems, specification mining infers general program rules that help to refine the system specifications and identify potential errors [186, 187]. It is widely used to understand undocumented software behavior, testing and verification, anomaly detection, analyze legacy software systems, and reverse engineering [173, 171, 186, 187]. To facilitate this analysis, software logs are often transformed into event logs that represent ordered sequences of events [172]. Hence, the extraction of event logs from software logs plays a crucial role in advancing research in specification mining.

As discussed in Section 2.4, PAIs record events (e.g., ERP, CRM) that facilitate generating event logs during process execution [61, 60]. However, the majority of software systems are non-process-aware information systems (e.g., legacy systems) that could benefit from applying mining operations [61], despite not being designed for such analysis. However, data quality, including the lack of well-structured and well-defined data, has been identified as a key challenge [62].

An *event* comprises three fundamental attributes: the name of the *activity* that triggered the event, the identifier of the *case* in which the activity that triggered the event was executed, and the *timestamp* at which the event was recorded [70]. An *event log* is a collection of events, grouped in traces, where a *trace* is the ordered, by timestamps, sequence of all events from the same case. XES [35] serves as a standard for event logs widely used for process mining and specification mining.

Software systems produce logs that are best suited for system debugging and performance analysis rather than for constructing abstract process models. Therefore, establishing a unified data model for software logs is essential to enable the application of mining techniques. Mapping software logs to an event log format is crucial to extending mining techniques to non-process-aware software systems. As identified in Section 3.3, the pri-

mary challenge of this transformation is a lack of standard in the structure and content of log data [33]. To address this gap, this chapter formalizes a conceptual data model to collect runtime execution data, aiming to improve the applicability of mining techniques in non-process-aware software systems. The resulting logs can be effectively used for software analysis tasks, including microservice migration and self-adaptation.

The remainder of this chapter is organized as follows: data extraction and analysis methodology to formalize the SEED model(Section 6.2), identified log properties and proposed SEED model(Section 6.3), compatibility and feasibility of the SEED model(Section 6.4), SEED compliance checking(Section 6.5), implementation of the SEED model to XES conversion(Section 6.6), and summary of the chapter(Section 6.7).

6.2 Data Extraction and Analysis Methodology

This section describes the data extraction methodology from the 50 primary studies identified in the systematic review presented in Section 3.3 to identify application areas utilizing software logs to the conversion of event logs. The methodology is formulated to address three research questions:

- DMQ1) What problems were solved using software logs to event log conversion in the field of process and specification mining?
- DMQ2) What properties in the software logs are used in process and specification mining applications?
- DMQ3) What relationships are observed between entities derived from software logs?

For each selected study, data were extracted to address the three research questions. Specifically, the problem context and application domain (DMRQ1), the software log properties and attributes used as inputs (DMRQ2), and the structural relationships identified among extracted entities from software logs (DMRQ3). The findings for DMRQ1, which address the problem domain of software log-to-event log conversion, are presented in Section 3.3.3, as they constitute a survey of existing work rather than a contribution of this chapter. The extracted data for DMRQ2 and DMRQ3 were iteratively analyzed to

identify usage patterns and entity relationships to form the design of the SEED conceptual model presented in the subsequent sections.

6.3 Results: From Software Logs to SEED Model

This section presents the findings of the data extraction and analysis, addressing DMRQ2 and DMRQ3. First, specific properties of the software logs used in the reviewed studies are identified and categorized (DMRQ2). The relationships among the extracted properties are formalized into the SEED model(DMRQ3).

6.3.1 Software log Properties Used For Mining Applications (DMRQ2)

The reviewed studies reveal that software logs are not uniform in structure and content. Based on the application domain, different subsets of log properties have been used for analysis. As illustrated in Figure 6.1, the properties used in software logs to event generation formalize a three-layer hierarchy, where each successive layer represents a specialized set of attributes. The most fundamental layer comprises *method-call information*, which is used in all application domains. The intermediate layer introduces *execution frequency data*, in addition to method-call information, for applications such as microservice identification, legacy system maintenance, runtime execution analysis, and behavior discovery. These applications require distinguishing between frequent and infrequent execution paths to produce accurate and representative models. The top-most or most specialized layer incorporates *time constraints*, for which precise timing data is required to identify behavioral bounds and ordering constraints.

The specific log properties used across the reviewed studies are summarized in Table 6.1. This includes temporal properties, such as time of invocation and method execution duration; context-identity properties, which correlate individual method calls to execution traces; and caller-callee relationships, which are most commonly used and primarily support the construction of directed call graphs for process discovery and model construction. Additional properties include execution metadata such as input and return types, thread identifiers, whether it is a constructor, and structural and hierarchical properties like depth and nested calls, collected from software logs.

Table 6.1: Software log properties used for mining.

Property	Description	Study	SEED
Timestamp Start/end time Duration	Timestamp is the time of invoking a method. Start and end timestamp defines the invoked time and completed time of methods and their difference used as duration.	[49], [166], [161], [188], [165], [156], [174], [8], [167], [157], [151]	✓
Session ID	Unique identifier of the session.	[49]	✓
Trace ID	Unique identifier of the request.	[166], [161], [189], [151], [190], [191]	✓
Method/ Callee method	Method being invoked - This can include the fully qualified name, method name, class, package, and input parameters and return type.	[49], [161], [188], [192], [165], [158], [156], [174], [193], [8], [194], [195], [189], [196], [172], [197], [159], [168], [198], [199]	✓
Method ID	Unique identifier assigned to method.	[161], [188], [158], [159]	✓
Method call instance/ Callee object/ target/ receiver	The unique hash code identifier for the runtime object of the class, which the triggered method belongs.	[161], [188], [192], [165], [158], [156], [8], [194], [159], [199]	✓
Class/ Callee class/ component/ Type of callee	The class of the invoking method.	[49], [161], [192], [165], [193], [196], [172], [197], [198], [199]	✓
Transaction type	Indicates the start, complete state of invoking method.	[161]	✓
Span ID/ Parent Span ID	Used by endpoint API calls to identify the caller and callee services.	[166]	✓
Caller class/ Type of caller	Class of the method initiates the call.	[196], [172], [197], [199]	✓
Caller object/ source/ sender	Unique hash code of the runtime class object, which initiates the call.	[188], [165], [158], [156], [8], [194], [159], [199]	✓
Caller method	Method initiates the call.	[188], [165], [158], [156], [159], [198]	✓
Input parameters	Input parameters of the method as a set of objects.	[156], [8], [199]	✓
Thread/Process identifier	Thread or process ID associated with the method being invoked.	[193], [8]	✓
Constructor context	Method calls that are made in the constructor.	[193]	✓
NestedCalls	The method being invoked contains other method invocations or not.	[198]	✓
Source code location	Source code position of the method - line number.	[199]	✓
Object collaboration	Sequence of method calls within a method.	[199]	×
Return parameters	Return parameters of the method being called.	[199]	✓
Created objects	Objects that are initiated inside the method being invoked.	[63]	×
Read/write access to objects	Read and write access to objects and fields inside the method.	[63], [168]	×
Affecting attributes	Updated attributes inside the method being invoked.	[63]	×
Temporal order of method calls	Ordered list of method calls sorted by timestamp.	[193]	✓
Method context as ancestors	Invoked methods within the method as a hierarchy.	[193]	✓
Thread read/write operations	Read/write access to resources.	[200]	×
Thread lock acquire/release	Acquired and released locks by the thread.	[200]	×
Depth	Distance from root to node in the hierarchical calling tree.	[201]	✓
Sequence of method calls	Invoked sequence of method calls as traces.	[195], [189], [202]	✓
User ID	User triggered the method call/ request.	[167]	✓

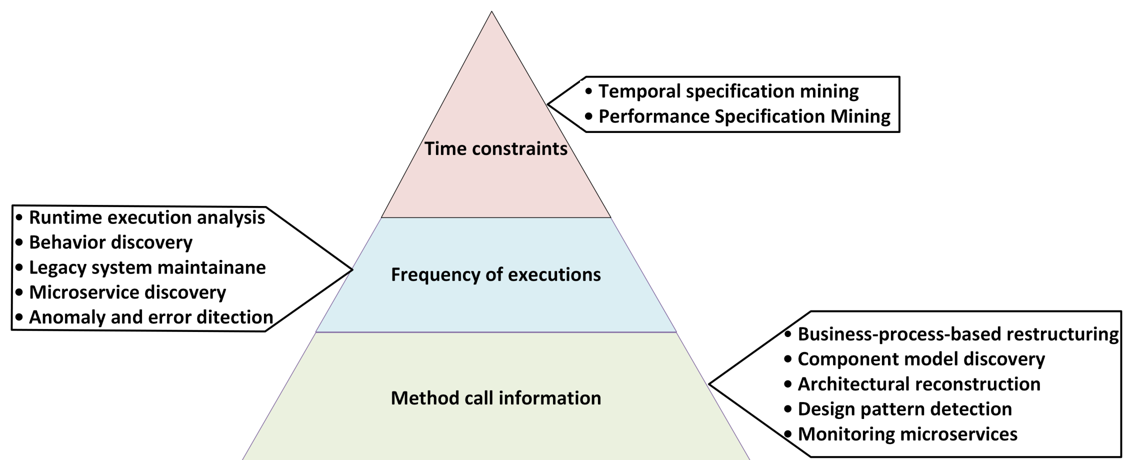


Figure 6.1: High-level usage of log properties in application areas.

The *SEED* column in Table 6.1 indicates whether each property is supported by the SEED model (see Section 6.3.2). The majority of the properties are supported by the SEED model. However, certain properties, such as object collaboration, object creation, read/write access to objects, thread-level read/write operations, and thread lock operations, are excluded. These were omitted because they operate at a lower level of abstraction than required for event log generation and are not consistently used across the reviewed mining applications. The following section Section 6.3.2 presents the SEED model and demonstrates how the log properties are organized into a coherent conceptual structure for software execution event data.

6.3.2 From Log Properties Relationships to SEED Model (DMRQ3)

The properties identified in DMRQ2 represent individual data fields observed across the review studies. These properties become analytically meaningful when organized into a coherent structure to capture their relationships. DMRQ3 addresses this by examining how these properties group into semantically coherent entities and formalize the SEED model, a conceptual data model for software execution event data. The model is based on the object-oriented programming, as the reviewed studies widely adopt object-oriented design.

6.3.2.1 Identifying SEED Entities

Properties that describe a static method definition: method name, return type, input parameters, constructor context, and nested method belong to the method entity. Properties that describe the static structure of a class: class name and class type belong to the class entity. The package entity captures a grouping of classes within the source code.

At the runtime execution level, properties related to runtime object instances on which a method is invoked, specifically, the caller and callee object hash codes, belong to the MethodInstance. Moreover, the properties that characterize the execution context of invocation, such as SpanID, Parent Span ID, which denotes the Caller deployment node IDs, and callee deployment IDs included in the MethodInstance. Moreover, MethodInstance includes the call stack depth and calling order, which define the position of the method instance within the execution trace. The runtime instance of a class is represented by the ClassInstance entity, identified by its object hash code. The Thread entity captures the execution thread associated with a method invocation, carrying the thread ID, read/write operation, and lock acquire/release properties. In a distributed deployment, the Node entity captures remote calls between nodes and is uniquely identified by the span ID. Moreover, event log-specific entities, Trace and Event, were incorporated into the entities to uniquely identify traces and their respective events. A summary of these entities and attributes is presented in Table 6.2.

Table 6.2: SEED model nodes and attributes.

Entity	**Attribute**
Trace	Trace ID, Session ID, User ID
Event	TimeStamp, Transaction type, Source code location
Method	Method name and ID, Input parameters, Return type Constructor context, Nested method
MethodInstance	Callee object hash code, Caller object hash code, Span ID, Parent span ID, Depth, Calling order
Class	Class name, Class type, Package name
ClassInstance	Object hash code
Package	Package name
Thread	Thread ID, Read/write operation, Lock acquire/release
Node	Span ID

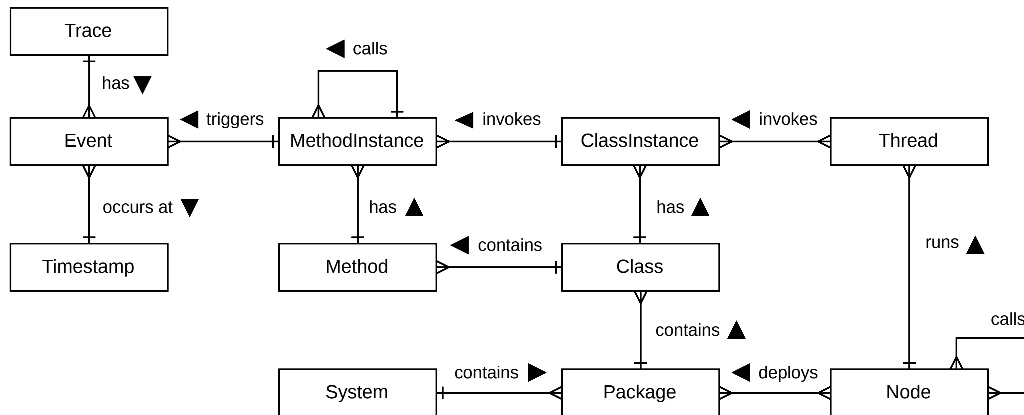


Figure 6.2: SEED model.

6.3.2.2 SEED Model Structure

The relationships among the identified log entities are illustrated as a data model (SEED) in Figure 6.2. The model supports both distributed tracing and single-deployed component tracing. The SEED model is organized into five conceptual layers: the source code layer, the runtime layer, the event layer, the trace layer, and the interaction layer, each representing a distinct stage in the lifecycle, from static software structure to recorded execution. The source code layer is the grouping of packages, classes, and methods. The runtime layer represents the dynamic object instantiation and execution threads of the source code. The event layer defines how runtime behavior is recorded and structured to formulate an event log. It focuses on atomic executions, similar to method-level invocations. The trace layer generates a sequence of records for events associated with a single request, transaction, or execution thread. The interaction layer aggregates the communication between services and components to capture the structural relationships. These five conceptual layers capture all the application domains identified in Section 3.3.3.

The software system contains multiple packages, where each package can be a collection of classes. The class can contain multiple methods. Each class creates an instance that is the runtime object of the class. The source code class is a blueprint. An instance is the concrete occurrence of that blueprint in memory. Similarly, each method has an instance associated with the respective class instance, which is the runtime execution of a method. Since one class contains many methods, the same relationship applies to instances, where

one class instance can contain multiple method instances. Moreover, one method calls another method, which is the caller-callee relationship of the method invoking, except for the main method. Hence, the runtime method instance can call another method instance. An instance of a method triggers an event. An instance of a method triggers multiple events, which include the start and end of the method instance and the timestamp of occurrence.

A collection of packages produces the source code, that is deployed in a node. A deployment node consists of multiple threads. One thread can access multiple class instances, and multiple threads can access one class instance. In a distributed setting, there are multiple nodes. These nodes interact via remote method calls. The caller-callee relationships across the nodes are used to identify component interactions. Hence, a node consists of a caller-callee relationship.

Typical event logs consist of multiple traces, which are groupings of events. An instance of a main method and all the method instances in the call stack triggered by that main method instance are considered as a trace. Hence, all the events it generates belong to the same trace, and ordered by the timestamp.

The SEED model answers RQ3 by systematically organizing the properties identified in RQ2 into coherent entities whose relationships capture the complete lifecycle from the static software structure through dynamic runtime interactions. The compliance and compatibility of the SEED model are discussed in the following sections.

6.4 SEED Model Compatibility & Feasiblity

Although the SEED model defines the structural relationships between software log entities, it is essential to confirm that SEED is analytically well-founded and compatible with existing mining tools. Hence, this section focuses on the compatibility and feasibility aspects of the SEED model.

6.4.1 Dimensional Modeling

From the data modeling perspective, the Event entity in SEED can be understood as the fact table of a star schema. A star schema is a way to organize data in a database that is specifically designed for analysis rather than storage. There are two types of tables: the fact table and the dimension table. The fact table sits at the center, recording individual

measurable events, whereas dimension tables surround the fact table like the points of a star, providing descriptive context for the fact table [203].

Each event record captures a discrete, measurable occurrence of a method invocation and references surrounding dimension-like entities that provide descriptive context: Timestamp (when), MethodInstance(Who), Thread (which resource), and Node (which deployment context). This dimensional structure reflects the inherent analytical nature of event logs, which are consumed by mining techniques in a manner analogous to how fact tables are consumed by Online Analytical Processing (OLAP) queries. Dimensional modeling was specifically designed for analytical workloads, and its benefits, including query efficiency, scalability, and interpretability, are well established in the data warehousing literature. Structuring software execution data according to this model, therefore, ensures that SEED is not merely an ad hoc collection of log properties, but a principled representation optimized for the analytical tasks it is intended to support. This dimensional foundation directly substantiates the claim that event data benefits significantly from structured and explicitly defined representation.

6.4.2 Compatibility with XES standards

The logs generated based on the SEED model are compatible with (eXtensible Event Stream) XES [1] standards. Figure 6.3 depicts the mapping between the SEED model and the XES standards. Blue color attributes are mandatory requirements in the XES standards. The remaining properties are defined as optional attributes.

A sample execution is illustrated in Figure 6.4. All event generation points from $e1$–$e12$ require having log entries with mandatory attributes and necessary optional attributes based on the system under analysis. The log entries are then converted to XES standards for the mining software system. After the execution of this Fig. 6.4, it will provide trace t=e_1,e_2,e_3,e_4,e_5,e_6,e_7,e_8,e_9,e_{10},e_9,e_{10}, e_{11}, e_{12}. A sample of an XES converted file is depicted in Figure 6.5.

[1] https://www.xes-standard.org/

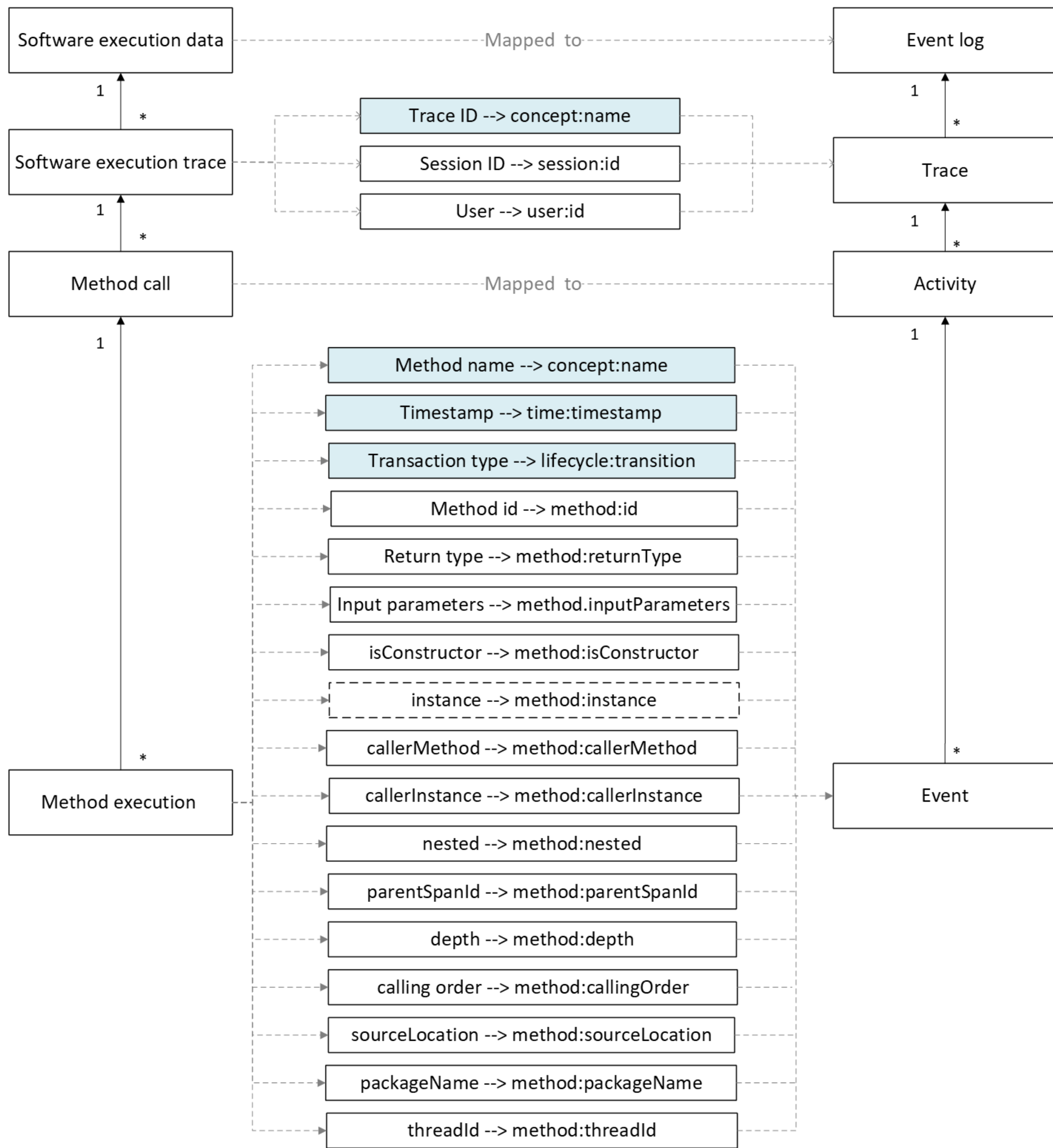


Figure 6.3: SEED model mapping to XES standard.

```
Class App{
  method main() {        main method start  e1
    X x1 = new X();      x1 # 5704s9v
    x1.a();
    x1.b();
  }                      main method end e12
}
```

```
Class X{
  method a(){              method a start e2
    Y y1 = new Y();        y1 # – e4959sc
    y1.c();
  }                        method a end e7
  Method b(){              method b start e8
    Y y2 = new Y();        y2 # – d4985es
     if(j=0; j<2, j++){
       y2.e();
     }
  }                        method b end e11
}
```

```
Class Y{
  method c(){              method c start e3
    d();
  }                        method c end e6
  Method d(){              method d start e4
  }                        method d end e5
  Method e(){              method e start e9
  }                        method e end e10
}
```

Figure 6.4: Sample program to apply SEED model.

```xml
<trace>
<string key="concept:name" value="Trace1"/><string key="sessionID" value="8204714096138912019" key="userID" value="12345"/>
    <event>
    <string key="concept:name" value="App.main"/>
    <string key="lifecycle:transition" value="start"/>
    <date key="time:timestamp" value="2025-08-01T12:51:20.465500"/>
    <string key="caseId" value="1722473480203471700"/>
    <string key="instance" value="d8730se"/>
    </event>
    <event>
    <string key="concept:name" value="X.a"/>
    <string key="lifecycle:transition" value="start"/>
    <date key="time:timestamp" value="2025-08-01T12:51:20.465501"/>
    <string key="caseId" value="1722473480203308200"/>
    <string key="instance" value="5704s9v"/>
    <string key="callerMethod" value="App.main"/>
    <string key="callerInstance" value="d8730se"/>
    </event>
    ......
</trace>
```

Figure 6.5: Sample XES output for the program.

6.4.3 Feasibility and Adoption Pathway

The practical feasibility of SEED depends on two distinct adoption scenarios: greenfield systems developed from scratch, and existing systems already in operation.

For greenfield systems, adopting SEED requires only that the logging mechanism be designed to capture the mandatory and relevant optional attributes defined in the SEED model. Then the execution logs will be natively structured according to the SEED model and can be converted directly to XES without any intermediate transformation. To support this pathway, the SEED log-to-XES conversion tool [2] was developed. The tool accepts software execution logs that conform to the SEED model and converts them into XES-compliant event logs. It provides configurable field-mapping mechanisms that align existing log field naming conventions with SEED attribute definitions, accommodating variation in how different systems label the same conceptual data. For example: mapping a

[2]https://github.com/thakshilad/SEED

system's *requestId* field to SEED's *traceId*, or *invokedMethod* field to SEED's *methodName*. This configurability ensures that greenfield systems do not need to adopt a rigid log format, but rather a flexible structural convention that meets the mandatory requirements. The resulting XES files are compatible with established process mining tools, including Disco [3] and ProM [4].

Existing systems already generate structured logs for debugging and performance monitoring, which are typically incompatible with the SEED model. For small-scale systems, log rewriting or re-labeling is a viable pathway. The configurable field mapping in the SEED tool supports this pathway by defining the correspondence between the existing log schema and the SEED model. For large-scale systems with high code volume, log rewriting becomes impractical, and instrumentation is the more appropriate pathway. This involves augmenting the running system with a monitoring probe that intercepts method invocations at runtime and emits structured log entries directly. Kieker [5] (refer Section 2.3) is a widely adopted instrumentation framework that supports this approach. The feasibility of this approach is demonstrated by implementing the SEED tool with Kieker-to-XES conversion, which automates the transformation of Kieker log files into XES-compliant event logs.

The two components in the SEED tool, the SEED log to XES conversion and the Kieker log to XES conversion, demonstrate that the SEED model is not only a theoretical construct but a practically applicable framework.

6.5 Compliance Rules and SEED Expressiveness

The key measure of the practical value of a conceptual model is its ability to satisfy the formal requirements that existing techniques impose on their studies. To evaluate this in SEED, compliance specifications were systematically extracted from the reviewed studies identified in Section 3.3. Compliance specifications are formal or semi-formal rules that define the structural and behavioral properties that software execution event logs must satisfy in order to support process and specification mining. The compatibility of the SEED model with each specification was assessed in three levels: *Directly supported* if the

[3]https://fluxicon.com/disco/
[4]https://promtools.org/
[5]https://kieker-monitoring.net/

SEED model contains the entities and attributes required to satisfy it without additional computation, *Derivable* if it can be computed or inferred from SEED attributes through post-processing without additional data, *Not supported* if it falls outside the intended abstraction level of SEED and cannot be satisfied from the current model structure.

The 42 compliance specifications were identified in the reviewed studies. Table 6.3 presents the identified compliance rules and the compatibility of SEED with the respective rules. Among the 42 compliance rules, only three specifications are explicitly not supported by SEED (36, 41, and 42). Rule 36, concerning object roles, is defined as a set of method signatures invoked during a collaboration, operates at an abstract level above individual method calls, and is more appropriately addressed as a post-processing analysis task. Rules 41 and 42, concerning thread read/write operations on shared resources and thread lock acquisition and release, respectively, represent low-level concurrency monitoring properties that fall below the abstraction level at which event log generation operates. These properties are relevant to runtime verification frameworks rather than process or specification mining tools, and their exclusion is consistent with the scope boundaries of the SEED model. Hence, the compliance analysis demonstrates that SEED provides broad and principled coverage of the analytical requirements identified in the literature, with clearly justified boundaries where specifications fall outside its intended scope.

Let the software log be S_L, event log be E_L, method call m, and event be e, $\#_n(e)$ is the value of attribute n for event e, and $\#_n(m)$ is the value of attribute n for method m. COM is the component of the software system. SD be the software execution data.

Table 6.3: Compliance specifications

✓- Directly supported ~ - Derivable × - Not supported			
ID	Description	Study	Status
1	The software event log is a finite multi-set of traces.	[161], [164], [8], [151],[201]	✓
2	A trace is a finite sequence of unique events.	[161], [188], [165], [201], [202], [198]	~
3	Each event can specify number of attributes.Mandatory attributes are an event name and timestamp. Events are ordered by logging time in each trace.	[201]	✓

4	Any event in the method call universe must include: method call ID, instance, class, caller method, caller class, caller object, start time, end time, transaction type, and owning component.	[161], [165], [158], [204]	✓
5	For any method call, nested method calls should be able to capture.	[198]	✓
6	Each event should have a unique classifier, which is method call ID and transaction type.	[161]	✓
7	Event log is consistent if and only if there are two events $e\&e'$ such that $\#_{method}(e) = \#_{method}(e'), \#_{instance}(e) = \#_{instance}(e'), and$ $\#_{transactionType}(e) \neq \#_{transactionType}(e')$.	[161]	~
8	For any two methods m_i and m_j, if $\#_{call}(m_i) = m_j$, then m_i is the caller. If $\#_{call}(m) = \emptyset$, then m is *main* method.	[188], [165], [158], [156], [159]	✓
9	Every method call is associated with a method with an object instance, and each instance originates from a class.	[188], [156]	✓
10	Each method call is mapped to a start and end timestamp.	[188], [204], [156]	✓
11	Event class names are unique identifiers. If two event classes $C1\&C2$ have the same names $C1.name = C2.name$, they represent the same set of events $C1.events = C2.events$. If $C1$ is a super class of $C2$, then $C2.events \subseteq C1.events$.	[201]	~
12	Each method call is linked to a component instance, whose execution data includes all calls on objects of its classes.	[188], [165], [158], [159]	~
13	Component instance set is all objects instantiated from its component. No shared objects and connections across objects belong to different component instances.	[188], [165]	✓
14	The union of all component instances is equal to the entire object set.	[188],[165]	~
15	The object on which the method call m executes must be a member of the component instance to which m is assigned.	[188]	~
16	Event ordering via directly followed, overlap, and contains relationships can be used to infer direct succession, causality, concurrency, choice, and nesting.	[166], [161]	~
17	Frequency ratios for directly follows, overlaps, and containment relations between method calls can be computed to quantify their occurrence and dominance within the event log.	[161]	~
18	Method interactions can be modeled as an object interaction graph, where nodes represent class object instances that execute at least one method, and edges represent method calls between these objects.	[188],[165], [158]	~
19	The top-level method set of a component can be derived as the set of methods that are called by methods belonging to other components.	[158]	~

20	The main event set comprises method calls triggered by external components. The main event log includes, for each trace, only the method calls belonging to this set.	[188],[165]	~
21	Nested event set contains two events *e*&*ne* belongs to the same trace, where the number of caller contexts of *e* equals the number of callee contexts of *ne*, which can be used to represent the internal structure of a component.	[188],[165]	~
22	The invoked event set consists of method calls directly invoked by the nested event set. These represent the child events of the corresponding nested method calls.	[188],[165]	~
23	A hierarchical software event log is constructed by starting with main events and recursively adding their callee events as child nodes, forming a tree-like structure.	[188],[165]	~
24	Candidate interface set of a component, identifies the methods accessed by external components or objects. A method *m* of component *C* is a top-level method if it is invoked by a method outside *C*. For each external caller *n*, the set of top-level methods invoked by *n* forms a candidate interface for *n*. The collection of all such sets, one per external caller, forms the candidate interface set of component *C*	[158], [159]	~
25	Method calls should associate with input parameter object sequences such that for each method call, there is a sequence of input parameter objects of the instance of the method	[204], [156]	✓
26	Given software execution data, it should be able to derive: the set of executed classes, executed method set of a class, input parameter set of every executed method, and invoked method set of each method.	[204], [156]	~
27	To enable process mining, events generated from a software system must include the following information: (i) the start and end timestamps of a method execution, (ii) the node where the event was generated, (iii) execution thread, (iv) the join point that triggered the event, (v) communication resource details (e.g., TCP/IP connection, IP address, and port), and (vi) the instrumented join point.	[63]	~
28	In a distributed setting, a node (web server) has multiple threads (node instances executed by threads). Each communication resource (channel) belongs to two different nodes. For any two different nodes, their set of communication resources and node instances must be disjoint.	[63]	✓

29	Given a communication resource and a communication interval, the set of events associated with the resource must be derivable.	[63]	✓
30	Two events $x, y \in E_L$ are directly related iff either: i) x and y generated by same thread. x executes within the duration of y. ii) Event y starts before event x, and the resources of x and y refer to the same communication channel.	[63]	~
31	Number of occurrences of a particular sequence in the event log –support, can be derived from the event log	[167], [205]	✓
32	Likelihood of another event occurring after a predefined sequence in the event log–confidence, can be derived from the event log.	[205]	✓
33	All events in a sequence are indexed by their position in the sequence.	[205]	✓
34	For any method, the interface set invoked by the method is defined as the interaction model of the method.	[159]	✓
35	The sequence of method calls within the execution of a method, m, and their receiver objects, form an object collaboration of the method.	[199]	✓
36	Objects share common method sets, which creates the concept of a role—defined as the set of methods invoked on an object during a collaboration. For an object o, its role is the set of method signatures s called on o within a call sequence S.	[199]	×
37	Given a hierarchical event log, the following properties should be derivable for each node: its depth (number of steps from root), parent node, event class, class of the event, set of leaf nodes, and the path from the root to the node.	[201]	~
38	Hierarchical Software Execution Event Log, $HLOG = (mainLog, Mapping : event-> subLog, layer)$. The main event log contains the top-level execution events. Mapping contains the association of nested events in the main log to the sub log. Layers indicate the depth or number of nesting levels in the event hierarchy.	[198]	✓
39	An interaction log can be generated from software log data using the set of objects and methods, capturing all possible finite sequences of method invocations.	[8]	✓
40	Hierarchical interaction model is a five-tuple with–set of objects, containers, object to container mapping function, set of messages, and allowed interactions.	[8]	✓
41	The events generated through instrumented concurrent program should include $read(T, R)\&write(T, R)$ which denotes the read and write operations performed by thread T on shared resource R.	[200]	×

42	The events generated through instrumented concurrent program should include $acquire(T, L)\&release(T, L)$ representing thread T acquiring and releasing lock L.	[200]	×

6.6 Implementation and Reproducibility

The implementation related to this chapter is publicly available at `https://github.com/thakshilad/SEED/`. The XES converter contains the required implementation for converting software logs/Kieker logs to XES format. The application requires Java 22 or higher version.

6.6.1 XES Converter

The XES converter is the primary implementation of SEED. It performs two operations: converting Kieker logs to XES format and converting general software logs to XES format. The format of the application input is specified through the *app.inputFormat* property in the application configuration file. This dual input support enables the converter to process both legacy Kieker instrumentation output and logs generated by directly applying the SEED data model. The output format is specified through the *app.outputFormat* property, to specify XES format or plain text format. Since the naming convention used for software logs varies across systems, the XES converter allows for configuring property mapping between SEED model and its corresponding field name in the software logs. This design enables the converter to process logs from systems that use different naming conventions without modifying the source code, directly realizing the interoperability objective of the SEED model.

6.6.2 Reproducibility

All resources required to reproduce the experiments reported in this chapter are publicly available at `https://github.com/thakshilad/SEED/`. The repository is organized into three folders. The XES converter contains the complete Java Spring Boot source code of the XES converter. Input folder specified sample input logs. The output file provides the final output of the system as a structured event representation in XES format. The

ExtractedData.xlsx file contains the complete dataset extracted during the studies reviewed for SEED model construction.

6.7 Summary

This chapter presented the Software Execution Event Data (SEED) model, a conceptual data model for representing the runtime execution data of object-oriented software systems in a form suitable for event-log-based mining.

The chapter presented the motivation for the unified event data model, identifying the absence of a standard structure for software execution logs as the primary constraint to applying process mining and specification mining techniques to non-process-aware software systems. A systematic review of 50 studies from process mining and specification mining was analyzed to formulate the SEED model, guided by three research questions addressing the problem domains utilizing software logs to event log conversion (DMRQ1), the properties utilized in those applications (DMRQ2), and the structural relationships among those properties (DMRQ3).

The review revealed six broad categories of problem domains addressed using software event logs: architecture reconstruction, legacy system maintenance, runtime behavior analysis, system monitoring, anomaly detection, and specification mining, and identified 29 distinct log properties spanning temporal, identity, caller-callee, execution context, and structural dimensions. These properties were systematically organized into nine semantically coherent entities: Event, Trace, MethodInstance, Method, ClassInstance, Class, Package, Thread, and Node, whose relationships constitute the SEED model, with the Event entity serving as the central fact table and surrounding entities functioning as analytical dimensions, providing a principled foundation for event data representation.

The compatibility of SEED with existing analytical requirements was evaluated through a compliance analysis of 42 specifications extracted from the reviewed studies. The analysis demonstrated that SEED directly supports 20 specifications and satisfies a further 19 through derivation, achieving a combined coverage rate of 93%. The three unsupported specifications fell outside the abstraction level at event log operations and were explicitly justified. Furthermore, the mapping from SEED model data to XES standards was defined to ensure compatibility between the proposed model and event log standards.

The practical applicability of SEED was demonstrated through an end-to-end feasibility

pipeline comprising software log-to-XES and the Kiker log-to-XES conversion tool, which is publicly available in the study repository. Generated XES logs are directly compatible with established process mining tools, confirming that SEED bridges the gap between raw software execution data and the structured event logs. Finally, these contributions establish SEED as a structured, empirically grounded, and practically applicable foundation for software system analysis using event-log-based mining techniques.

The chapter proposed the SEED model as the foundational data layer that underpins the execution-data-driven contributions of this thesis. Together with the *MIST* framework (Chapter 4) and Service Colonies (Chapter 5), SEED completes the three interconnected contributions of this thesis: performance-aware microservice identification from execution traces, an architectural style for self-adaptive microservices utilizing runtime execution data, and a unified data model that standardizes runtime execution event data representation. The following chapter concludes the thesis by reflecting the individual and collective significance, discussing the limitations and directions for future research.

Chapter 7

Conclusions and Future Directions

This thesis addressed three interconnected challenges: identifying performance-aware microservices in legacy monolithic systems using runtime execution data, the autonomous adaptation of microservice systems in dynamic operating environments, and the standardization of software execution event data to maximize its utility across software management activities. This chapter summarizes the contributions of the three research questions, identifies the limitations and directions for future research, and offers final remarks of the thesis.

7.1 Summary of Contributions

This thesis addressed three interconnected contributions, each addressing one of the identified research gaps and grounded on a shared foundation of runtime execution data.

Research Question 1: How can historical executions of a software system be used to help migrate it to a microservice system that exhibits better performance? Based on the systematic literature review of 117 primary studies on reengineering systems to microservices, the *MIST* framework addresses the gap in the use of software execution data and performance evaluation for identifying microservices. *MIST* is a semi-automated framework for identifying microservice boundaries in legacy monolithic systems by mining sequential patterns from historical execution traces. By analyzing low-level, repetitive, and consistent method invocation sequences captured by the Kieker monitoring framework,

MIST enables the systematic decomposition of monolithic applications into independently executable microservices that can be deployed separately while maintaining communication with the other system components.

The structural quality of the identified microservices was evaluated against established measures of coupling, cohesion, and modularity. *MIST* achieves the best values for coupling and cohesion among existing benchmark studies, demonstrating that sequential pattern mining over historical execution traces can produce small, highly cohesive, and loosely coupled service boundaries, which are essential for scalability, maintainability, and functional independence.

The empirical evaluation further compares the five identification factors (i.e., support, confidence, pattern length, average depth, and average execution time) impact on the throughput and latency of the resulting microservices across two case studies. The results demonstrated that support is a stronger predictor of performance decline, whereas average depth is a stronger predictor of improved performance, providing the first systematic empirical evidence that these parameters jointly influence the structural and runtime performance characteristics of the identified microservices. Critically, *MIST* empirically analyzes runtime performance using throughput and latency, providing evidence that microservice migration can deliver operational benefits rather than performance regression.

Research Question 2: How can a microservice system be effectively adapted in a dynamic environment based on its execution data? Based on the systematic literature review of 21 studies on self-adaptive microservices-based systems, the requirements for a decentralized, agent-based, and proactive decision-making system are identified. In response, Service colonies introduced a novel architectural style for developing software systems as groups of autonomous, interacting services. Each inhabitant service in a colony is driven by its aim to deliver services to users, either external users or other inhabitants, and can proactively decide to self-replicate, split into multiple services, or merge with other inhabitants based on past performance and proactive observation of the environment. Each inhabitant continuously generates and shares runtime execution events with the colony to support decentralized adaptation decisions. This bottom-up adaptive model enables the overall system to shrink during periods of low workload and scale specific high-demand functionality during workload bursts, minimizing resource utilization while maximizing service quality over time.

The effectiveness of the proposed approach was evaluated empirically and compared against a baseline system. The results demonstrated measurable improvements in response time and resource utilization, confirming that execution-data-driven decentralized adaptation delivers operational benefits. The evaluation further demonstrated that transitions between steady states can be achieved without restarting services while maintaining system performance. As a complex adaptive system, Service Colonies inherits the benefits of distributed architectures, such as resilience, robustness, adaptability, scalability, and decentralized control, while addressing the specific operational challenges of microservice-based deployments.

Research Question 3: How can the execution data of a software system be collected to maximize the utility of the data for the management of the system? By analyzing 50 primary studies that use software logs to construct event logs across six problem domains, namely architecture reconstruction, legacy system maintenance, runtime behavior analysis, monitoring, anomaly detection, and specification mining, 29 distinct log properties were identified and used to design SEED, a unified conceptual model for software execution event data. SEED organizes these properties into semantically coherent entities, for instance, Event, Trace, MethodInstance, Method, ClassInstance, Class, Package, Thread, and Node, structured across five conceptual layers: the source code layer, the runtime layer, the event layer, the trace layer, and the interaction layer. The model was designed with interoperability and integration with existing standards in focus, and was mapped to the XES standard to ensure compatibility with established process mining tools.

The SEED model was validated against 42 compliance specifications extracted from the reviewed studies. The validation demonstrated that SEED directly supports 20 specifications and satisfies a further 19 through derivation, achieving a combined coverage rate of 93% across diverse application domains. The three unsupported specifications fall outside the abstraction level of event log generation and are explicitly justified. An end-to-end feasibility pipeline comprising log-to-XES and Kieker-to-XES conversion tools confirms that SEED bridges the gap between raw software execution data and the structured event logs required for mining, maximizing the utility, interoperability, and reusability of runtime execution data for microservice identification, self-adaptation, and broader software system management activities.

7.2 Future Research Directions

The contributions of this thesis suggest several promising directions for future research in the domains of performance-aware microservices, self-adaptive microservices, and execution data analysis.

7.2.1 Performance-Aware Microservice Identification

The current microservice identification approach in this thesis is limited to the analysis of sequential patterns alone. Although sequential patterns provide insight into runtime behavior and service interaction boundaries, practical microservice migration involves database partition strategies, data ownership, distributed transaction management, and context-aware microservice boundaries. Future work can systematically examine how sequential patterns, when combined with these architectural constraints, influence both structural quality and runtime performance of the resulting microservices.

The performance evaluation and optimization capabilities of the proposed framework can be extended beyond throughput and latency. Future research can investigate multi-objective optimization approaches that consider scalability, reliability, efficiency, operational cost, security, and maintainability alongside latency and throughput-based metrics. Integration of optimization algorithms, machine learning techniques, and predictive analytics can support automated trade-off analysis and adaptive architectural decision-making, allowing microservice systems to achieve balanced performance across multiple operational objectives simultaneously.

Furthermore, enhancing the cost model for microservice identification to formulate a generalizable model that accounts for the impact of individual attributes on microservice quality remains a future direction. The current approach to microservice identification relies on a cost model that assigns equal weights to parameters. This limitation can be addressed by refining our cost model to enable more flexible and systematic identification of microservices.

7.2.2 AI-Assisted Microservice Code Generation

The current approach generates microservices by extracting code at the method level. Although this technique can isolate executable functionality, it does not fully address

the broader challenges of transforming a monolithic system into production-ready microservice code, and it remains a significant research challenge. In practice, microservice migration requires the automatic generation of communication interfaces and data transfer mechanisms, and the restructuring of source code to reflect contextual inter-class and inter-component relationships.

Future work can extend the proposed approach by incorporating Large Language Models (LLMs) together with Retrieval-Augmented Generation (RAG) techniques to support intelligent microservice code generation and refactoring. RAG can improve the code transformation process by providing the language model with contextual information retrieved from the original codebase, including related classes, methods, and architectural patterns. Instead of relying solely on the internal knowledge of the LLM, the retrieval layer can dynamically provide relevant source code fragments and architectural information, enabling the generation of context-aware microservices. Such an extension can advance microservice identification towards a fully automated, AI-assisted microservice migration pipeline, enabling intelligent code extraction, service generation, and the modernization of legacy systems.

7.2.3 Behavioral Equivalence of Microservice Systems

A critical open problem concerns validating behavioral equivalence during both microservice identification and self-adaptation. In microservice migration, the structure of the legacy system changed through the identification of microservices and their independent deployment. In adaptation of service colonies, the structure of the system changes continuously through splitting, merging, and replication operations. This thesis evaluates the operational performance of the resulting systems, but the behavioral consistency between the original system and the migrated or self-adapted system has not been systematically analyzed. Ensuring that the functional behavior of the system is preserved during architectural restructuring is critical to its reliability and correctness.

A promising direction is the application of conformance checking techniques from the field of process mining. Execution event logs generated from both the original system and the migrated or self-adapted system can be compared against reference process models derived from the original system, enabling the identification of behavioral deviations and functional inconsistencies. Future work can investigate automated behavioral consistency

validation pipelines that can be applied both as a post-migration quality assurance step in microservice identification and as a continuous monitoring mechanism in self-adaptive systems.

7.2.4 Service Colonies

The proof-of-concept presented in this thesis constitutes an initial step toward realizing the vision of service colonies, and several directions remain open for future investigation. An interesting direction is enhancing proactive decision-making by incorporating advanced forecasting techniques grounded in runtime execution data. Predictive models, including reinforcement learning, time-series analysis, and data-driven anomaly detection, can enable colony inhabitants to anticipate workload changes and initiate adaptations before performance degradation occurs, rather than responding to observed conditions.

To validate the results in a large-scale industrial setting, future research is needed. Future work can conduct comprehensive surveys with industry professionals and develop in-depth real-world case studies to evaluate the applicability, scalability, and practical relevance of service colonies in an operational environment. Integration of more sophisticated learning and decision-making algorithms into a real-world application would enable evaluation under realistic conditions. A systematic comparative evaluation of service colony against state-of-the-art self-adaptive systems using consistent benchmarks and evaluation criteria remains an important open research question.

Additionally, future work can develop a framework for an automated service deployment pipeline to address the current reliance on pre-deployed services, design a communication interface for generating communication between colony inhabitants and system users, and investigate colony inhabitant architectures and their interaction principles to identify configurations that maximize adaptability, resilience, and performance.

7.2.5 Adaptive and Privacy-Aware Logging for SEED

Two complementary directions for future research concern the practical deployment of the SEED model in production systems. First, investigating the impact of log granularity on system performance is an important open problem. Modern distributed microservice-based systems can generate large volumes of execution data continuously when instrumented according to the SEED model. Fine-grained logging and extensive instrumentation can

introduce significant runtime overhead, increasing storage consumption, data processing delays, and network overhead. Although SEED focuses on standardizing runtime execution data, the performance implications of varying logging granularity levels have not been addressed. Future research should investigate adaptive, performance-aware logging strategies that adjust the level of monitoring and data recording based on workload conditions, balancing the trade-off between analytical utility and operational efficiency by varying logging detail during abnormal behavior and stable execution periods.

Second, privacy and data governance considerations in runtime execution data collection have not been addressed in the SEED conceptual model. Execution data can contain sensitive information about user behavior, business operations, and system internals. Incorporating privacy-preserving mechanisms, including data anonymization, access control, and governance policies, into the SEED model and its associated log collection framework is an important direction for future work, particularly for deployments in regulated industries.

7.2.6 End-to-End Legacy Systems Modernization Pipeline

The three research directions addressed in this thesis, that is, microservices identification from execution traces (*MIST*), self-adaptive microservice systems (Service Colonies), and unified execution data representation (SEED), have been developed and evaluated independently. A significant open research direction is integrating these contributions into a unified, end-to-end legacy system modernization pipeline.

Such a pipeline can begin by instrumenting a legacy monolithic system with the SEED data model to collect standardized execution-event data. Historical execution traces accumulated over time would then be analyzed by *MIST* using sequential pattern mining to identify microservice boundaries. The AI-assisted code generation framework discussed above can automatically produce deployable microservice implementations. Finally, the generated microservices can be deployed as a service colony, enabling autonomous self-adaptation in response to runtime conditions. Integrating these stages into a coherent, validated pipeline for real-world legacy system modernization, from initial instrumentation through to self-adaptive microservice deployment, represents the most ambitious and potentially impactful direction for future work arising from this thesis.

7.3 Final Remarks

This thesis demonstrated that runtime execution data can serve as an accurate and informative foundation for addressing interconnected challenges in identifying microservice boundaries in legacy systems, in the autonomous adaptation of microservice systems in dynamic environments, and in the standardization of software execution event data. The three contributions, *MIST*, Service Colonies, and SEED, collectively advance the state of the art by moving beyond static source code analysis and infrastructure-level metrics toward a principled, execution-data-driven approach to the engineering, operation, and management of microservice-based systems.

MIST demonstrated that historical execution traces, mined using sequential pattern mining, can accurately capture the operational behavior of legacy systems and produce microservices that are not only structurally sound but also empirically validated for runtime performance. Service Colonies demonstrated that treating each microservice as an autonomous agent capable of monitoring its own execution behavior and making proactive adaptation decisions is a feasible and effective approach to reducing the operation burden of microservice systems in dynamic environments. SEED demonstrated that a principled, empirically grounded conceptual data model for software execution event data can achieve broad coverage of the analytical requirements imposed by existing mining techniques, providing a reusable and interoperable foundation for execution-data-driven software analysis.

Together, these contributions establish that runtime execution collected systematically through principled instrumentation, represented in a standardized conceptual form, and analyzed with appropriate mining techniques can substantially improve the efficiency and sustainability of microservice-based software systems across their full engineering lifecycle: from the initial identification of microservices in a legacy codebase, through empirically validated migration, to the ongoing autonomous operation and self-adaptation of deployed systems in production environments.

Bibliography

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architectue in Practice*. Addison-Wesley Professional, 4th edition, 2021.

[2] Newman s. *Building Microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2015.

[3] J. Lewis and M. Fowler. Microservices: A definition of this new architectural term. `https://martinfowler.com/articles/microservices.html`, 2014. [Online; accessed 27-July-2023].

[4] Daniele Wolfart, Wesley K. G. Assunção, Ivonei F. da Silva, Diogo C. P. Domingos, Ederson Schmeing, Guilherme L. Donin Villaca, and Diogo do N. Paza. Modernizing legacy systems with microservices: A roadmap. In *International Conference on Evaluation and Assessment in Software Engineering*, EASE '21, page 149–159, 2021.

[5] Nabor Chagas Mendonça, Pooyan Jamshidi, David Garlan, and Claus Pahl. Developing self-adaptive microservice systems: Challenges and directions. *IEEE Softw.*, 38(2):70–79, 2021.

[6] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng. Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering*, 47:987–1007, 2021.

[7] Jacopo Soldani and Antonio Brogi. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Comput. Surv.*, 2022.

[8] Tijmen de Jong and Jan Martijn E. M. van der Werf. Process-mining based dynamic software architecture reconstruction. In *European Conference on Software Architecture*, pages 217–224, 2019.

[9] Davide Taibi and Kari Systä. From monolithic systems to microservices: A decomposition framework based on process mining. In *International Conference on Cloud Computing and Services Science*, pages 153–164, 2019.

[10] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, pages 20357–20374, 2022.

[11] Roberta Capuano and Henry Muccini. A systematic literature review on migration to microservices: a quality attributes perspective. In *ICSA-C*, pages 120–123, 2022.

[12] Danny Weyns and Usman M. Iftikhar. Activforms: A formally founded model-based approach to engineer self-adaptive systems. *ACM Trans. Softw. Eng. Methodol.*, 32:1–48, 2023.

[13] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. Tools and benchmarks for automated log parsing. In *International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP ’19, page 121–130, 2019.

[14] Utkarsh Desai, Sambaran Bandyopadhyay, and Srikanth G. Tamilselvam. Graph neural network to dilute outliers for refactoring monolith application. In *Conference on Artificial Intelligence(AAAI)*, 2021.

[15] Vikram Nitin, Shubhi Asthana, Baishakhi Ray, and Rahul Krishna. CARGO: AI-guided dependency analysis for migrating monolithic applications to microservices architecture. In *International Conference on Automated Software Engineering*, 2023.

[16] G. Mazlami, J. Cito, and P. Leitner. Extraction of microservices from monolithic software architectures. In *International Conference on Web Services (ICWS)*, pages 524–531, 2017.

[17] Omar Al-Debagy and Peter Martinek. Dependencies-based microservices decomposition method. *International Journal of Computers and Applications*, 44:814–821, 2021.

[18] Wuxia Jin, Ting Liu, Qinghua Zheng, Di Cui, and Yuanfang Cai. Functionality-oriented microservice extraction based on execution trace clustering. In *International Conference on Web Services (ICWS)*, pages 211–218, 2018.

[19] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. Mono2micro: A practical and effective tool for decomposing monolithic java applications to microservices. In *FSE*, page 1214–1224, 2021.

[20] Bo Liu, Jingliu Xiong, Qiurong Ren, Shmuel Tyszberowicz, and Zheng Yang. Log2ms: a framework for automated refactoring monolith into microservices using execution logs. In *International Conference on Web Services (ICWS)*, pages 391–396, 2022.

[21] Tiago Matias, Filipe F. Correia, Jonas Fritzsch, Justus Bogner, Hugo S. Ferreira, and André Restivo. Determining microservice boundaries: A case study using static and dynamic software analysis. In *Software Architecture*, pages 315–332, 2020.

[22] Munezero Immaculee Joselyne, Gaurav Bajpai, and Frederic Nzanywayingoma. A systematic framework of application modernization to microservice based architecture. In *International Conference on Engineering and Emerging Technologies (ICEET)*, pages 1–6, 2021.

[23] Miguel Brito, Jácome Cunha, and João Saraiva. Identification of microservices from monolithic applications through topic modelling. In *Annual ACM Symposium on Applied Computing*, page 1409–1418, 2021.

[24] Mohamed Daoud, Asmae El Mezouari, Noura Faci, Djamal Benslimane, Zakaria Maamar, and Abdelaziz El Fazziki. Towards an automatic identification of microservices from business processes. *International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 42–47, 2020.

[25] Mathawee Tusjunt and Wiwat Vatanawood. Refactoring orchestrated web services into microservices using decomposition pattern. In *International Conference on Computer and Communications (ICCC)*, pages 609–613, 2018.

[26] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. Migrating from monolithic architecture to microservices: A rapid review. In *SCCC*, pages 1–7, 2019.

[27] Roberta Capuano and Henry Muccini. A systematic literature review on migration to microservices: A quality attributes perspective. In *International Conference on Software Architecture Companion (ICSA-C)*, pages 120–123, 2022.

[28] Nabor C. Mendonca, Pooyan Jamshidi, David Garlan, and Claus Pahl. Developing self-adaptive microservice systems: Challenges and directions. *IEEE Softw.*, 38:70–79, 2021.

[29] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, pages 41–50, 2003.

[30] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4, 2009.

[31] David Sinreich. An architectural blueprint for autonomic computing, 2006. IBM White Paper.

[32] Messias Filho, Eliaquim Pimentel, Wellington Pereira, Paulo Henrique M. Maia, and Mariela I. Cortés. Self-adaptive microservice-based systems: Landscape and research opportunities. In *Proceedings of the 2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 167–178. IEEE, 2021.

[33] łLukasz Korzeniowski and Krzysztof Goczyła. Landscape of automated log analysis: A systematic literature review and mapping study. *IEEE Access*, 10:21892–21913, 2022.

[34] Opentelemetry. `https://opentelemetry.io/`, 2005. Accessed: 2026-03-21.

[35] IEEE. IEEE standard for eXtensible Event Stream (XES) for achieving interoperability in event logs and event streams. *IEEE Std 1849-2023 (Revision of IEEE Std 1849-2016)*, pages 1–55, 2023.

[36] Giuseppe A. Di Lucca, Anna Rita Fasolino, Patrizia Guerra, and Silvia Petruzzelli. Migrating legacy systems towards object-oriented platforms. In *International Conference on Software Maintenance*, ICSM '97, page 122–129, 1997.

[37] Eunjoo Lee, Byungjeong Lee, Woochang Shin, and Chisu Wu. A reengineering process for migrating from an object-oriented legacy system to a component-based system. In *International Conference on Computer Software and Applications*, COMPSAC '03, page 336, 2003.

[38] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. Challenges when moving from monolith to microservice architecture. In *Current Trends in Web Engineering*, pages 32–47, 2018.

[39] C. Abrams and R.W Schulte. Service-Oriented Architecture Overview and Guide to SOA Research. `https://doveltech.com/wp-content/uploads/2017/10/serviceoriented_architecture.pdf`, 2008. [Online; accessed 27-July-2023].

[40] Maryam Razavian and Patricia Lago. A systematic literature review on soa migration. *J. Softw. Evol. Process*, 27:337–372, 2015.

[41] M. P. Papazoglou and W.-J. van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007.

[42] Ravi Khadka, Amir Saeidi, Slinger Jansen, Jurriaan Hage, and Geer P. Haas. Migrating a large scale legacy application to soa: Challenges and lessons learned. In *Working Conference on Reverse Engineering (WCRE)*, pages 425–432, 2013.

[43] H.M. Sneed. Integrating legacy software into a service oriented architecture. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 11 pp.–14, 2006.

[44] Tasneem Salah, M. Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. The evolution of distributed systems towards microservices architecture. In *International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 318–325, 2016.

[45] Jonas Fritzsch, Justus Bogner, Stefan Wagner, and Alfred Zimmermann. Microservices migration in industry: Intentions, strategies, and challenges. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 481–490, 2019.

[46] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, 2017.

[47] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33:42–52, 2016.

[48] Holger Knoche and Wilhelm Hasselbring. Using microservices for legacy software modernization. *IEEE Software*, 35:44–49, 2018.

[49] Davide Taibi and Kari Systä. From monolithic systems to microservices: A decomposition framework based on process mining. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science*, pages 153–164, 2019.

[50] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: A systematic approach to service decomposition. In *Service-Oriented and Cloud Computing*, pages 185–200, 2016.

[51] Deepali Bajaj, Anita Goel, and S. C. Gupta. Greenmicro: Identifying microservices from use cases in greenfield development. *IEEE Access*, 10:67008–67018, 2022.

[52] Christoph Schröer, Felix Kruse, and Jorge Marx Gómez. A qualitative literature review on microservices identification approaches. In *Service-Oriented Computing*, pages 151–168, 2020.

[53] Imen Trabelsi, Manel Abdellatif, Abdalgader Abubaker, Naouel Moha, Sébastien Mosser, Samira Ebrahimi-Kahou, and Yann-Gaël Guéhéneuc. From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis. *Journal of Software: Evolution and Process*, 35(10), 2023.

[54] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35:684–702, 2009.

[55] Thomas Ball. The concept of dynamic analysis. In *Software Engineering — ESEC/FSE '99*, pages 216–234, 1999.

[56] R. Pérez-Castillo, B. Weber, I.G.-R. de Guzmán, and M. Piattini. Process mining through dynamic analysis for modernising legacy systems. *IET Software*, pages 304–319, 2011.

[57] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *ICPE 2012*, pages 247–248, 2012.

[58] Majid Rafiei and Wil M. P. van der Aalst. Mining roles from event logs while preserving privacy. In *Business Process Management Workshops*, pages 676–689, 2019.

[59] Wil M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[60] Marlon Dumas, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede. *Process-Aware Information Systems*. John Wiley & Sons, Ltd, 2005.

[61] Ricardo Pérez-Castillo, Barbara Weber, Ignacio García-Rodríguez, and Mario Piattini. Improving event correlation for non-process aware information systems. In *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, pages 33–42, 2012.

[62] Brian Keith and Vianca Vega. Process mining applications in software engineering. In *Trends and Applications in Software Engineering*, pages 47–56, 2017.

[63] Maikel Leemans and Wil M. P. van der Aalst. Process mining in software systems: discovering real-life business transactions and process models from distributed systems. In *International Conference on Model Driven Engineering Languages and Systems*, pages 44–53, 2015.

[64] Philippe Fournier Viger, jerry lin, Uday Rage, Yun Sing Koh, and Rincy Thomas. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, pages 54–77, 2017.

[65] R. Agrawal and R. Srikant. Mining sequential patterns. In *International Conference on Data Engineering (ICDE)*, pages 3–14, 1995.

[66] Jiawei Han, Micheline Kamber, and Jian Pei. 6 - mining frequent patterns, associations, and correlations: Basic concepts and methods. In *Data Mining (Third Edition)*, pages 243–278. Morgan Kaufmann, 2012.

[67] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. page 3–17. Springer-Verlag, 1996.

[68] Mohammed Javeed Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, pages 31–60, 2001.

[69] Jian Pei, Jiawei Han, B. Mortazavi-Asl, Jianyong Wang, H. Pinto, Qiming Chen, U. Dayal, and Mei-Chun Hsu. Mining sequential patterns by pattern-growth: the prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering*, 16:1424–1440, 2004.

[70] Wil M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer, 2016.

[71] Wil M. P. van der Aalst. Process mining: Overview and opportunities. *ACM Transactions on Management Information Systems*, 3(2), July 2012.

[72] Maikel Leemans and Wil M. P. van der Aalst. Process mining in software systems: Discovering real-life business transactions and process models from distributed systems. In *MODELS*, pages 44–53, 2015.

[73] Nicholas R. Jennings. On agent-based software engineering. *Artif. Intell.*, 117(2):277–296, 2000.

[74] Hongbign Wang, Xin Chen, Qin Wu, Qi Yu, Xingguo Hu, Zibin Zheng, and Athman Bouguettaya. Integrating reinforcement learning with multi-agent techniques for adaptive service composition. *ACM Trans. Auton. Adapt. Syst.*, 12:1–42, 2017.

[75] Roger Anderson Schmidt and Marcello Thiry. Microservices identification strategies a review focused on model-driven engineering and domain driven design approaches. In *Iberian Conference on Information Systems and Technologies (CISTI)*, 2020.

[76] Christoph Schröer, Felix Kruse, and Jorge Marx Gómez. A qualitative literature review on microservices identification approaches. In *Service-Oriented Computing*, pages 151–168, 2020.

[77] Michel Cojocaru, Ana Oprescu, and Alexandru Uta. Attributes assessing the quality of microservices automatically decomposed from monolithic applications. *International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 84–93, 2019.

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

[79] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. From monolith to microservices: A classification of refactoring approaches. *CoRR*, abs/1807.10059, 2018.

[80] Daniele Wolfart, Wesley K. G. Assunção, Ivonei F. da Silva, Diogo C. P. Domingos, Ederson Schmeing, Guilherme L. Donin Villaca, and Diogo do N. Paza. Modernizing legacy systems with microservices: A roadmap. In *International Conference on Evaluation and Assessment in Software Engineering (EASE 2011)*, page 149–159, 2021.

[81] Manel Abdellatif, Anas Shatnawi, Hafedh Mili, Naouel Moha, Ghizlane El Boussaidi, Geoffrey Hecht, Jean Privat, and Yann-Gaël Guéhéneuc. A taxonomy of service identification approaches for legacy software systems modernization. *Journal of Systems and Software*, 173:110868, 2021.

[82] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

[83] Barbara Kitchenham and Pearl Brereton. A systematic review of systematic review process research in software engineering. *Information and Software Technology*, 55, 2013.

[84] Robert C. Nickerson, Upkar Varshney, and Jan Muntermann. A method for taxonomy development and its application in information systems. *European Journal of Information Systems*, 22(3):336–359, 2013.

[85] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2012.

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

[87] Yuyang Wei, Yijun Yu, Minxue Pan, and Tian Zhang. A feature table approach to decomposing monolithic applications into microservices. In *Asia-Pacific Symposium on Internetware*, page 21–30, 2021.

[88] Shmuel Tyszberowicz, Robert Heinrich, Bo Liu, and Zhiming Liu. Identifying microservices using functional decomposition. In *Dependable Software Engineering. Theories, Tools, and Applications*, pages 50–65, 2018.

[89] Adambarage Anuruddha Chathuranga De Alwis, Alistair Barros, Colin Fidge, and Artem Polyvyanyy. Discovering microservices in enterprise systems using a business object containment heuristic. In *On the Move to Meaningful Internet Systems. OTM 2018 Conferences*, pages 60–79, 2018.

[90] Adambarage Anuruddha Chathuranga De Alwis, Alistair Barros, Artem Polyvyanyy, and Colin Fidge. Function-splitting heuristics for discovery of microservices in enterprise systems. In *Service-Oriented Computing*, pages 37–53, 2018.

[91] Alexander Krause, Christian Zirkelbach, Wilhelm Hasselbring, Stephan Lenga, and Dan Kröger. Microservice decomposition via static and dynamic analysis of the monolith. In *International Conference on Software Architecture Companion (ICSA-C)*, pages 9–16, 2020.

[92] Pooja Kherwa and Poonam Bansal. Topic modeling: A comprehensive review. *EAI Endorsed Trans. Scalable Inf. Syst.*, 7:e2, 2018.

[93] Ilaria Pigazzini, Francesca Arcelli Fontana, and Andrea Maggioni. Tool support for the migration to microservice architecture: An industrial case study. In *European Conference on Software Architecture*, pages 247–263, 2019.

[94] Luciano Baresi, Martin Garriga, and Alan De Renzis. Microservices identification through interface analysis. In *Service-Oriented and Cloud Computing*, pages 19–33, 2017.

[95] Omar Al-Debagy and Peter Martinek. A new decomposition method for designing microservices. *Period. Polytech. Electr. Eng. Comput. Sci.*, 63:274–281, 2019.

[96] Khaled Sellami, Mohamed Aymen Saied, and Ali Ouni. A hierarchical dbscan method for extracting microservices from monolithic applications. In *International Conference on Evaluation and Assessment in Software Engineering*, page 201–210, 2022.

[97] Mark Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 69:026113, 2004.

[98] Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 76:036106, 2007.

[99] Jakob Löhnertz and Ana Oprescu. Steinmetz: Toward automatic decomposition of monolithic software into microservices, 2020.

[100] Pranava Chaudhari, Amit K. Thakur, Rahul Kumar, Nilanjana Banerjee, and Amit Kumar. Comparison of NSGA-III with NSGA-II for multi objective optimization of adiabatic styrene reactor. *Materials Today: Proceedings*, 57:1509–1514, 2022.

[101] Hisao Ishibuchi, Ryo Imada, Yu Setoguchi, and Yusuke Nojima. Performance comparison of NSGA-II and NSGA-III on various many-objective test problems. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 3045–3052, 2016.

[102] Sinan Eski and Feza Buzluca. An automatic extraction approach: Transition to microservices architecture from monolithic application. In *International Conference on Agile Software Development: Companion*, 2018.

[103] Luís Nunes, Nuno Santos, and António Rito Silva. From a monolith to a microservices architecture: An approach based on transactional contexts. In *Software Architecture*. Springer International Publishing, 2019.

[104] Pascal Zaragoza, Abdelhak-Djamel Seriai, Abderrahmane Seriai, Anas Shatnawi, and Mustapha Derras. Leveraging the layered architecture for microservice recovery. In *International Conference on Software Architecture (ICSA)*, pages 135–145, 2022.

[105] Xiaoxiao Sun, Salamat Boranbaev, Shicong Han, Huanqiang Wang, and Dongjin Yu. Expert system for automatic microservices identification using API similarity graph. *Expert Systems*, 2022.

[106] Omar Al-Debagy. A microservice decomposition method through using distributed representation of source code. *Scalable Comput. Pract. Exp.*, pages 39–52, 2021.

[107] Fredy H. Vera-Rivera, Eduard G. Puerto-Cuadros, Hernán Astudillo, and Carlos Mauricio Gaona-Cuevas. Microservices backlog-A model of granularity specification and microservice identification. In *International Conference on Services Computing-SCC 2020*, page 85–102, 2020.

[108] Teng Wang, Xiang He, Hanchuan Xu, Zhiying Tu, and Zhongjie Wang. Epf4m: An evolution-oriented programming framework for microservices. In *2021 IEEE International Conference on Services Computing (SCC)*, pages 174–182. IEEE, 2021.

[109] Françoise André, Erwan Daubert, and Guillaume Gauvrit. Towards a generic context-aware framework for self-adaptation of service-oriented architectures. In *Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*, ICIW '10, page 309–314. IEEE, 2010.

[110] Sree Ram Boyapati and Claudia Szabo. Self-adaptation in microservice architectures: A case study. In *Proceedings of the 2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 42–51. IEEE, 2022.

[111] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti, and Raffaela Mirandola. Moses: A framework for qos driven runtime adaptation of service-oriented systems. *IEEE Transactions on Software Engineering*, 38(5):1138–1159, 2012.

[112] Guillaume Gauvrit, Erwan Daubert, and Francoise Andre. Safdis: A framework to bring self-adaptability to service-based distributed applications. In *Proceedings of the 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 211–218. IEEE, 2010.

[113] Nikolaus Huber, André Hoorn, Anne Koziolek, Fabian Brosig, and Samuel Kounev. Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments. *Serv. Oriented Comput. Appl.*, 8:73–89, 2014.

[114] Peini Liu, Xinjun Mao, Shuai Zhang, and Fu Hou. Towards reference architecture for a multi-layer controlled self-adaptive microservice system. In *Proceedings of the 2018 30th International Conference on Software Engineering and Knowledge Engineering*, pages 236–241. KSI Research Inc., 2018.

[115] Daniel A. Menascé, Hassan Gomaa, Sam Malek, and João Pedro Sousa. SASSY: A framework for self-architecting service-oriented systems. *IEEE Softw.*, 28(6):78–85, 2011.

[116] Nathalia Nascimento, Paulo Alencar, and Donald Cowan. Self-adaptive large language model (llm)-based multiagent systems. In *International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pages 104–109. IEEE, 2023.

[117] Nuno Oliveira and Luís S. Barbosa. A self-adaptation strategy for service-based architectures. In *2014 Eighth Brazilian Symposium on Software Components, Architectures and Reuse*, pages 1–10. IEEE, 2014.

[118] Diego Perez-Palacin and José Merseguer. Performance sensitive self-adaptive service-oriented software using hidden markov models. *ACM Sigsoft Software Engineering Notes*, 36:40–40, 2011.

[119] Harald Psaier, Lukasz Juszczyk, Florian Skopik, Daniel Schall, and Schahram Dustdar. Runtime behavior monitoring and self-adaptation in service-oriented systems. In *2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 164–173. IEEE, 2010.

[120] Komal Sarda, Zakeya Namrud, Marin Litoiu, Larisa Shwartz, and Ian Watts. Leveraging large language models for the auto-remediation of microservice applications: An experimental study. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, pages 358—-369. ACM, 2024.

[121] Johanneke Siljee, Ivor Bosloper, Jos Nijhuis, and Dieter Hammer. Dysoa: making service systems self-adaptive. In *Proceedings of the Third International Conference on Service-Oriented Computing*, ICSOC'05, pages 255—-268. Springer-Verlag, 2005.

[122] Shuai Zhang, Mingjiang Zhang, Lin Ni, and Peini Liu. A multi-level self-adaptation approach for microservice systems. In *Proceedings of the 2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pages 498–502. IEEE, 2019.

[123] Ouanes Aissaoui, Fadila Atil, and Abdelkrim Amirat. *Towards a Generic Reconfigurable Framework for Self-adaptation of Distributed Component-Based Application*, pages 399–408. Springer, 2013.

[124] Emad Albassam, Jason Porter, Hassan Gomaa, and Daniel A. Menascé. Dare: A distributed adaptation and failure recovery framework for software systems. In *International Conference on Autonomic Computing (ICAC)*, pages 203–208. IEEE, 2017.

[125] Luciano Baresi, Sam Guinea, and Giordano Tamburrelli. Towards decentralized self-adaptive component-based systems. In *Proceedings of the 2008 International Work-*

shop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '08, page 57–64. ACM, 2008.

[126] Hadaytullah, Sriharsha Vathsavayi, Outi Räihä, Kai Koskimies, and Allan Gregersen. Applying genetic self-architecting for distributed systems. In *Proceedings of the 2012 Fourth World Congress on Nature and Biologically Inspired Computing (NaBIC)*, pages 44–52. IEEE, 2012.

[127] Afaf Mousa, Jamal Bentahar, and Omar Alam. Multi-objective self-adaptive composite saas using feature model. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 77–84. IEEE, 2018.

[128] Istio Authors. Architecture. `https://istio.io/latest/docs/ops/deployment/architecture/`, 2026. Accessed: 2026-01-06.

[129] A. Nicolas-Plata, J. L. Gonzalez-Compean, and V. J. Sosa-Sosa. A service mesh approach to integrate processing patterns into microservices applications. *Cluster Computing*, 27:7417–7438, 2024.

[130] Nabor C. Mendonça, David Garlan, Bradley Schmerl, and Javier Cámara. Generality vs. reusability in architecture-based self-adaptation: the case for self-adaptive microservices. In *European Conference on Software Architecture: Companion Proceedings (ECSA '18)*, pages 1–6. ACM, 2018.

[131] Phil Calçado. Pattern: Service mesh. `https://philcalcado.com/2017/08/03/pattern_service_mesh.html`, 2017. Accessed: 2026-01-06.

[132] Buoyant. What is a service mesh?, 2026. Accessed: 2026-01-06.

[133] Martin Fowler. Circuit breaker. `https://martinfowler.com/bliki/CircuitBreaker.html`, 2014. Accessed: 2026-01-06.

[134] Etienne Rivière and Spyros Voulgaris. Gossip-based networking for internet-scale distributed systems. In *E-Technologies: Transformation in a Connected World*, pages 253–284. Springer, 2011.

[135] Anne-Marie Kermarrec and Maarten van Steen. Gossiping in distributed systems. *SIGOPS Oper. Syst. Rev.*, 41:2–7, 2007.

[136] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

[137] Omid Gheibi, Danny Weyns, and Federico Quin. Applying machine learning in self-adaptive systems: A systematic literature review. *ACM Trans. Auton. Adapt. Syst.*, 15(3):9:1–9:37, 2020.

[138] Lukas Malburg, Maximilian Hoffmann, and Ralph Bergmann. Applying MAPE-K control loops for adaptive workflow management in smart factories. *J. Intell. Inf. Syst.*, 61(1):83–111, 2023.

[139] Andreas Metzger, Clément Quinton, Zoltán Ádám Mann, Luciano Baresi, and Klaus Pohl. Realizing self-adaptive systems via online reinforcement learning and feature-model-guided exploration. *Computing*, 106(4):1251–1272, 2024.

[140] Svein O. Hallsteinsen, Kurt Geihs, Nearchos Paspallis, Frank Eliassen, Geir Horn, Jorge Lorenzo, Alessandro Mamelli, and George Angelos Papadopoulos. A development framework and methodology for self-adapting applications in ubiquitous computing environments. *J. Syst. Softw.*, 85(12):2840–2859, 2012.

[141] Martin Pfannemüller, Martin Breitbach, Christian Krupitzer, Markus Weckesser, Christian Becker, Bradley R. Schmerl, and Andy Schürr. REACT: A model-based runtime environment for adapting communication systems. In *Proceedings of the 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 65–74. IEEE, 2020.

[142] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. Formal design and verification of self-adaptive systems with decentralized control. *ACM Trans. Auton. Adapt. Syst.*, 11:1–35, 2017.

[143] Jiyoung Oh, Claudia Raibulet, and Joran Leest. Analysis of mape-k loop in self-adaptive systems for cloud, iot and cps. In *Proceedings of the 2022 International Conference on Service-Oriented Computing—ICSOC 2022 Workshops*, pages 130–141. Springer, 2023.

[144] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. *On Patterns for Decentralized Control in Self-Adaptive Systems*, pages 76–107. Springer, 2013.

[145] Brell Peclard Sanwouo Chekam, Clément Quinton, and Paul Temple. *Breaking the Loop: AWARE is the New MAPE-K*, page 626–630. ACM, 2025.

[146] Terence Wong, Markus Wagner, and Christoph Treude. Self-adaptive systems: A systematic literature review across categories and domains. *Inf. Softw. Technol.*, 148:106934, 2022.

[147] Sheng Wang, Zhijun Ding, and Changjun Jiang. Elastic scheduling for microservice applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):98–115, 2021.

[148] Mahmoud Imdoukh, Imtiaz Ahmad, and Mohammad Alfailakawi. Machine learning based auto-scaling for containerized applications. *Neural Computing and Applications*, 32:9745–9760, 2020.

[149] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[150] Thomas J. Glazier, David Garlan, and Bradley R. Schmerl. Automated management of collections of autonomic systems. In *ACSOS*, pages 82–91. IEEE, 2020.

[151] Andre Cristiano Kalsing, Gleison Samuel do Nascimento, Cirano Iochpe, and Lucineia Heloisa Thom. An incremental process mining approach to extract knowledge from legacy systems. In *International Enterprise Distributed Object Computing Conference*, pages 79–88, 2010.

[152] Ying Zou and Maokeng Hung. An approach for extracting workflows from e-commerce applications. In *International Conference on Program Comprehension (ICPC'06)*, pages 127–136, 2006.

[153] David Garlan. Software architecture: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 91–101, 2000.

[154] Robert C. seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[155] Andres J. Ramirez and Betty H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems(SEAMS '10)*, page 49–58, 2010.

[156] Cong Liu, Boudewijn F. van Dongen, Nour Assy, and Wil M. P. van der Aalst. Detecting behavioral design patterns from software execution data. In *Evaluation of Novel Approaches to Software Engineering*, pages 137–164, 2019.

[157] Koki Kato, Tsuyoshi Kanai, and Sanya Uehara. Source code partitioning using process mining. In *Business Process Management*, pages 38–49, 2011.

[158] Cong Liu, Boudewijn van Dongen, Nour Assy, and Wil M. P van der Aalst. Component interface identification and behavioral model discovery from software execution data. In *Conference on Program Comprehension*, pages 97–107, 2018.

[159] Ting Lu, Cong Liu, Hua Duan, and Qingtian Zeng. Mining component-based software behavioral models using dynamic analysis. *IEEE Access*, 8:68883–68894, 2020.

[160] Vladimir Rubin, Irina Lomazova, and Wil M. P. van der Aalst. Agile development with software process mining. In *Proceedings of the 2014 International Conference on Software and System Process*, pages 70–74, 2014.

[161] Cong Liu. Automatic discovery of behavioral models from software execution data. *IEEE Transactions on Automation Science and Engineering*, 2018.

[162] Vinod Muthusamy, Aleksander Slominski, Vatche Ishakian, Rania Khalaf, Johnathan Reason, and Szabolcs Rozsnyai. Lessons learned using a process mining approach to analyze events from distributed applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, pages 199–204, 2016.

[163] Saulius Astromskis, Andrea Janes, and Michael Mairegger. A process mining approach to measure how users interact with software: An industrial case study. In *Proceedings of the 2015 International Conference on Software and System Process*, pages 137–141, 2015.

[164] Cong Liu, Shi Wang, Shangce Gao, Feng Zhang, and Jiujun Cheng. User behavior discovery from low-level software execution log. *IEEJ Transactions on Electrical and Electronic Engineering*, 13:1624–1632, 2018.

[165] Cong Liu, Boudewijn van Dongen, Nour Assy, and Wil M.P. van der Aalst. Component behavior discovery from software execution data. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2016.

[166] Vinay Raj and G Punnam Chander. Monitoring of microservices architecture based applications using process mining. In *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 486–494, 2022.

[167] Daniele Gadler, Michael Mairegger, Andrea Janes, and Barbara Russo. Mining logs to model the use of a system. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 334–343, 2017.

[168] F. Martinelli, F. Mercaldo, V. Nardone, A. Orlando, A. Santone, and G. Vaglini. Model checking based approach for compliance checking. *Information Technology and Control*, pages 278–298, 2019.

[169] Francesco Blefari, Francesco Aurelio Pironti, and Angelo Furfaro. Toward a log-based anomaly detection system for cyber range platforms. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, 2024.

[170] Marcello Cinque, Raffaele Della Corte, and Antonio Pecchia. Discovering hidden errors from application log traces with process mining. In *2019 15th European Dependable Computing Conference (EDCC)*, pages 137–140, 2019.

[171] Jorge Da Silva, Miren Illarramendi, and Asier Iriarte. Using runtime information of controllers for safe adaptation at runtime: A process mining approach. In *Computer Safety, Reliability, and Security. SAFECOMP 2023 Workshops: ASSURE, DECSoS, SASSUR, SENSEI, SRToITS, and WAISE, Toulouse, France, September 19, 2023, Proceedings*, pages 85–94, 2023.

[172] David Lo and Shahar Maoz. Mining hierarchical scenario-based specifications. In *International Conference on Automated Software Engineering*, pages 359–370, 2009.

[173] Pradeep K. Mahato and Apurva Narayan. Mints: Unsupervised temporal specifications miner. In *International Conference on Software Quality, Reliability and Security (QRS)*, pages 841–851, 2021.

[174] Marc Brünink and David S. Rosenblum. Mining performance specifications. In *FSE*, pages 39–49, 2016.

[175] P. Fournier-Viger, C. W. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, and H. T. Lam. The SPMF open-source data mining library Version 2. In *PKDD*, pages 36–40, 2016.

[176] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. *JavaParser: Visited*. Leanpub, 2023.

[177] Ahmad Banijamali, Pasi Kuvaja, Markku Oivo, and Pooyan Jamshidi. Kuksa*: Self-adaptive microservices in automotive systems. In *PROFES*, volume 12562 of *Lecture Notes in Computer Science*, pages 367–384. Springer, 2020.

[178] Wesley K. G. Assunção, Luciano Marchezan, Lawrence Arkoh, Alexander Egyed, and Rudolf Ramler. Contemporary software modernization: Strategies, driving forces, and research opportunities. *ACM Trans. Softw. Eng. Methodol.*, 2024.

[179] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers Inc., 2nd edition, 2009.

[180] Markus Luckey and Gregor Engels. High-quality specification of self-adaptive software systems. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 143–152, 2013.

[181] João Pablo S. da Silva, Miguel Ecar, Marcelo S. Pimenta, Gilleanes T. A. Guedes, Luiz Paulo Franz, and Luciano Marchezan. A systematic literature review of uml-based domain-specific modeling languages for self-adaptive systems. In *International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '18)*, page 87–93, 2018.

[182] Christian Krupitzer, Felix Maximilian Roth, Martin Pfannemüller, and Christian Becker. Comparison of approaches for self-improvement in self-adaptive systems. In *ICAC*, pages 308–314. IEEE Computer Society, 2016.

[183] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21(6):592–597, June 1972.

[184] Wil M. P. van der Aalst. *Process Mining: A 360 Degree Overview*, pages 3–34. Springer International Publishing, 2022.

[185] Martin Macak, Lukas Daubner, Mohammadreza Fani Sani, and Barbora Buhnova. Process mining usage in cybersecurity and software reliability analysis: A systematic literature review. *Array*, 2022.

[186] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16, 2002.

[187] Zhiqiang Zuo and Siau-Cheng Khoo. Mining dataflow sensitive specifications. In *Formal Methods and Software Engineering*, pages 36–52, 2013.

[188] Cong Liu. Discovery and quality evaluation of software component behavioral models. *IEEE Transactions on Automation Science and Engineering*, 18:1538–1549, 2021.

[189] R. Jeevarathinam and Antony Selvadoss Thanamani. Transaction mapping based approach for mining software specifications. In *NaBIC*, pages 1596–1599, 2009.

[190] A.C. Kalsing, Cirano Iochpe, Lucinéia Thom, and G.S. Nascimento. Evolutionary learning of business process models from legacy systems using incremental process mining. *ICEIS*, pages 58–69, 2013.

[191] André Cristiano Kalsing, Cirano Iochpe, Lucinéia Heloisa Thom, and Gleison Samuel do Nascimento. Re-learning of business process models from legacy system using incremental process mining. In *Enterprise Information Systems*, pages 314–330, 2014.

[192] Cong Liu, Boudewijn van Dongen, Nour Assy, and Wil M. P van der Aalst. A general framework to detect behavioral design patterns. In *International Conference on Software Engineering*, pages 234–235, 2018.

[193] David Lo and Siau-Cheng Khoo. Smartic: towards building an accurate, robust and scalable specification miner. In *International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 265–275, 2006.

[194] Dirk Fahland, David Lo, and Shahar Maoz. Mining branching-time scenarios. In *International Conference on Automated Software Engineering*, pages 443–453, 2013.

[195] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *International Conference on Software Engineering*, pages 51–60, 2008.

[196] David Lo and Shahar Maoz. Specification mining of symbolic scenario-based models. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 29–35, 2008.

[197] David Lo and Shahar Maoz. Scenario-based and value-based specification mining: better together. In *International Conference on Automated Software Engineering*, pages 387–396, 2010.

[198] Yahui Tang, Tong Li, Rui Zhu, Fei Du, Jishu Wang, Zifei Ma, and Francisco Ortin. A discovery method for hierarchical software execution behavior models based on components. *Sci. Program.*, 2021, 2021.

[199] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *International Conference on Automated Software Engineering*, pages 371–382, 2009.

[200] Angelo Ferrando and Giorgio Delzanno. Hypermonitor: A python prototype for hyper predictive runtime verification. In *Reachability Problems*, pages 171–182, 2023.

[201] Evgenii V. Stepanov and Alexey A. Mitsyuk. Extracting high-level activities from low-level program execution logs. *Automated Software Engineering*, 31, 2024.

[202] Xiaoting Zhong, Nan Zhang, and Zhenhua Duan. An approach for automatically generating traces for python programs. In *International Conference on Dependable Systems and Their Applications (DSA)*, pages 262–268, 2022.

[203] Ralph Kimball. A dimensional modeling manifesto. *DBMS*, 10(9):58–70, 1997.

[204] Cong Liu, Boudewijn van Dongen, Nour Assy, and Wil M. P. van der Aalst. A framework to support behavioral design pattern detection from software execution data. In *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 65–76, 2018.

[205] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 227–247, 2008.

[206] Florian Auer, Valentina Lenarduzzi, Michael Felderer, and Davide Taibi. From monolithic systems to microservices: An assessment framework. *Information and Software Technology*, 137, 2021.

[207] Hamdy Michael Ayas, Philipp Leitner, and Regina Hebig. The migration journey towards microservices. In *International Conference on Product-Focused Software Process Improvements(PROFES 2021)*, page 20–35, 2021.

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

[209] Lingli Cao and Cheng Zhang. Implementation of domain-oriented microservices decomposition based on node-attributed network. In *International Conference on Software and Computer Applications*, page 136–142, 2022.

[210] Andrés Carrasco, Brent van Bladel, and Serge Demeyer. Migrating towards microservices: Migration and architecture smells. In *International Workshop on Refactoring*, page 1–6, 2018.

[211] Luiz Carvalho, Alessandro Garcia, Wesley K. G. Assunção, Rafael de Mello, and Maria Julia de Lima. Analysis of the criteria adopted in industry to extract microservices. In *International Workshop on Conducting Empirical Studies in Industry (CESI) and International Workshop on Software Engineering Research and Industrial Practice*, pages 22–29, 2019.

[212] Adambarage Anuruddha Chathuranga De Alwis, Alistair Barros, Colin Fidge, and Artem Polyvyanyy. Availability and scalability optimized microservice discovery from enterprise systems. In *On the Move to Meaningful Internet Systems: OTM 2019 Conferences*, pages 496–514, 2019.

[213] Daniel Escobar, Diana Cárdenas, Rolando Amarillo, Eddie Castro, Kelly Garcés, Carlos Parra, and Rubby Casallas. Towards the understanding and evolution of monolithic applications as microservices. In *Latin American Computing Conference (CLEI)*, pages 1–11, 2016.

[214] Francisco Freitas, André Ferreira, and Jácome Cunha. Refactoring java monoliths into executable microservice-based applications. In *Brazilian Symposium on Programming Languages*, page 100–107, 2021.

[215] Andrei Furda, Colin Fidge, Olaf Zimmermann, Wayne Kelly, and Alistair Barros. Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency. *IEEE Software*, 35:63–72, 2018.

[216] Neil Lapuz, Paul Clarke, and Yalemisew Abgaz. Digital transformation and the role of dynamic tooling in extracting microservices from existing software systems. In *Systems, Software and Services Process Improvement*, pages 301–315, 2021.

[217] Zhiding Li, Chenqi Shang, Jianjie Wu, and Yuan Li. Microservice extraction based on knowledge graph from monolithic applications. *Information and Software Technology*, 150, 2022.

[218] Zhongshan Ren, Wei Wang, Guoquan Wu, Chushu Gao, Wei Chen, Jun Wei, and Tao Huang. Migrating web applications from monolithic structure to microservices architecture. In *Asia-Pacific Symposium on Internetware*, 2018.

[219] Nuno Santos and António Rito Silva. A complexity metric for microservices architecture migration. In *International Conference on Software Architecture (ICSA)*, pages 169–178, 2020.

[220] Samuel Santos and António Rito Silva. Microservices identification in monolith systems: Functionality redesign complexity and evaluation of similarity measures. *Journal of Web Engineering*, 21:1543–1582, 2022.

[221] Tatjana D. Stojanovic, Sasa D. Lazarevic, Milos Milic, and Ilija Antovic. Identifying microservices using structured system analysis. In *International Conference on Information Technology (IT)*, pages 1–4, 2020.

[222] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, pages 22–32, 2017.

[223] Davide Taibi and Kari Systä. A decomposition and metric-based evaluation framework for microservices. In *Cloud Computing and Services Science*, pages 133–149, 2020.

[224] Rahul Yedida, Rahul Krishna, Anup Kalia, Tim Menzies, Jin Xiao, and Maja Vukovic. Lessons learned from hyper-parameter tuning for microservice candidate identification. In *International Conference on Automated Software Engineering (ASE)*, pages 1141–1145, 2021.

[225] Wesley K. G. Assunção, Thelma Elita Colanzi, Luiz Carvalho, Alessandro Garcia, Juliana Alves Pereira, Maria Julia de Lima, and Carlos Lucena. Analysis of a many-objective optimization approach for identifying microservices from legacy systems. *Empirical Software Engineering*, 2022.

[226] Wesley K. G. Assunção, Thelma Elita Colanzi, Luiz Carvalho, Juliana Alves Pereira, Alessandro Garcia, Maria Julia de Lima, and Carlos Lucena. A multi-criteria strategy for redesigning legacy features as microservices: An industrial case study. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 377–387, 2021.

[227] Matteo Camilli, Carmine Colarusso, Barbara Russo, and Eugenio Zimeo. Domain metric driven decomposition of data-intensive applications. In *International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 189–196, 2020.

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

[229] MohammadHadi Dehghani, Shekoufeh Kolahdouz-Rahimi, Massimo Tisi, and Dalila Tamzalit. Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning. *Software and System Modeling*, page 1115–1133, 2022.

[230] Justas Kazanavičius, Dalius Mazeika, and Diana Kalibatiene. An approach to migrate a monolith database into multi-model polyglot persistence based on microservice architecture: A case study for mainframe database. *Applied Sciences*, 12:6189, 2022.

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

[232] Salvatore Augusto Maisto, Beniamino Di Martino, and Stefania Nacchia. From monolith to cloud architecture using semi-automated microservices modernization. In *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, 2020.

[233] Fredy H. Vera-Rivera, Eduard Puerto, Hernán Astudillo, and Carlos Mauricio Gaona. Microservices backlog—A genetic programming technique for identification and

evaluation of microservices from user stories. *IEEE Access*, pages 117178–117203, 2021.

Appendix A

Literature Review Study List

This appendix presents the complete set of 117 studies used for the literature review study presented in Section 3.1, reengineering software systems into microservices.

ID	Full reference
1	M. Gysel et al., Service cutter: A systematic approach to service decomposition, in: Service-Oriented and Cloud Computing, 2016 [50].
2	D. Taibi, K. Syst¨a, From monolithic systems to microservices: A decomposition framework based on process mining, International Conference on Cloud Computing and Services Science, 2019 [9].
3	F. Auer et al., From monolithic systems to microservices: An assessment framework, Information and Software Technology, 2021 [206].
4	H. Michael Ayas et al., The migration journey towards microservices, in: Product-Focused Software Process Improvement: 22nd International Conference, 2021 [207].
5	B. Deepali et al., Partial Migration for Re-architecting a Cloud Native Monolithic Application into Microservices and FaaS.
6	A. Muhammad et al., Unsupervised Learning Approach for Web Application Auto-Decomposition into Microservices. Journal of Systems and Software, 2019
7	D. Bajaj et al., Greenmicro: Identifying microservices from use cases in greenfield development, IEEE, 2022 [51].
8	M. H. Gomes Barbosa, P. H. M. Maia, Towards identifying microservice candidates from business rules implemented in stored procedures, IEEE International Conference on Software Architecture Companion, 2020 [208].
9	R. Belafia et al., From Monolithic to Microservice Architecture: The Case of Extensible and Domain-Specific IDEs. ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), 2021.
10	K. Bozan, K. Lyytinen, G. M. Rose, How to transition incrementally to microservice architecture, Commun. ACM, 2020.
11	M. Brito et al., Identification of microservices from monolithic applications through topic modelling, 36th Annual ACM Symposium on Applied Computing, 2021 [23].
12	V. Bushong et al., "Using Static Analysis to Address Microservice Architecture Reconstruction," International Conference on Automated Software Engineering, 2021.
13	L. Cao, C. Zhang, Implementation of domain-oriented microservices decomposition based on node-attributed network, 11th International Conference on Software and Computer Applications, 2022 [209].
14	A. Carrasco et al., Migrating towards microservices: Migration and architecture smells, 2nd International Workshop on Refactoring, 2018 [210].

15	L. Carvalho et al., Analysis of the criteria adopted in industry to extract microservices, 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice, 2019 [211].
16	C. Tomas Cerny et al., On isolation-driven automated module decomposition.Conference on Research in Adaptive and Convergent Systems, 2018.
17	C. Nacha et al., Software Architectural Migration: An Automated Planning Approach. ACM Trans. Softw. Eng. Methodol, 2021.
18	Christoforou, A. et al., Supporting the Decision of Migrating to Microservices Through Multi-layer Fuzzy Cognitive Maps. Service-Oriented Computing. ICSOC, 2017.
19	da Silva, C.E. et al., SPReaD: service-oriented process for reengineering and DevOps. SOCA, 2022.
20	Hugo H. O. S. da Silva et al.,Towards a Roadmap for the Migration of Legacy Software Systems to a Microservice based Architecture. 9th International Conference on Cloud Computing and Services Science, 2019 [24].
21	M. Daoud et al.,Towards an Automatic Identification of Microservices from Business Processes, 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2020.
22	Dattatreya, V. et al., Design Patterns and Microservices for Reengineering of Legacy Web Applications, 2021.
23	de Almeida, M.G., Canedo, The Adoption of Microservices Architecture as a Natural Consequence of Legacy System Migration at Police Intelligence Department.ICCSA 2022.
24	A. A. C. De Alwis et al., Discovering microservices in enterprise systems using a business object containment heuristic, On the Move to Meaningful Internet Systems. OTM 2018 [89].
25	A. A. C. De Alwis et al., Function-splitting heuristics for discovery of microservices in enterprise systems, Service-Oriented Computing, 2018 [90].
26	A. A. C. De Alwis et al., Availability and scalability optimized microservice discovery from enterprise systems, On the Move to Meaningful Internet Systems: OTM 2019 [212].
27	De Alwis, A.A.C. et al., Remodularization Analysis for Microservice Discovery Using Syntactic and Semantic Clustering. Advanced Information Systems Engineering. CAiSE 2020.
28	De Alwis, A.A.C. et al., Microservice Remodularisation of Monolithic Enterprise Systems for Embedding in Industrial IoT Networks. CAiSE 2021.
29	U. Desai et al., Graph neural network to dilute outliers for refactoring monolith application, Conference on Artificial Intelligence, 2021 [14].
30	E. Djogic, S. Ribic and D. Donko, ”Monolithic to microservices redesign of event driven integration platform,” 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2018.
31	D. Escobar et al., Towards the understanding and evolution of monolithic applications as microservices, XLII Latin American Computing Conference, 2016 [213].
32	S. Eski, F. Buzluca, An automatic extraction approach: Transition to microservices architecture from monolithic application, 19th International Conference on Agile Software Development, 2018 [102].
33	F. Freitas et al., A. Ferreira, J. Cunha, Refactoring Java Monoliths into Executable Microservice-Based Applications, 2021 [214].
34	A. Furda et al., Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency, IEEE Software 35, 2018 [215].
35	Gutiérrez-Fernández et al., Redefining a Process Engine as a Microservice Platform. Business Process Management Workshops, 2016.
36	A. O. R. Ishida et al., K., Extracting Micro Service Dependencies Using Log Analysis,IEEE 29th Annual Software Technology Conference (STC),2022.
37	Md Rofiqul Islam and Tomas Cerny. Business process extraction using static analysis. 36th IEEE/ACM International Conference on Automated Software Engineering, 2021.
38	A. Janes and B. Russo, Automatic Performance Monitoring and Regression Testing During the Transition from Monolith to Microservices, ISSREW, 2019.

39	W. Jin et al., Service candidate identification from monolithic systems based on execution traces, IEEE Transactions on Software Engineering 47, 2021 [6].
40	M. I. Joselyne et al., A systematic framework of application modernization to microservice based architecture, International Conference on Engineering and Emerging Technologies (ICEET), 2021 [22].
41	I. J. Munezero et al., Partitioning Microservices: A Domain Engineering Approach, IEEE/ACM Symposium on Software Engineering in Africa, 2018.
42	A. K. Kalia et al., Mono2Micro: A practical and effective tool for decomposing monolithic Java applications to microservices, 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021 [19].
43	Anup K. Kalia et al., Mono2Micro: an AI-based toolchain for evolving monolithic enterprise applications to a microservice architecture. 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020.
44	M. Kamimura et al., Extracting Candidates of Microservices from Monolithic Application Code, 25th Asia-Pacific Software Engineering Conference, 2018.
45	Khoshnevis, S. A search-based identification of variable microservices for enterprise SaaS. Front. Comput. Sci. 17, 2023
46	A. Krause et al., Microservice decomposition via static and dynamic analysis of the monolith, IEEE International Conference on Software Architecture Companion, 2020 [91].
47	N. Lapuz et al., Digital transformation and the role of dynamic tooling in extracting microservices from existing software systems, Systems, Software and Services Process Improvement,2021 [216].
48	S. Li, H. Zhang et al., A dataflow-driven approach to identifying microservices from monolithic applications, Journal of Systems and Software 157, 2019 [86].
49	Z. Li, C. Shang, J. Wu, Y. Li, Microservice extraction based on knowledge graph from monolithic applications, Information and Software Technology 150,2022 [217].
50	B. Liu et al., Method of Microservices Division for Complex Business Management System Based on Dual Clustering, International Conference on Mechanical, Control and Computer Engineering, 2020
51	J. Löhnertz, A. Oprescu, Steinmetz: Toward automatic decomposition of monolithic software into microservices, 2020 [99].
52	T. Matias et al., Determining microservice boundaries: A case study using static and dynamic software analysis, Software Architecture,2020 [21].
53	V. Nitin et al, CARGO: AI-guided dependency analysis for migrating monolithic applications to microservices architecture, 37th IEEE/ACM International Conference on Automated Software Engineering, 2023 [15].
54	park, J. et al., Approach to Identify Microservices based on Analysis Class Model. International Journal of Advanced Science and Technology, 28, 2019.
55	I. Pigazzini et al., Tool support for the migration to microservice architecture: An industrial case study, European Conference on Software Architecture, 2019 [93].
56	T. Prasandy et al.,Migrating Application from Monolith to Microservices, International Conference on Information Management and Technology 2020.
57	Z. Ren et al., Migrating web applications from monolithic structure to microservices architecture, 10th Asia-Pacific Symposium on Internetware, 2018 [218].
58	Y. Romani et al.,Towards Migrating Legacy Software Systems to Microservice-based Architectures: a Data-Centric Process for Microservice Identification, 19th International Conference on Software Architecture Companion (ICSA-C), 2022.
59	Saidi, M. et al., Automatic Microservices Identification Across Structural Dependency. Hybrid Intelligent Systems. 2022.
60	N. Santos, A. Rito Silva, A complexity metric for microservices architecture migration, 2020 IEEE International Conference on Software Architecture (ICSA), 2020 [219].
61	S. Santos, A. R. Silva, Microservices identification in monolith systems: Functionality redesign complexity and evaluation of similarity measures, Journal of Web Engineering 21, 2022 [220].
62	Sarita and S. Sebastian, Transform Monolith into Microservices using Docker, International Conference on Computing, Communication, Control and Automation, 2017.

63	Casper Schröder et al., Search-based software re-modularization: a case study at Adyen. 43rd International Conference on Software Engineering: Software Engineering in Practice, 2021.
64	Christoph Schröer, Towards Microservice Identification Approaches for Architecting Data Science Workflows, Procedia Computer Science, 2021.
65	C. Schroer, S. Wittfoth and J. M. Gomez, A Process Model for Microservices Design and Identification, IEEE 18th International Conference on Software Architecture Companion, 2021.
66	K. Sellami et al., A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications, 26th International Conference on Evaluation and Assessment in Software Engineering, 2022 [96].
67	Selmadji, A. et al., Re-architecting OO Software into Microservices. Service-Oriented and Cloud Computing, 2018.
68	A. L. Shastry et al., Approaches for migrating non cloud-native applications to the cloud, IEEE 12th Annual Computing and Communication Workshop and Conference, 2022.
69	T. D. Stojanovic et al., Identifying microservices using structured system analysis, 24th International Conference on Information Technology (IT), 2020 [221].
70	X. Sun et al., Expert system for automatic microservices identification using API similarity graph, Expert Systems, 2022 [105].
71	D. Taibi et al., Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation, IEEE Cloud Computing 4 ,2017 [222].
72	D. Taibi, K. Syst¨a, A decomposition and metric-based evaluation framework for microservices, Cloud Computing and Services Science, 2020 [223].
73	M. Tusjunt, W. Vatanawood, Refactoring orchestrated web services into microservices using decomposition pattern, IEEE 4th International Conference on Computer and Communications (ICCC), 2018 [25].
74	S. Tyszberowicz et al., Identifying microservices using functional decomposition, Dependable Software Engineering. Theories, Tools, and Applications, 2018 [88] .
75	F. H. Vera-Rivera et al., Microservices backlog—A model of granularity specification and microservice identification, 17th International Conference, 2020 [107].
76	E. Volynsky et al., Architect: A Framework for the Migration to Microservices, International Conference on Computing, Electronics Communications Engineering (iCCECE), 2022.
77	Y. Wei et al., A feature table approach to decomposing monolithic applications into microservices, 12th Asia-Pacific Symposium on Internetware, 2021 [87].
78	R. Yedida et al., Lessons learned from hyper-parameter tuning for microservice candidate identification, 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021 [224].
79	P. Zaragoza et al., Leveraging the layered architecture for microservice recovery, IEEE 19th International Conference on Software Architecture (ICSA), 2022 [104].
80	O. Al-Debagy, P. Martinek, Dependencies-based microservices decomposition method, International Journal of Computers and Applications 44, 2021 [17].
81	Almeida, J.F., Silva, A.R., Monolith Migration Complexity Tuning Through the Application of Microservices Patterns. ECSA 2020.
82	W. K. G. Assun¸c˜ao et al., Analysis of a many-objective optimization approach for identifying microservices from legacy systems, Empirical Softw. Engg. (2022)[225].
83	W. K. G. Assun¸c˜ao et al., A multi-criteria strategy for redesigning legacy features as microservices: An industrial case study, IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2021 [226].
84	Wesley K. G. Assunção et al., Variability management meets microservices: six challenges of re-engineering microservice-based webshops. 24th ACM Conference on Systems and Software Product Line, 2020.
85	L. Baresi et al., Microservices identification through interface analysis, Service-Oriented and Cloud Computing, 2017 [94].
86	G. Mazlami et al., Extraction of microservices from monolithic software architectures, IEEE International Conference on Web Services (ICWS), 2017 [16].
87	O. Al-Debagy, P. Martinek, A new decomposition method for designing microservices, Period. Polytech. Electr. Eng. Comput. Sci. 63, 2019 [95] .

88	W. Jin et al., Functionality-oriented microservice extraction based on execution trace clustering, IEEE International Conference on Web Services (ICWS), 2018 [18].
89	I. Trabelsi et al., From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis, Journal of Software: Evolution and Process, 2022 [53].
90	O. Al-Debagy, A microservice decomposition method through using distributed representation of source code, Scalable Comput. Pract.Exp., 2021 [106].
91	Levcovitz, A. et al., Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. 2016.
92	M. J. Amiri, "Object-Aware Identification of Microservices," IEEE International Conference on Services Computing(SCC), 2018.
93	S. Agarwal et al., Monolith to Microservice Candidates using Business Functionality Inference, 2021 IEEE International Conference on Web Services (ICWS), 2021.
94	C. Bandara and I. Perera, Transforming Monolithic Systems to Microservices - An Analysis Toolkit for Legacy Code Evaluation, 20th International Conference on Advances in ICT for Emerging Regions (ICTer), 2020.
95	M. Camilli et al., Domain metric driven decomposition of data-intensive applications, IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2020 [227].
96	L. Carvalho et al., On the performance and adoption of search-based microservice identification with toMicroservices, IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020 [228].
97	C. Andreas et al., Migration of Software Components to Microservices: Matching and Synthesis, 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), 2019.
98	M. Cojocaru et al., MicroValid: A Validation Framework for Automatically Decomposed Microservices, IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2019.
99	M. Dehghani et al., Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning, Softw. Syst. Model.,2022 [229].
100	F. -D. Eyitemi and S. Reiff-Marganiec, System Decomposition to Optimize Functionality Distribution in Microservices with Rule Based Approach, IEEE International Conference on Service Oriented Systems Engineering (SOSE), 2020.
101	G. Filippone et al., Migration of Monoliths through the Synthesis of Microservices using Combinatorial Optimization, IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2021.
102	N. Ivanov and A. Tasheva, A Hot Decomposition Procedure: Operational Monolith System to Microservices, International Conference Automatics and Informatics (ICAI), 2021.
103	J. Kazanavičius et al., An Approach to Migrate a Monolith Database into Multi-Model Polyglot Persistence Based on Microservice Architecture: A Case Study for Mainframe Database [230]
104	D. Kuryazov et al., Towards Decomposing Monolithic Applications into Microservices, IEEE 14th International Conference on Application of Information and Communication Technologies (AICT), 2020.
105	L. De Lauretis, From Monolithic Architecture to Microservices Architecture, IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2019.
106	C.-Y. Li et al., Microservice migration using strangler fig pattern: A case study on the green button system, International Computer Symposium (ICS), 2020 [231].
107	J. Lin et al., Migrating web applications to clouds with microservice architectures, International Conference on Applied System Innovation (ICASI), 2016.
108	Log2MS: a framework for automated refactoring monolith into microservices using execution logs [20] B. Liu et al., Log2ms: a framework for automated refactoring monolith into microservices using execution logs, IEEE International Conference on Web Services(ICWS), 2022 [20].
109	S. A. Maisto et al., From monolith to cloud architecture using semi-automated microservices modernization, Advances on P2P, Parallel, Grid, Cloud and Internet Computing, 2020 [232].
110	R. Mishra et al., Transition from Monolithic to Microservices Architecture: Need and proposed pipeline, International Conference on Futuristic Technologies (INCOFT), 2022.
111	Nunes, L. et al.,From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts, ECSA, 2019.

112	Ana Santos and Hugo Paula, Microservice decomposition and evaluation using dependency graph and silhouette coefficient, 15th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS), 2021.
113	A. Selmadji et al., From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach, IEEE International Conference on Software Architecture (ICSA), 2020.
114	A. Shimoda and T. Sunada, Priority Order Determination Method for Extracting Services Stepwise from Monolithic System, 7th International Congress on Advanced Applied Informatics (IIAI-AAI), 2018.
115	F. H. Vera-Rivera et al., Microservices backlog—A genetic programming technique for identification and evaluation of microservices from user stories, IEEE Access. 2021 [233].
116	Z. Yang et al., A Microservices Identification Approach based on Problem Frames, IEEE 2nd International Conference on Software Engineering and Artificial Intelligence (SEAI), 2022.
117	J. Zhao and K. Zhao, Applying Microservice Refactoring to Object-2riented Legacy System, 8th International Conference on Dependable Systems and Their Applications (DSA), 2021.