Static Microservice Architecture Recovery Using Model-Driven Engineering

Nuha Alshuqayran

A thesis submitted in partial fulfilment of the requirements of the University of Brighton for the degree of Doctor of Philosophy

May 2020

Acknowledgements

Undertaking this PhD has given me numerous meaningful insights and knowledgeable experiences, and completion of this work would not have been possible without the participation and support I obtained from several people. I wish to thank my supervisors, Dr Nour Ali and Dr Roger Evans, for imparting their profound knowledge and consistently guiding me. I would not have been able to complete my PhD without adequate guidance, constant encouragement and meaningful feedback from my supervisors. Their supervision, from the introductory level of thesis to the concluding level, enabled me to develop proper understanding of the subject and helped me in completing each phase with maximum efficacy. I would also like to thank my husband for being a strong support system throughout my PhD, as he quit his job during this time to travel with me so that I could complete my thesis. His support has been commendable. Furthermore, I would like to extend deepest regards to my family members, including my parents, brothers and sisters for supporting me by rendering their constant, love, affection and encouragement during the writing of my thesis and at every step of my life. I feel extremely fortunate to have had the guidance, strength and support of my father and mother in accomplishing this stepping stone in my professional life.

Declaration

I declare that the research contained in this thesis, unless otherwise formally indicated within the text, is the original work of the author. The thesis has not been previously submitted to this or any other university for a degree, and does not incorporate any material already submitted for a degree.

NUHA ALSHUQAYRAN

May 2020

Abstract

The architecture of software systems plays a significant role in the different stages of the software lifecycle, including, for example, evolution, maintenance and re-use. Software architecture represents the high-level design of a software system, consisting of software elements and the relationships that allow the architecture to properly function. During the last decade, changes in the software development industry have led in the direction of developing software in a new architectural style called microservice architecture. This thesis presents research in support of this. Software developed using microservice architecture is complex and distributed, and involves several technologies and components. Reverse engineering, and specifically architecture recovery, can aid in the understanding and maintenance of microservice systems. This thesis presents a Microservice Architecture Recovery (MiSAR) approach, based on the paradigm of Model-Driven Engineering (MDE), that recovers the architecture of microservice systems statically. MiSAR aims to comprehend the complexities of microservice architecture by developing a bottom-up reverse engineering process. The process of reverse engineering starts from the code to a Platform-Specific Model (PSM) that supports the technology of the implemented microservice system, leading to a Platform-Independent Model (PIM) at the architectural level. MiSAR follows an MDE approach and includes two key components: a metamodel, which abstracts the concepts of a particular microservice architecture in a technology-independent manner, and mapping rules, which map an implemented microservice-based system into an architectural model which instantiates the metamodel. To design and develop MiSAR, two empirical studies were conducted which analyse existing software systems that employ the microservice architectural style. Based on the results of these studies, MiSAR can produce effective and expressive architectural models of implemented microservice systems, which are crucial for developers.

Table of Contents

1. Introduction 16

1.1. Introduction 16

1.2. Research Aims and Objectives 20

1.3. Research Motivation and Research Questions 21

1.4. Research Contributions 22

1.5. Thesis Structure 24

1.6. Publications 25

A Systematic Mapping Study in Microservice Architecture 26

2.1. Introduction 26

2.2. The Need for a Systematic Mapping Study 26

2.3. Research Method 27

2.4. Results 33

2.5. Discussion 40

2.6. Summary 43

Background and Related Work 44

3.1. Introduction 44

3.2. Background 44

3.2.1. Microservice Architecture 44

3.2.2. Model-Driven Engineering 47

3.2.3. Characteristics of Model Transformation Approaches 51

3.2.4. Major Categories of Model Transformation Approach 53

3.2.5. Model Transformation Languages, Tools and Standards 56

3.2.6. Re-engineering and Reverse Engineering 57

3.2.7. Software Architecture Recovery 59

3.2.8 Architecture Recovery Frameworks 66

3.3. Related work 70

3.3.1 Overview of Model-Driven Architecture Recovery Approaches 70
3.3.2 Microservice Architecture Recovery Approaches 81
3.3.3 Model-Driven Approaches for Microservices 87
3.3.4 Comparison of the Related Approaches 90
3.4. Research Gap 92
3.5. Summary 94
Research Methodology 97
4.1. Introduction 97
4.2. Research Methodology 97
4.3. System Studies 102
4.4. Summary 103
Microservice Architecture Recovery (MiSAR) 104
5.1. Introduction 104
5.2. Overview of MiSAR 104
5.3. Microservice Application Platform-Specific Model 106
5.4. Empirical study to define MiSAR 107
5.4.1. Selection of Systems to Study 109
5.4.2. Research Design 111
5.4.3. Results 122
5.5. Summary 141
Improving the Initial Artefacts of MiSAR: An Empirical Study 142
6.1. Introduction 142
6.2. MiSAR Abstraction Levels 142
6.3. Identification of PSM metamodel 143
6.4. Study Design 149
6.4.1. Study Aim and Research Questions 149
6.4.2. Selecting the Case Studies 150

6.4.3. Research Design 151

6.5. Results 154

6.6. Summary 200

MiSAR Metamodels and Mapping Rules: Features and Implementation 202

7.1. Introduction 202

7.2. MiSAR Implementation Environment 202

7.3. Ecore Metamodel Implementation 203

7.4. QVT, the Transformation Rules Implementation 205

7.4.1. Model Type Definitions 208

7.4.2. Transformation Declaration 208

7.4.3. Main Function 208

7.4.4. Mapping 209

7.5. Mapping Rule Features 214

7.6. Summary 223

An Evaluation of MiSAR Artefacts through Microservice Architecture Recovery: A Case Study 224

8.1. Introduction 224

8.2. Evaluation Methodology 224

8.3. Case Study 226

8.3.1. Design 226

8.3.2. Case Selection 227

8.3.3. Application of MiSAR 229

8.3.4. Consistency Checks 246

8.3.5. Results 266

8.3.6. Updates to MiSAR's Repository 270

8.4. Summary 271

Discussion 272

9.1. Introduction 272

9.2. Discussion of Study 1 and Study 2: Empirical Studies 272

9.3. Discussion of Study 3: Evaluation 282

9.4. Previous Studies 285

9.5. Considerations and Positive Aspects of MiSAR 292

9.6. Observations and Lessons Learned 295

9.7. Summary 297

Conclusions and Further Work 298

10.1. Introduction 298

10.2. Major Topics Addressed 298

10.3. Research Stages 300

10.4. Summary of Contributions 301

10.5. Threats to Validity 302

10.6. Future Directions 306

References 308

Appendices 319

List of Figures

Figure 1-1: The growing number of searches for the keywords ‘microservice architecture’, according to a Google Trends report. 18
Figure 1-2: Thesis structure diagram. 25
Figure 2-1: The top 10 keywords in the literature....34
Figure 2-2: The distribution of microservice challenges in the literature. 35
Figure 2-3: Research approaches against the number of papers with different challenges. 36
Figure 3-1: The MDA framework (Miller et al., 2003). 48
Figure 3-2: The four-layer meta-modelling architecture (Di Ruscio et al., 2012). 50
Figure 3-3: Basic concepts of model transformation (Di Ruscio et al., 2012). 51
Figure 3-4: Reverse and forward engineering (Chikofsky and Cross, 1990). 58
Figure 3-5: A process for SAR (Ducasse and Pollet, 2009, p. 576). 61
Figure 3-6: A bottom-up process (Ducasse and Pollet, 2009). 62
Figure 3-7: A top-down process (Ducasse and Pollet, 2009)....62
Figure 3-8: A hybrid process (Ducasse and Pollet, 2009). 63
Figure 3-9: Extract-abstract-present paradigm....66
Figure 3-10: The filtering and clustering framework (Mendonça and Kramer, 1996). 67
Figure 3-11: The compliance checking framework (Mendonça and Kramer, 1996). 67
Figure 3-12: The main conception considered in the approaches. 70
Figure 3-13: General principles of a) model discovery and b) model understanding (Brunelière et al. 2014). 73
Figure 3-14: Overall approach (Cosentino et al., 2012)....74
Figure 3-15: MARBLE framework (Pérez-Castillo, De Guzmán, et al., 2011). 76
Figure 3-16: a) The PSM metamodel. b) The DCD metamodel (El Beggar et al., 2013). 78
Figure 3-17: Fleurey et al.’s (2007) reverse engineering process. 79
Figure 3-18: MicroART metamodel for microservice-based systems (Granchelli, Cardarelli, Francesco, et al., 2017). 82
Figure 3-19: Data model (Mayer and Weinreich, 2018). 86
Figure 3-20: The architecture extraction process. 87
Figure 3-21: Metamodel proposed by Düllmann and van Hoorn (2017). 88

Figure 3-22: Metamodels a, b and c proposed by Rademacher, Sorgalla, et al., (2019). 89
Figure 3-23: Technology Modeling Language defined by Rademacher et al. (2019) 90
Figure 4-1: Research framework (Hevner et al., 2004, p. 80). 98
Figure 4-2: MiSAR methodological approach. 101
Figure 5-1: Approach overview. 105
Figure 5-2: Representation of microservice concept at PSM/PIM layer. 107
Figure 5-3: Study steps. 108
Figure 5-4: Packages and classes extracted from source code. 114
Figure 5-5: Using Zipkin to trace transactions. 115
Figure 5-6: Dynamic analysis using Zipkin. 115
Figure 5-7: The bridge concept. 117
Figure 5-8: Concrete example for API gateway concept. 117
Figure 5-9: Microservice conceptual map. 118
Figure 5-10: Microservice architecture concerns. 119
Figure 5-11: Sketch of the related architectural concepts under one cluster. 120
Figure 5-12: Architectural concepts, counts and system references. 124
Figure 5-13:Microservice architecture metamodel at the PIM level. 127
Figure 5-14: The partial instance diagram of case study 1 after the recovery of the registry microservice. 141
Figure 6-1: MiSAR abstraction levels. 143
Figure 6-2: The PSM metamodel identification steps. 144
Figure 6-3: PSM model tree. 145
Figure 6-4: PSM metamodel (Java PSM metamodel is reduced). 147
Figure 6-5: Java PSM metamodel. 148
Figure 6-6: Empirical process for enhancing and refining MiSAR. 151
Figure 6-7: PIM instance recovered for edge-service from case study 7 using PIM metamodel (Version 1). 156
Figure 6-8: Enhanced PIM instance recovered for edge-service, discovery-service and config-service microservice from case study 7. 161
Figure 6-9: (a) PIM instances recovered for Kafka microservice from case study 4 based on PIM metamodel (Version 1). (b) Enhanced PIM instances recovered for *kafka* microservice based on enhanced PIM metamodel (Version 2). 163

Figure 6-10: PIM metamodel (version 2)....164
Figure 6-11: Synchronous request-response between card-statement- composite, card and statement microservices from case study 1. 166
Figure 6-12: PIM instance recovered for card-statement-composite microservice from case study 1 based on PIM metamodel (version 2). 167
Figure 6-13: PIM metamodel (version 3)....169
Figure 6-14: Enhanced PIM instance recovered for card-statement-composite microservice from case study 1 based on enhanced PIM metamodel (version 3). ..170
Figure 6-15:Asynchronous message-driven inter-service communication between weatherservice and weatherbackend microservices (case study 2)....172
Figure 6-16: Internal setup of rabbitmq message broker from case study 2. 173
Figure 6-17: PIM metamodel (version 4)....176
Figure 6-18: Enhanced PIM instance recovered for weatherbackend microservice from case study 2 based on PIM metamodel (version 4). 177
Figure 6-19: PIM instance recovered for *recommendation-service* microservice from case study 4 based on PIM metamodel (version 4). 179
Figure 6-20: Final PIM metamodel (version 5). 180
Figure 6-21: Enhanced PIM instance recovered for *recommendation-service* microservice from case study 4 based on enhanced PIM metamodel (version 5). ..181
Figure 6-22: MiSAR mapping rule metamodel 184
Figure 6-23: Recovered PSM instance of Microservices Sample....192
Figure 6-24: Recovered PIM instance of *Microservices Sample.* 193
Figure 6-25: Architecture diagram at architectural level of the recovered PIM instance. 195
Figure 6-26: Microservice diagram of recovered PIM instance of service-one microservice. 196
Figure 6-27: Recovered PIM Instance of service-one microservice. 196
Figure 6-28: Documented architecture diagram of Microservices Sample provided by the developer. 198
Figure 6-29: Documented architecture diagram of service-one microservice in Microservice sample. 199
Figure 7-1: The relationship between transformation rules, models, metamodels. .203
Figure 7-2: The four classes of Ecore used in the Ecore implementation (Barendrecht, 2010). 204

Figure 7-3: Ecore implementation (XMI tree view) of MiSAR PSM. 204
Figure 7-4: Ecore implementation (XMI tree view) of MiSAR PIM. 205
Figure 7-5: Ecore implementation (Ecore diagram) of MiSAR metamodel at PSM level. 206
Figure 7-6: Ecore implementation (Ecore diagram) of MiSAR metamodel at PIM level. 207
Figure 7-7: A mapping rule instance. 217
Figure 7-8: Lines in POM file of the “account-service” project that originated the mapping rule. 218
Figure 7-9: An alternative mapping rule instance. 219
Figure 7-10: Lines in configuration file of the “service-one” project that originated the alternative mapping rule. 219
Figure 7-11: MiSAR mapping rules classification. 221
Figure 8-1: Steps of the MiSAR architecture recovery process. 225
Figure 8-2: TrainTicket architectural diagram. 228
Figure 8-3: Artefact collection. 229
Figure 8-4: Resulting PSM model as viewed in Eclipse QVTo project (from left to right). 230
Figure 8-5: Docker Container definition instance retrieved for “ts-auth-service” container. 232
Figure 8-6: Lines that generated Docker Container definition instance for “ts-auth-service”. 232
Figure 8-7: DependencyLibrary instance retrieved for “ts-auth-service” Spring Java project. 232
Figure 8-8: Lines that generated DependencyLibrary instance for “ts-auth-service”. 233
Figure 8-9: ConfigurationProperty instance retrieved for “ts-auth-service” Spring Java project. 233
Figure 8-10: Lines that generated ConfigurationProperty instance for “ts-auth-service”. 234
Figure 8-11: DependencyLibrary instance retrieved for “ts-auth-service” Spring Java project. 234
Figure 8-12: Lines that generated DependencyLibrary instance for “ts-auth-service”. 235

Figure 8-13: JavaClassType instance retrieved for “ts-auth-service” Spring Java project............236
Figure 8-14: Java source code that generated the Java Class Type instance for “ts-auth-service”............236
Figure 8-15: DockerContainerDefinition instance retrieved for non-JVM projects. 237
Figure 8-16: MicroserviceProject instance retrieved for non-JVM projects. 237
Figure 8-17: PIM model for TrainTicket recovered by MiSAR. 239
Figure 8-18: Infrastructure microservice instance recovered for “ts-auth-service”. 240
Figure 8-19: Microservice instance recovered for “ts-ui-dashboard”. 242
Figure 8-20: Microservice instance recovered for “ts-voucher-service”. 243
Figure 8-21: Microservice instance recovered for “ms-monitoring-core”. 243
Figure 8-22: Microservice instance recovered for “ts-ticket-office-service”. 244
Figure 8-23: Microservice instance recovered for “ts-news-service”. 245
Figure 8-24: Microservice instance recovered for “jaeger”. 245
Figure 8-25: TrainTicket’s documentation for “ts-auth-service”............248
Figure 8-26: Generating PSM attribute for recovered infrastructure pattern component element. 250
Figure 8-27: Generating PSM attribute for recovered service dependency element. 250
Figure 8-28: TrainTicket’s documentation for “ts-ui-dashboard”. 251
Figure 8-29: TrainTicket’s documentation for “ts-voucher-service”............257
Figure 8-30: Generating PSM attribute for recovered service dependency element. 258
Figure 8-31: Docker Container Link instance retrieved for “ts-voucher-service”...259
Figure 8-32: Lines that generated DockerContainerLink instance for “ts-voucher-service”............259
Figure 8-33: TrainTicket’s documentation for “ts-ticket-office-service”. 263

List of Tables

Table 2-1: The research questions and their motivations..27
Table 2-2: The selection criteria. 28
Table 2-3: Publications selected...29
Table 2-4: Keywords associated with the challenges in the literature. 34
Table 2-5: The diagrams used in the literature and their annotations. 37
Table 2-6: The quality attributes mentioned in the literature and their alternatives. .39
Table 3-1: Research questions guiding the study..71
Table 3-2: The main mapping rules for transformation (written in a natural language) (El Beggar et al., 2013). 77
Table 3-3: Transformation mappings between the PIM and PSM metamodels (Akkiraju et al., 2012). 81
Table 3-4: Mapping of the extracted information and the MicroART-DSL(Granchelli, Cardarelli, Francesco, et al., 2017). 84
Table 3-5: Comparison of the related approaches..95
Table 5-1: The research questions and their motivation 108
Table 5-2: The selection criteria 109
Table 5-3: Studies selected for analysis 110
Table 5-4: Technology mapping to microservice concerns 119
Table 5-5: Mapping rules to identify API Gateway. 131
Table 5-6: Mapping rules to identify containerisation. 132
Table 5-7: Mapping rules applied for *registry* microservice in case study 1 (PiggyMetrics). 133
Table 6-1: Selected systems 153
Table 6-2: Mapping rules applied in edge-service in case study 7 157
Table 6-3: Mapping rules applied in bookstore-consul-discovery in case study 9 ..158
Table 6-4: Mapping rules applied in Kafka in case study 4.....................................162
Table 6-5: Technologies encountered in the systems analyzed 182
Table 6-6: Sample of MISAR mapping rules structured dataset. 185
Table 6-7: Sample of mapping rules analysis 188
Table 6-8: Artefacts collected for Microservices Sample from its GitHub repository 191
Table 7-1: Resolve functions (Barendrecht, 2010). 214

Table 8-1: All recovered PIM elements for “ts-auth-service”. 240
Table 8-2: All recovered PIM elements for “ts-ui-dashboard”...............................242
Table 8-3: All recovered PIM elements for “ts-voucher-service”. 243
Table 8-4: All recovered PIM elements for “ms-monitoring-core”........................244
Table 8-5: All recovered PIM elements for “ts-ticket-office-service”....................244
Table 8-6: All recovered PIM elements for “ts-news-service”...............................245
Table 8-7: All recovered PIM elements for “jaeger”...245
Table 8-8: Expected elements for “ts-auth-service” as per the documentation vs MiSAR result. 249
Table 8-9: Expected elements for “ts-ui-dashboard” as per the documentation vs MiSAR result 251
Table 8-10: Expected elements for “ts-voucher-service” as per the documentation vs MiSAR result. 258
Table 8-11: Expected elements for “ms-monitoring-core” as per the documentation vs MiSAR result. 261
Table 8-12: Expected elements for “ts-ticket-office-service” as per the documentation vs MiSAR result..263
Table 8-13: Expected elements for “ts-news-service” as per the documentation vs MiSAR result. 265
Table 8-14: Expected elements for “jaeger” as per the documentation vs MiSAR result. 266
Table 8-15: Evaluation metrics for MiSAR recovery of TrainTicket system..........268
Table 8-16: Time spent at each stage of MiSAR recovery for TrainTicket............269

Chapter 1

Introduction

1.1. Introduction

Software architecture is the backbone of any software system. It is an important asset for many software engineering activities, including migration, impact analysis, and system knowledge and maintenance. Software architecture refers to a software system's high-level structure and the particular discipline of designing such structures, as well as the proper reporting of them (Krikhaar, 1997).

Although software architecture has a variety of definitions, it is clear that it is a core aspect of any software system. The most widely accepted definition is that of Bass et al. (2003, p. 21), which states that software architecture is the 'the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them'. Based on this definition, there are three main aspects of software architecture: a structure or structures, elements, and relationships. A structure can be defined as a number of components or elements related to each other and that are held together. Software systems are comprised of many structures (Mens et al., 2010). As such, a single structure cannot be considered architecture. Relationships refer to how different elements within a structure and different structures within a system affect each other, and influence how the structure or system functions.

During the last decade, software technology industries have witnessed rapid development and a dramatic increase in demand for new systems and software. This has boosted the technology economy. A software application will go through continuously evolving changes in its lifecycle, and with every change in an application there will be a resulting change and evolution in its architecture. Not only do architectures change, but developers and teams also change, and where understanding of a current architecture is implicit, knowledge can be lost as the documentation may be either incomplete or outdated (Pashov and Riebisch, 2004).

As a software application evolves, its conceptual architecture often no longer represents the true nature of its implemented architecture. This phenomenon is known as architectural erosion, drift or mismatch (Ducasse and Pollet, 2009). It is therefore hard for developers to be able to successfully refactor, migrate, upgrade or change a system, as they need to have a holistic understanding of the system, and it is critical that they understand its underlying structures (Garlan, 2000). To overcome these issues, the technique known as software architecture recovery (SAR), or reverse architecting, has recently received considerable attention (Ducasse and Pollet, 2009; Ali et al., 2018). SAR is used for the purpose of obtaining the actual architectural structure of a large system, and for obtaining a description of the software architecture from system artefacts such as code. This allows developers to have control over and understanding of improvements in the system and software.

Software companies today emphasise continuous delivery in order to provide increased value to their customers. Microservice architecture (MSA) is among the most effective strategies for achieving this (Pahl et al., 2018). Many enterprises have built software using the microservice architectural style and it is becoming increasingly popular. Microservices are characterised as having fast release and development lifecycles, which is helpful when new features have to be introduced regularly in order for a business to stay competitive (Lewis and Fowler, 2014).

The MSA style views each system as a collection of small-granularity and independent service components, where each service component contains one or more modules and all modules are microservices (Dragoni et al., 2017). According to Namiot and Sneps-sneppe (2014), microservice architecture is a software architecture pattern which helps in developing a distributed application that contains a number of small independent components that can be termed microservices. ‘Microservice’ can be considered a new term in reference to software architecture, and demonstrates a new trend in the field (Newman, 2015). There were no major studies of the microservice phenomenon before 2014, because of a lack of consensus (Pahl and Jamshidi, 2015). After 2015, as shown in Figure 1-1, Google searches have started to reflect the perspective of MSA as a growing concept.

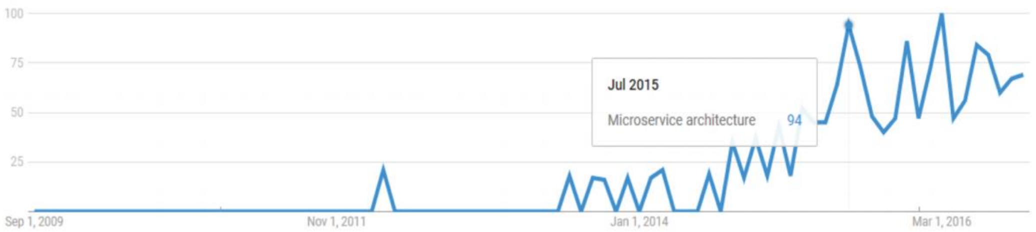


Figure 1-1: The growing number of searches for the keywords 'microservice architecture', according to a Google Trends report.[1]

The MSA style emerged due to the dynamics and pace of today's world. Software applications need to keep up with the fast pace of change and be as agile as possible, to maintain themselves firmly in the market, accommodate ever-changing business needs, and satisfy their diverse clients, such as desktop/mobile browsers, native mobile applications and third parties through APIs. It is very difficult to fulfil these requirements using monolithic applications. This has led to an architectural shift from the monolithic style -a single logical executable- to the microservice style (Lewis and Fowler, 2014; Rahman and Gao, 2015a). MSA evolved from traditional service-oriented architecture and is used to perform highly cohesive business functions compared to SOA. SOA focuses on a wider scope of enterprises services while the microservices focus on doing one thing well (Newman, 2015). In order to perform these functions, microservices divide a system into small fragments, making this the dominant architectural style (Lewis and Fowler, 2014).

Various benefits are encountered in the utilisation of MSA, such as reliability, increased agility, resilience, scalability, developer productivity, maintainability, separation of concerns and ease of deployment (Lewis and Fowler, 2014; Newman, 2015; Daya et al., 2015). However, MSA introduces a level of complexity into applications. The challenge of not fully knowing a real software architecture and understand its underlying structures is increased due to the nature of MSA, as microservices are dynamic (Woods, 2016), small and distributed, and MSA composed of many microservices which have dependencies among each other and different services may use different technologies (Lewis & Fowler, 2014; Eberhard, 2016).

[1] https://trends.google.com/trends/explore?date=2009-09-01%202016-10-03&geo=US&q=Microservice%20architecture.

Each microservice is fine-grained and running independently of the others, each in their own distributed container. By definition microservices are developed quickly and provides more agility of the system, which result in continuous architectural changes; therefore, it can be stated that not every system is built using a well-documented architecture, and often the documentation of the architecture is not kept up to date, especially if the MSA is operated by multiple teams and disciplines (Sorgalla et al., 2018; Cerny et al., 2018). Moreover, these architectures follow an evolutionary design, which is very hard to manage, and which makes it difficult to keep track of the architectural constraints that may be put in place by architects. Software architects often have little knowledge of the as-implemented architecture of their systems, and often face the challenge of not knowing in detail the underlying structures of the software system architecture (Lewis and Fowler, 2014).

Architecture recovery is a promising approach to aiding comprehension of the complexity of MSA in a way that allows developers/architects to understand an architecture's implemented structure. Architecture recovery is supported by use of the Model-Driven Engineering (MDE) approach (Brambilla et al., 2017). The MDE approach brings various benefits. The main one is that it considers models as first class citizens, which abstract the complexities of the systems and support their comprehension. MDE approach raises the abstraction level of the development lifecycle because it shifts the emphasis from code to models (Kent, 2002; Gaševic et al., 2009)

This thesis outlines the Microservice Architectural Recovery (MiSAR) approach, which supports the recovery of architectural models of microservice systems and that can unveil their architectural aspects. MiSAR follows the MDE paradigm and aims to comprehend the complexities of MSA by developing a bottom-up, model-driven transformation, use of static analyses for recovering architecture by modelling the artefacts themselves. This study focuses on the Platform-Independent Model (PIM) and Platform-Specific Model (PSM) abstraction levels in regard to modelling MSA. These models are the critical core of reverse engineering; PIM supports the architectural model recovered and PSM supports the technology of the implemented microservice system.

To define MiSAR, an empirical study was conducted on eight open-source microservice-based systems implemented in the Java and Spring Cloud frameworks, with the aim of identifying the MiSAR artefacts. The study resulted in initial MiSAR artefacts: the metamodel, which supports the creation of microservice architectural models, and mapping rules, which map microservice source systems into the metamodel. Next, another empirical study on nine open-source microservice-based systems was conducted. The focus here was on refining existing MiSAR artefacts incrementally and achieving improved artefacts. The study resulted in a refined version of the MiSAR artefacts, which are able to generate architectural models of implemented microservice systems. The efficiency and effectiveness of the MiSAR technique were measured and evaluated in a case study, using precision and recall metrics. This case study involved a large open-source microservice system. It demonstrates that MiSAR artefacts can produce effective and expressive architectural models of implemented microservice systems.

1.2. Research Aims and Objectives

The main objective of this research was to develop an approach that allows software engineers to recover the architecture of microservice systems and which addresses the problem of understanding the complexity of microservice architecture. To achieve this objective, the following sub-objectives had to be achieved:

Obj1- To identify the microservice architectural elements/concepts needed to recover a microservice-based system and relate these concepts together.

Obj2- To define mappings between a given microservice architecture and the implementations of that architectural style.

Obj3- To identify the information that needs to be extracted from source artefacts in order to allow microservice software architecture to be properly recovered.

Obj4- To develop a process that allows software engineers to recover the architecture of microservices.

Obj5- To develop a prototype tool/technique that will validate this approach.

Obj6- To validate the approach and technique by conducting empirical studies.

1.3. Research Motivation and Research Questions

The popularity and success of architecture recovery solutions in extracting architectural information is commendably strong in the area of architecture recovery in general (Ducasse and Pollet, 2009; Raibulet et al., 2017). Nonetheless, there is a dearth of available research analysing architecture recovery within the area of microservices. Architecture recovery for microservices is a gap in the current state of the art. This awareness became apparent from a recent literature survey by Di Francesco et al. (2017, p. 1), who report that 'in the literature area only little work on reverse engineering and architecture recovery in microservice architecture has been described', and this was confirmed by my own literature survey (Chapter 2, Section 2.6) (Alshuqayran et al., 2016). The motivation for the work presented in this thesis was to provide architectural recovery support for this emerging architectural style. The work presented here provides an approach to such a recovery process. This research topic is new and academically immature, which means there are great opportunities for further research into the discipline's concepts and various areas.

The main research question of this thesis is: What are the architecture recovery processes that allow software engineers to recover the architecture of microservice systems? This main question is divided into the following sub-questions:

RQ1- What are the microservice architectural elements/concepts that need to be present in metamodels in order to abstract microservice-based systems at a platform-independent model level? (Obj1)

RQ2- What are the mapping rules that can transform microservice-based implementations into architectural models? (Obj2)

RQ3- What are the suitable elements in the source model to be able to create a platform-specific model for the recovery of the architecture model? (Obj3)

RQ4- What is an appropriate process/technique for microservice architecture recovery? (Obj4, Obj5 and Obj6)

1.4. Research Contributions

The main contribution of this thesis is to provide practitioners and researchers with a theoretical background for the recovery process that allows the recovery of architectural models of existing microservice systems. This research will have significant implications for future research, as it presents an effective approach to recovering microservice architectural models. The following are explanations of all the original contributions made by this thesis:

C1: A systematic mapping study: The systematic mapping study presented here outlines the gaps and prevailing trends in microservice architecture research.

C2: A modelling language for microservice architecture: A systematic approach was used in developing a modelling language, known as a metamodel. This approach has been developed empirically to support comprehension of recovered architectural models. It enhances the effectiveness and efficiency of the recovery of architectural models. The use of a metamodel in particular is an effective approach for architectural system recovery.

C3: Mapping rules transformation: Another key contribution of this thesis is a set of mapping rules that transform microservice implementations to architectural concepts. Such mapping rules are essential when developing automatic model transformation and in abstracting the hidden complexities of a software architecture. These mapping rules act as a catalyst that allows tool developers to create reverse engineering tools. In the area of model transformation, this thesis contributes model transformations (Code-to-PSM-to-PIM), which enable the generation of the architectural model from the microservice architecture implementation. The key contribution here lies in the transformations specification which show how the mapping rules of high-level models to actual code is carried out.

C4: MiSAR, a microservice architecture recovery approach: MiSAR is a novel approach that follows a bottom-up approach using model-driven engineering based on static analysis, aiming to generate architecture models of existing microservice applications, and to provide a proper understanding of existing microservice

architectures. I present an in-depth empirical investigation into microservice-based systems for the purposes of defining and evaluating MiSAR. In addition to this, it was verified and proved that MiSAR artefacts are feasible in practice.

C5: Microservice architecture recovery process: The thesis provides a complete description of a microservice architecture recovery process, and provides an application of one full instance of a process that can be used directly for the recovery of software system architectures. I verified that the recovery process is efficient for the architecture recovery of small, medium-sized and larger software systems.

C6: Contributions to Practice: This thesis contributes to the knowledge base concerning microservice architecture recovery. The findings of this thesis have significant implications for academic study. The research findings will be useful for:

a) Creating architecture documentation: The approach offers comprehensive knowledge and detailed information pertaining on individual microservices and its components and offers the overall view of all microservices in the system architecture, their types and dependencies.
b) System understanding: A software system data is captured on a fine-grained level and is depicted at an optimum level of abstraction, this approach is necessary to explain the design of the software and provide information on microservices and its dependencies. This knowledge assists in understanding the structure of the system, to reason about its components and properties, help in taking decisions considering the constraints of design, and to check the conformance of architecture recovered against the planned system architecture.
c) Choosing existing reverse engineering tools: Considering the engineers' need for reusing and further prolonging the current reverse engineering tools, it is mandatory to compare them and choose the most suitable piece after ensuring the successful fulfilment of their objectives. MiSAR can be helpful in the comparison of the criteria with the metamodels, model transformation on the grounds of the features construed in MiSAR.
d) MiSAR can act as a reference for reverse engineering groups of practitioners and researchers. It provides the community with an important body of

knowledge to guide future research on metamodels, transformation rules and the corresponding reverse engineering tools.

1.5. Thesis Structure

The following map of the thesis, as also shown in Figure 1-3, indicates the thesis chapters and the relationship between them.

Chapter 2, A Systematic Mapping Study in Microservice Architecture, introduces the literature review performed in regard to the microservice architecture style.

Chapter 3, Background and Related Work provides an overview of related work in the areas of microservice architecture, software architecture recovery and model-driven engineering.

Chapter 4, Research Methodology, describes the research methodology used to accomplish the objectives of this thesis.

Chapter 5, MiSAR: Microservice Architecture Recovery Approach, provides an overview of MiSAR. This chapter presents the first version of MiSAR produced by conducting an empirical study to define the initial MiSAR artefacts (metamodel and mapping rules) that support the architectural recovery of microservice systems.

Chapter 6, Improving the Initial Artefacts of MiSAR: An Empirical Study, presents a second empirical study, which validates and enhances the initial artefacts of MiSAR presented in Chapter 5, in order to define the final version of MiSAR.

Chapter 7, MiSAR Implementation, describes how the metamodels and mapping rules were implemented using MDE tools and standards.

Chapter 8, Evaluation of MiSAR via a Case Study, assesses the proposed MiSAR approach and its outcomes.

Chapter 9, Discussion, discusses the results of the empirical studies and states the key findings of this thesis.

Chapter 10, Conclusion, summarises the outcomes of this research and outlines the essence of the thesis by briefly discussing the outcomes. It highlights the limitations of the research and proposes directions for future studies.

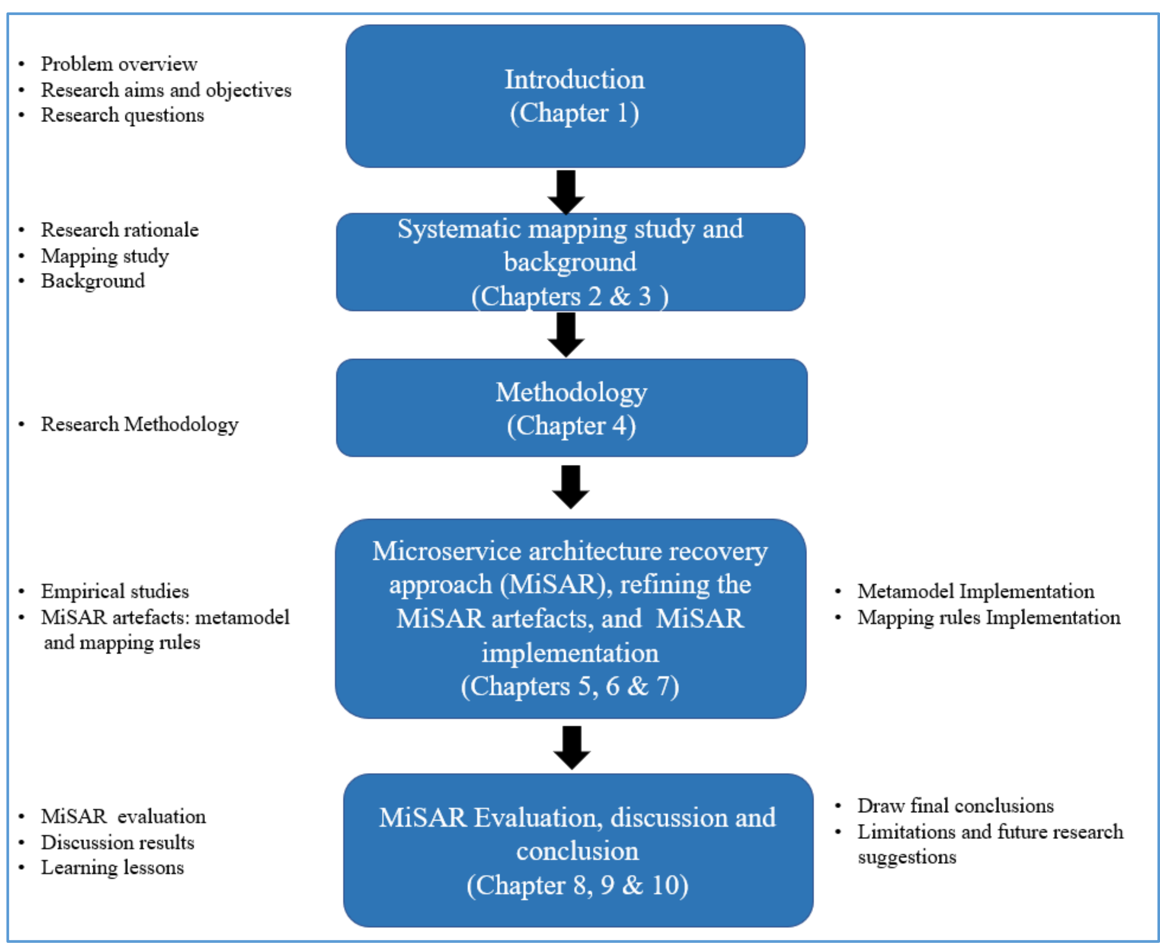


Figure 1-2: Thesis structure diagram.

1.6. Publications

The following publications were a result of research performed in this thesis:

Alshuqayran, N., Ali, N., and Evans, R., 2016, November. A Systematic Mapping Study in Micro Service Architecture. In Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA) (pp. 44-51). Cited 188 times as of May 5th, 2020. (Chapter 2.)

Alshuqayran, N., Ali, N., and Evans, R., 2018, April. Towards Micro Service Architecture Recovery: An Empirical Study. In 2018 IEEE International Conference on Software Architecture (ICSA) (pp. 47-4709). Cited 6 times as of May 5th, 2020. (Chapter 5.)

Empirically Defining a Model-Driven Microservice Architecture Recovery Approach. Submitted to IEEE Transactions on Software Engineering (TSE). (Chapter 6 and Chapter 7.)

Chapter 2

A Systematic Mapping Study in Microservice Architecture

2.1. Introduction

The primary goal of this chapter is to identify whether there is a gap in the state of the art in the area of microservice architecture recovery at the time of conducting this work. As the microservice architecture area, is an emerging one, the secondary goal is to obtain an overview and explore how previous research has supported microservices. At an initial stage, a systematic mapping study is conducted in order to depict the relevant gap and trends in previous research. I also attempted to discover any specific areas of microservice architecture that were not explored, and identify areas where there is a lack of published research. As this review was conducted at the start of the research, the state of the art review considers only research conducted until 2016. The state of the art review after 2016 that is related to the thesis topic will be presented in Chapter 3.

2.2. The Need for a Systematic Mapping Study

Even though microservices emerged from the software industry and have been the focus of practitioners during the last decade (Newman, 2015; Lewis and Fowler, 2014), academic researchers have not kept pace. They have only recently started investigating this approach and providing original research to support it, such as new methodologies, processes and tools (Newman, 2015). The motivation of this mapping study was the lack of available studies regarding the microservices style. One such study was located (Pahl and Jamshidi, 2015); however, the study was limited to providing a temporal overview of microservice research.

2.3. Research Method

Systematic mapping studies are comprehensive and rigorous reviews of specific research questions in an area or a topic, and aim to identify gaps in the literature and where new or better primary studies are needed (Kitchenham and Charters, 2007). In this section, a systematic mapping study of microservice architecture is presented, following the guidelines outlined in Budgen et al. (2008), Kitchenham and Charters (2007), Petersen and Feldt (2008), and Fernandez et al. (2011). Initially, a set of research questions were drafted for investigation during the study. The motivation behind each research question was reviewed and refined. Subsequently, selected papers were assessed against quality criteria, and a classification scheme was iteratively developed, closely following a synthesis method. In summary, the review was established by conducting the following steps:

A) **Research questions**: The research questions and the motivations are outlined in Table 2-1.

Table 2-1: The research questions and their motivations.

Research Question	Motivation
RQ1: What are the architectural challenges that microservice systems face?	The aim was to explore all the published studies that were relevant to microservices, to highlight their gaps and look for future solution foundations.
RQ2: What architectural diagrams/views are used to represent microservice architectures?	The aim was to identify and investigate the possible methods and models that best describe different aspects and levels of microservice architecture.
RQ3: What quality attributes related to microservices are presented in the literature?	The aim was to recognise and disclose the gaps in current research and hence set the direction for future research.

B) **Search strategy:** The terms ‘micro-service’, ‘micro service’, ‘microservice’ and ‘micro-service architecture’ were searched in articles published in journals, conferences and workshops. Sources from books, theses, talks and blog posts were excluded. The research was restricted to articles published between 2014 and 2016, as there was no consensus on the term microservice architecture in the field prior to that date, according to Pahl and Jamshidi (2015). Three online libraries were used: IEEExplore, ACM DL and Scopus (which includes Springer).

C) **Selection of primary studies:** Before selection, articles were initially cross-checked for relevance against the relevant research questions. The titles, abstracts and keywords were scanned to determine the relevance of the articles, and whether they should be included or excluded for the purposes of this study, based on the criteria listed in Table 2-2. After applying the exclusion and inclusion criteria, a total of 33 articles were collected. Table 2-3 lists all the selected publications.

Table 2-2: The selection criteria.

	Criteria
Inclusion	• Studies presenting a definition of microservice architecture. • Studies that focus on microservice architecture and implementation. • Studies that focus on a platform to run systems following a microservice-style architecture. • Studies that focus on a specific challenge within microservices (e.g. fault tolerance, acceptance testing, etc.). • Studies that implement microservice style architecture for a specific business or technical domain. • Studies that make comparisons between monolithic and microservice architectures.
Exclusion	• Papers using the term microservice but not to refer to the architectural style. • Papers which do not have real data to support the proposed design/methodology/architecture. • Studies that do not have microservices as their primary research topic or analysis. • Studies that focus on platforms that are not primarily designed to run microservices, though they may allow it.

Table 2-3: Publications selected.

ID	Paper Name	Format
1	Sustaining runtime performance while incrementally modernizing transactional monolithic software towards microservices (Knoche, 2016).	Conference
2	The hidden dividends of microservices (Killalea, 2016).	Journal
3	An architecture for self-managing microservices (Toffetti et al., 2015).	Workshop
4	Synapse: A microservices architecture for heterogeneous-database web applications (Viennot et al., 2015).	Conference
5	A reference architecture for real-time microservice API consumption (Gadea et al., 2016).	Workshop
6	Exploring the impact of situational context: A case study of a software development process for a microservices architecture (Rory et al., 2016).	Workshop
7	Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud (Villamizar et al., 2015).	Conference
8	Microservice-based architecture for the NRDC (Le et al., 2015).	Conference
9	Container and microservice driven design for cloud infrastructure DevOps (Kang et al., 2016).	Conference
10	Scalable microservice-based architecture for enabling DMTF profiles (Malavalli and Sathappan, 2015).	Conference
11	Experience on a microservice-based reference architecture for measurement systems (Vianden et al., 2014).	Conference
12	Microservice based tool support for business process modelling (Alpers et al., 2015).	Workshop
13	Designing a smart city Internet of Things platform with microservice architecture (Krylovskiy et al., 2015).	Conference
14	Microservices (Thones, 2015).	Journal
15	A reusable automated acceptance testing architecture for microservices in behavior-driven development (Rahman and Gao, 2015b).	Conference
16	Microservices architecture based cloudware deployment platform for service computing (Guo et al., 2016).	Conference
17	Security-as-a-service for microservices-based cloud applications (Yuqiong et al., 2015).	Conference
18	CYCLOPS: A microservice-based approach for dynamic rating, charging and billing for cloud (Patanjali et al., 2015).	Conference
19	Microservices validation: Mjolnirr platform case study (Savchenko et al., 2015a).	Conference
20	Data-Driven workflows for microservices: Genericity in Jolie (Safina et al., 2016).	Conference
21	Distributed systems of microservices using Docker and Serfnode (Stubbs et al., 2015).	Workshop
22	Location and context-based microservices for mobile and Internet of Things workloads (Bak et al., 2015).	Conference
23	Performance evaluation of microservices architectures using containers (Amaral et al., 2016).	Conference
24	CIDE: An integrated development environment for microservices (Liu et al., 2016).	Conference
25	Microservices and their design trade-offs: a self-adaptive roadmap (Hassan and Bahsoon, 2016).	Conference

26	SeCoS: Web of Things platform based on a microservices architecture and support of time-awareness (Zeiner et al., 2016).	Journal
27	Apache airavata as a laboratory: Architecture and case study for component-based gateway middleware (Marru et al., 2015).	Workshop
28	Microservices validation: Methodology and implementation (Savchenko and Radchenko, 2015).	Workshop
29	Learning-based testing of distributed microservice architectures: Correctness and fault injection (Meinke and Nycander, 2015).	Conference
30	Automated deployment of a microservice-based monitoring infrastructure (Ciuffoletti, 2015).	Journal
31	A microservice approach for near real-time collaborative 3D objects annotation on the web (Nicolaescu et al., 2015).	Conference
32	Multi cloud deployment with containers (Jambunathan and Kalpana, 2016).	Journal
33	Migrating healthcare applications to the cloud through containerization and service brokering (François et al., 2015).	Conference

D) **Key wording and classification:** Once papers were selected, a qualitative assessment was conducted to create an outline model for the quality of work. This helps to abstract various possible dimensions for characterisation and categorisation. As a result, the research classification approach performed in Wieringa et al. (2006) was found to be generally applicable for this research and was used to classify the papers as: evaluation research, opinion paper, solution proposal, experience paper, validation research and philosophical paper. Subsequent to the first round of review, the following keywords were identified to be mapped and linked to the challenges of creating microservice-style systems (RQ1).

- **Communication/integration:** Communication and integration have many facets in a microservice-style architecture. Defining a correct communication strategy is vital to the design. The strategy involves identifying the right protocol, response time expectations, timeouts and API design.

  Keywords: API, REST, sockets, TCP, gateway, circuit breakers, load balancer, proxy.

- **Service discovery:** This is the ability of various services to discover each other in a consistent manner. It is important for a system to have a standard and consistent process via which services can register and announce themselves. This helps the consuming services to discover the endpoints and locations of other services. It also involves deciding the right consumer strategy and specifying how API gateways are configured to report service availability and discovery.

  Keywords: discovery, registration, service registry.

- **Performance:** It was commonly observed that introducing microservice architecture to the software industry often adds more 'chatty' communication between different services. For example, fulfilling one single business functional requirement would result in orchestrating multiple service calls together, which in return introduces additional lag to the end-user experience. Due to bounded contexts, data that is frequently used by a single microservice is often owned by another. This requires creating data sharing and synchronisation primitives to avoid the communication overhead caused by data copying, which happens during the service invocations.

  Keywords: QoS, performance, service-level agreement (SLA), speed, simulation.

- **Fault tolerance:** This is the ability of a system to recover from a partial failure. It is up to microservice developers to take this into consideration and provide proper mechanisms to gracefully recover or stop any failure from cascading or migrating to other parts of the system. This is normally expected in cloud environments where infrastructure as a service (IaaS) causes inevitable failures.

  Keywords: fault, failure, recovery, tolerance.

- **Security:** Security is a major challenge that must be carefully thought through in microservice architecture. Services communicate with each other in various ways, creating a trust relationship. For some systems, it is vital that a user is identified in all the chains of a service communication happening between microservices. OAuth and OAuth2 are well-known solutions that are employed by designers to handle security challenges.

  Keywords: secure, authentication, authorisation, OAuth, OAuth2, encryption, vulnerability, attack.

- **Tracing and logging:** In microservice-based systems, tracing and central logging are vital for developers to understand the system's behaviour as a whole. Breaking up monolithic systems into microservices involves techniques that are traditionally employed for debugging and profiling systems. Various techniques and solutions are emerging to solve this problem. Distributed tracing is the ability of a system to track a chain of service calls to identify a single transaction or a single user request. Logging is another critical component of any system. Logs are important for auditing and debugging purposes. Special attention must be paid in carefully designing a central logging and aggregation system for developers to continue debugging systems in an appropriate manner.

  Keywords: tracing, logging, debugging, profiling.

- **Application performance monitoring (APM):** APM is an infrastructural characteristic. It involves measuring individual microservices' performance to assess the health of and existing SLAs for a system.

  Keywords: monitoring, APM, health monitoring.

- **Deployment operations:** Deployment operations and scaling are fundamental infrastructure concerns. Selecting the right platform significantly influences the architecture of a microservice system. Container orchestration tools and structured platform as a service (PaaS) solutions provide various features that make deployment and operations very trivial activities. However, selecting the right solution is critical, as all of these platforms come with their own set of assumptions and opinions, which the designer has to follow in order to utilise

the selected platform to its potential. Scaling microservices can become a challenge if the right architecture is not followed. There are several guidelines, such as 12-factor application and cloud-native designs[2], which need to be followed to make service scaling easier. Most of these guidelines demand statelessness and portability by decoupling service runtime from OS and platform resources.

Keywords: operations, orchestration deployment, scaling, auto-scale, rolling upgrades, images, container.

E) **Data extraction strategy and quality assessment:** Data for this study was extracted using a machine learning-based PDF extraction library called GROBID,[3] which extracts the PDF data into structured TEI-encoded documents, with a particular focus on technical and scientific publications. Kibana[4] was used to perform the visual analysis and generate various charts. First, the selected keywords for the 'challenges' part were analysed using Kibana visualisation, in order to understand their distribution in the underlying population. This gave a quantitative indication of the possible classifications. The selected papers were then classified based on a review of actual content where the research questions are the driving element of the analysis.

2.4. Results

- *Significant keywords*

At a high level, the following are the most significant keywords from the previously mentioned keyword lists. Figure 2-1 lists the top terms found in the literature. It can be observed that 'deployment', 'cloud' and 'performance' are the words that dominate the papers; 'deployment' is the most discussed topic, appearing in 31 out of 33 papers, followed by 'cloud' and 'performance', in 23 papers.

[2] https://12factor.net./
[3] https://github.com/kermitt2/grobid
[4] https://github.com/elastic/kibana

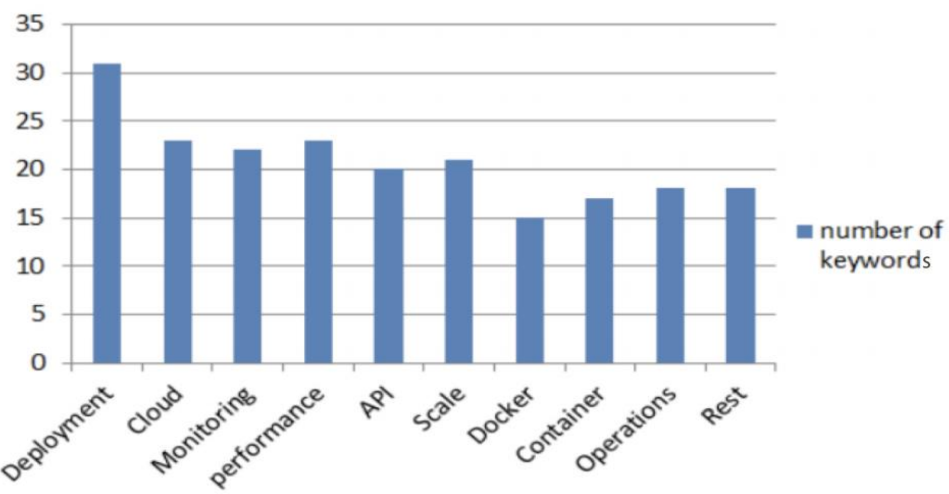


Figure 2-1: The top 10 keywords in the literature.

➤ *Challenges of microservice system architecture (RQ1)*

I identified papers which actively address one or more of the challenges mentioned in Section 2.3, D. The classified papers either present a solution, address a challenge as their primary or secondary topic, or discuss a challenge to a certain depth. Furthermore, I quantitatively searched for earlier presented keywords associated with the challenges in the papers, and presented the count of papers mentioning one or more of those keywords. Table 2-4 and Figure 2-2 show the results of the above classification.

Table 2-4: Keywords associated with the challenges in the literature.

Challenges	Keywords	Mentions
Communication/Integration	API, REST, sockets, TCP, gateway, circuit breakers, load balancer, proxy, routing, router	29
Service Discovery	Registration, service registry	11
Performance	QoS, performance, SLA	28
Fault Tolerance	Fault, failure, recovery, tolerance, healing	23
Security	Secure, authentication, authorisation, OAuth, OAuth2, encryption, vulnerability, attack	13
Tracing and Logging	Tracing, logging, debugging, profiling	8
Application Performance Monitoring	Monitoring, application performance monitoring	24
Deployment Operations	Operations, orchestration deployment, configuration, scaling, auto-scale, rolling upgrades, images, container	34

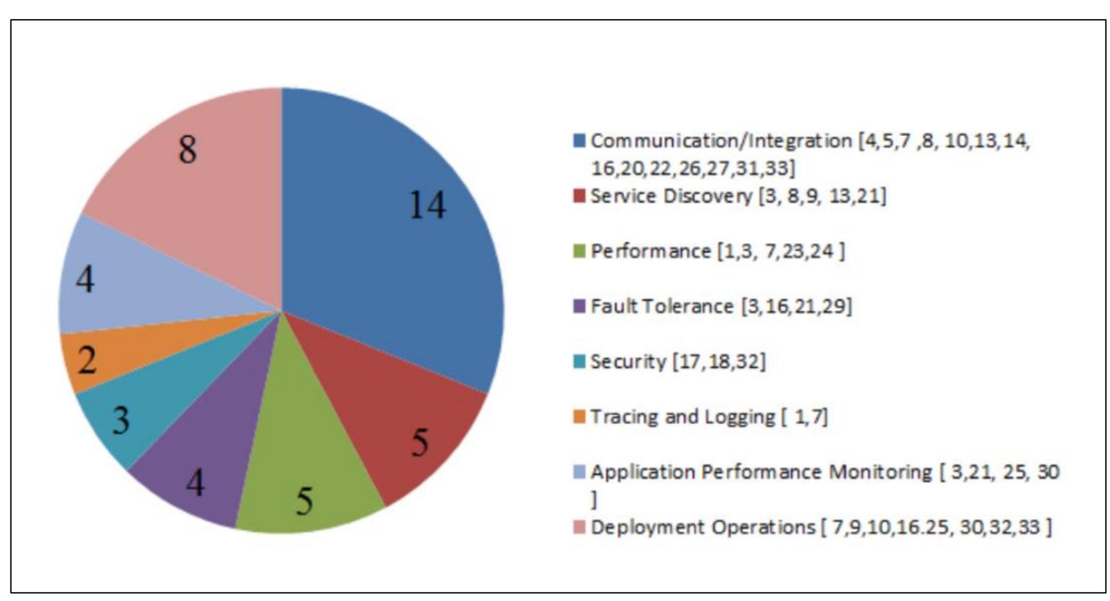


Figure 2-2: The distribution of microservice challenges in the literature.

- *Research paper approaches*

Papers were classified using approaches presented in Wieringa et al. (2006). Since the microservice architectural style is an emerging field, a lot of research is focused on presenting evaluation research or solution proposals, followed by validation research. A lack of experience reports and opinion papers is also a clear indication of the emerging nature of the research. Figure 2-3 presents the approaches plotted against the number of papers with different challenges, which gives a combined view of the selected studies and their distribution over these two dimensions. The size of the bubble represents the number of papers. It can be observed from the figure that 'communication' and 'deployment' are well ahead of the other challenges. It can also be noticed that the 'communication' and 'deployment' challenges have more validation and evaluation papers.

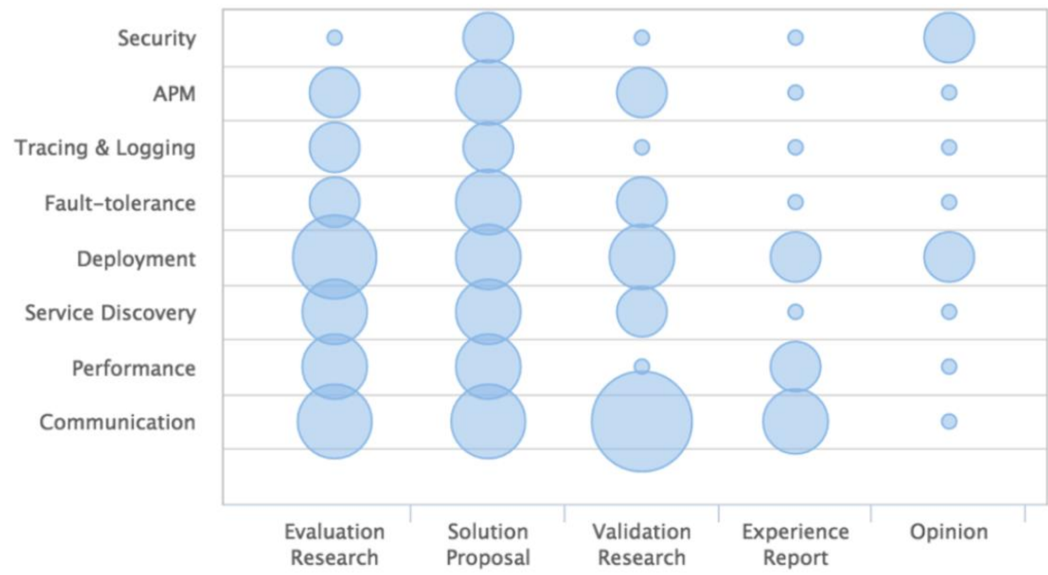


Figure 2-3: Research approaches against the number of papers with different challenges.

- *Microservice architectural views/diagrams (RQ2)*

Solution proposal and validation research types of papers were the main sources to answer this question, as they paid more attention to architectural modelling than other papers. In particular, the design and implementation sections of these papers provided figures with views/diagrams, along with their detailed explanations. However, although component/context diagrams were dominant in the literature, a wide variety of other graphical modelling views was also presented, although with no clear justification provided for the choice of a particular diagram. This lack of consistency in diagrammatic presentation may indicate a need to propose a comprehensive modelling view/language that best covers and describes microservice-based architecture. The graphical architectural views found in the literature were various and can be categorised into informal drawings with free boxes and lines, sets of UML diagrams each covering different aspects of the architecture, graphs with vertices and arrows, and finally diagrams for SQL/NoSQL relational databases.

Table 2-5 shows the diagrams used in the literature and their annotations, and sets of papers that included each type of diagram. Interestingly, it was noticed that there is no distinction between component diagrams and container diagrams in the literature. This implies that the trend in microservice architecture is to suggest placing one microservice, i.e. component, in one running environment, i.e. container, in order to achieve the ultimate independence and isolation of the microservices. In addition to the description diagrams covered earlier, description languages are also included in the literature to provide a more elaborate view of architecture details. Categories of different formats of description languages mentioned in the study included:

- Standard modelling languages, e.g. RAML and YAML.
- Specifically-designed modelling languages, e.g. CAMLE.
- Standard specification languages, e.g. JavaScript (Node.js), JSON, Ruby.
- Specifically-designed specification languages, e.g. Jolie.
- Pseudocode for algorithms.

Table 2-5: The diagrams used in the literature and their annotations.

Diagram	Annotation	Paper ID numbers
Component/ Container	Each microservice is represented as a square/rectangle/oval and each line represents communication or data flow between components	1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32 and 33
Process/Behaviour	Each process of a microservice is a rounded square/rectangle and each arrow indicates an activity flow	1, 9, 11, 15, 22, 27 and 28
Sequence	UML diagrams	3, 5 and 18
Execution Timeline	Time grows from left to right on the x-axis	4 and 9
Deployment	UML diagrams. Execution is represented as a rectangle parallel to the x-axis	7 and 32
Class	UML diagrams	20 and 30
Use Case	UML diagrams	28 and 33
Type Graph	Represents the needed resources and connection topology for each node (orchestrator) where cardinalities on edges represent the minimum and maximum number of allowed connections	3
Instance Graph	Represents the deployed service topology and components for each node (orchestrator)	3
Dependency Graph	Each node in the graph represents a microservice and an arrow indicates a dependency	4

- *Quality attributes related to microservices in the literature (RQ3)*

To approach this question, a generalisation of attribute names was necessary at first, since many alternative terms found in the papers indicated the same meaning for one attribute. Table 2-6 shows each attribute and its alternative terms. It was noticed in the literature that well-known quality attributes of microservice architecture, such as modularity, scalability, independence and maintainability, were presented in almost all of the papers reviewed. Some attributes scored fewer occurrences, which implies lack of consideration. These attributes were security (IDs 15, 18, 23, 32, 33), load balancing (IDs 1, 20) and organisational alignment (IDs 13, 15).

In addition to the results of research questions 2 and 3 above, a possible relationship between quality attributes in the literature and the model views presented was investigated. For each quality attribute, the modelling diagrams included in the papers mentioning that attribute were checked. This intersection method attempted to answer what type of modelling view is more suitable to demonstrate and/or test particular quality attributes in the architecture. More elaboration on the findings and insights derived from the results is provided in the next section.

Table 2-6: The quality attributes mentioned in the literature and their alternatives.

Attribute	Number of Papers	Alternative Terms and Expressions of Similar Meaning
Scalability	26	Expandable, evolutionary
Independence	19	Reducing complexity, isolation, loose coupling, decouple, distributed, containerisation, autonomy
Maintainability	17	Expandable, adaptability, changeability, flexible implementation, dynamically changing
Deployment	13	-
Health Management	13	Resilience, reliability, disaster recovery, no single point of failure
Modularity	13	Single responsibility, reduce complexity, separate business concern, specialisation, customisable
Manageability	12	Self-managed, decentralised management, audibility
Performance	9	Response times, transaction duration, throughput, efficiency
Re-usability	7	Pluggable
Technology Heterogeneity	7	Portability, freedom to choose a lot of technologies or programming languages
Agility	6	Iterative, incremental, continuous delivery
Security	5	-
Load Balancing	2	Workload intensity distribution
Organisational Alignment	2	Cross-functional team reduce the conflict between developers and testers
Open Interface	1	Microservices should provide an open description of their APIs, GUIs and communication message format

2.5. Discussion

It can be observed from the results of the review that microservices architecture research is still in its infancy. Since the style is born out of industry, it has been noted that there are wide gaps between the current industry level and that of academia. Most of the papers in this study were found to be either at the 'solution proposal' or 'solution validation' stage, with validations based on lab-controlled experiments only. There are very few experience reports and opinion papers on the microservices architectural style.

Moreover, microservice security is a very important challenge, which has not yet been well researched. Even in industry, lots of service-based applications do not employ stringent security controls. It is also noted that tracing is one of the most common problems faced by all microservice-style systems. Tracing a request through all the hoops of business functions is a very difficult problem that demands attention from the academic community. Only a few solutions are currently available in industry[5]. These solutions can help discover communication patterns, which can be used to discover dependencies between services. A dependency graph helps architects in refactoring and making decisions with confidence.

As regards RQ2, the literature presents different types of modelling diagrams and languages that describe aspects of microservice architecture, as well as its lifecycle. Context and container/component diagrams with UML notations, for example, are extensively used to provide a high-level static view of microservice architecture. To describe low-level design details, UML-class diagrams are used, accompanied by ERD data models, pseudocode for algorithms and additional textual description. UML use cases are used mainly for model validation and testing of microservices, whereas UML sequence diagrams are used to sketch the communication between microservices. A particular kind of graph is used for model deployment orchestration and automation, called a type graph/instance graph. Each type graph represents the connection topology and resources needed to deploy a microservice, while instance graphs represent each orchestrator service with its components.

[5] https://zipkin.io.

Interesting modelling languages presented in the literature were RAML, YAML and CAMLE. RAML and YAML (Swagger) are open standard modelling languages used to describe APIs of REST-like messages, needed for interacting and communicating with microservices. CAMLE is a specifically designed conceptual graphical design for service-oriented systems that integrates with a modelling language for agent-oriented systems called CAOPLE. According to the source paper, the CAMLE/CAOPLE modelling method proved its efficiency in modelling the microservice architecture of CIDE, the proposed integrated development environment for building microservice systems. Code snippets of standard specification languages such as JavaScript, JSON, Node.js and Ruby were used to describe the data model of messages communicated between microservices. A novel programming language called Jolie (Safina et al., 2016) was used to program and describe the architecture of its IDE, which is also built using microservices.

Based on the previous analysis, it can be noted that modelling microservices with UML standard notations is comparable to creating another comprehensive modelling notation, and also comparable to the use of informal drawings with free boxes and lines accompanied by a narrative. However, since a typical microservice-based system consists of a number of containers, and each container in turn contains one or more components, i.e. microservices, which in turn are implemented by one or more classes, then UML standard notation can provide a common set of abstractions and notations to describe microservice architecture. Therefore, using several UML diagrams, e.g. context, container, component, class, use case, sequence, each showing a different part of the entire architecture, will allow one to communicate software designs in an effective and an efficient way.

The results of RQ3, as in Table 2-6, show higher occurrences of, and hence more focus on, scalability, re-usability, performance, fast and agile development, and maintainability. On the other hand, fewer occurrences, implying the need for future research, were found for security, load distribution (for multi-cloud deployment with containers), continuous integration, organisational management and DevOps, as well as the automation of container management and deployment.

Finally, having investigated the view model to quality attribute papers' overlap, the following findings have emerged:

(i) Papers concerning scalability, re-usability, maintainability, manageability and deployment quality attributes also used component/container, class and deployment UML diagrams to demonstrate the potential of implementing those attributes;
(ii) Use case and sequence UML diagrams, in addition to execution timelines, helped in comparing and validating the quality attributes of performance, deployment, security, maintainability and self-manageability of microservice architecture;
(iii) Instance graphs/type graphs enabled the author of paper [ID=3] to trace and validate the quality attributes of health management, manageability and deployment automation;
(iv) Dependency graphs co-occurred with independence and maintainability quality attributes, and were also used to trace and test them.

The literature suggested many future trends, such as the following:

(i) Invent and automate approaches to empower the DevOps team, so that development and operation functions are cooperative, hence enabling the rapid and agile development and upgrade of applications, as well as deploying them on multiple platforms to meet customer needs;
(ii) Investigate the impact of the interrelationship between a process (service) and its context (situational factors) on microservice software process decisions;
(iii) Allocate a specific programming language, e.g. Jolie IDE, to develop microservices, e.g. CIDE.

2.6. Summary

This systematic mapping study has looked into the available studies on microservice architecture that conducted until 2016. The study used two qualitative and quantitative synthesis methods and addressed three key research questions. The first question addresses the architectural challenges that microservice systems face, and explored all the published articles and studies that highlighted the gaps in microservices research, and made suggestions for future solutions and initiatives. The second research question investigates which architectural diagrams and views, in addition to any methods or models, are used to represent microservice architectures. The last research question states the possible quality attributes related to microservices that are presented in the literature.

With the help of a systematic mapping study, the literature can be thoroughly examined in order to depict the prevailing topics that are discussed in prior microservice research. The outcomes from this analysis have assisted in finding the gaps and the prevailing trends in previous research. After having carefully combed through many research papers, it was concluded that existing literature and research has not paid much attention to microservice architecture recovery. Therefore, I decided to pursue this study to fill this gap in the research. I define the scope of the research as first defining the fundamental concepts and pillars of microservice architecture, and secondly proposing a novel process to recover this kind of architecture.

Chapter 3

Background and Related Work

3.1. Introduction

This chapter provides the foundations, basic concepts and terms which are relevant to the theme of the thesis. The background section is introduced first, followed by a discussion of related work, and then concluding remarks are presented at the end of the chapter. The background section introduces the field of microservice architecture, followed by the basic concepts from the areas of model-driven engineering, reverse engineering and software architecture recovery, and the terminology associated with this field used in my approach, in order to provide the reader with the necessary background information. The related work section analyses and compares the existing gaps in the available approaches and techniques that are related to the topic.

3.2. Background

3.2.1. Microservice Architecture

Over the past few years, the term 'Microservice Architecture' has gained significant momentum as a term describing a new tendency of designing, developing software applications as a set of independent services (Newman, 2015). While no precise definition pertaining to this architectural style has been propounded, there are common characteristics of it revolving around the business capability, automated deployment, and intelligence in the endpoints (Lewis and Fowler, 2014).

The most widely adopted definition of microservice architecture describes it as an approach for designing a single application as consisting of small services, where these services run in own single processes and also interact with lightweight mechanisms, which usually are RESTful APIs, and there is a bare minimum of centralized management of these services (Lewis and Fowler, 2014). According to Newman, (2015), microservice architecture is a new architectural style that consists of

applications, which are an accumulation of independent individual services, and these services are of a single business capability.

The microservice term was first introduced in 2011 in a workshop of software architects held in May 2011, near Venice. The prime agenda of the workshop was to describe and discuss the perspectives of participants regarding architectural style which had been recently exploring (Lewis and Fowler, 2014). As the name indicates, 'micro' refers to a small object; in essence, a mini/smaller version of a service or task (Nycander, 2015).

Before the evolution of the microservice architectural style, most of the well-known Internet companies, like Amazon, Gilt, Sound Cloud, Netflix, etc., were using a monolithic architecture. A monolithic architecture is large, and has all application logics running in a single process and application server (Lewis and Fowler, 2014). A monolithic approach is not suitable for high-volume websites, since for any change, whether small or significant, the whole team need to coordinate and apply the required changes in their application. In a monolithic architecture, it is thus very challenging and costly to upgrade and change applications. In short, most of the big companies have shifted to microservices, this paradigm shift also termed as microservitization (Hassan et al., 2019).

The architecture of the microservice style has provided the solution that was needed to simplify integration and to upgrade the tasks involved, which include: (a) providing greater modularity and scalability (Lewis and Fowler, 2014), (b) providing faster deployment of code changes and service delivery (Gruman and Morrison, 2014), (c) componentization in microservice architecture following a more loosely-coupled approach (Newman, 2015), (d) independence in development and deployment, as promised by this architectural style (Eberhard, 2016), and (e) providing the ability to grow and develop teams in a more efficient manner, eventually achieving agility (Lewis and Fowler, 2014). However, these benefits come with challenges, such as discovering services over the network, security management, communication optimisation, having multiple components within a single architecture, and cooperation and orchestration among microservices (Esposito et al., 2016).

There are a number of similarities between microservices and Service-Oriented Architecture (SOA). The most common similarity is that they are service-based and distributed architectures. This feature allows access to services through remote access protocols, like Representational State Transfer (REST), Simple Object Access Protocol (SOAP), etc. There are many benefits of using distributed architecture over other architectures, such as layer-based and monolithic architecture. These benefits include better decoupling, scalability, and control over the processes of developing, testing and deployment (Richards, 2015). Distributed architecture is comprised of components that are inclined towards being more self-contained, and this allows better control over changes and ease of maintenance. Thus, applications' robustness and responsiveness are increased. In addition to this, a distributed architecture is easily modified with loosely coupled and modular applications. Both SOA and microservice architectural styles help in structuring a particular system by allowing a group of primary architectural components that perform business and non-business functions called services to work together (Richards, 2015).

There is a clear distinction between SOA and microservice architecture (Villamizar et al., 2015). (1) In SOA, applications use heavyweight technologies like SOAP, and other web service standards, including enterprise service bus (ESB), to unite the architectural services. In microservice applications, ESBs are not used and instead lightweight communication protocols such as REST. According to Martin Fowler (Lewis and Fowler, 2014), endpoints are important in microservices and make them different to SOA. Smart endpoint characteristic facilitates the microservices communication between each other. (2) Databases are handled differently, in SOA a global data model is used, with a shared database, while microservice architecture has a database for every service (Villamizar et al., 2015; Savchenko et al., 2015). (3) The size of the service. SOA usually integrates large, complex and monolithic applications. In microservice architecture, services are not always tiny, but they are almost always much smaller than in SOA. As a result, a SOA application usually consists of a few large services, whereas a microservice-based application will consist of 10s or 100s of smaller services (Nycander, 2015). In terms of service granularity, the capability of a software system is differently aligned with each service in SOA, but one capability is aligned with each service in MSA; this alignment is done in MSA to reduce service interactions and overlapping of several services, and relevant concepts of domain and

operations from other services are isolated. By contrast, SOA has a variety of distributed software system capabilities, so no service granularity is followed, and service size varies from fine-grained applications to coarse-grained enterprise services (Rademacher et al., 2017). It is worth mentioning that microservices architecture is not a new idea, as it is built up from its precedents.

3.2.2. Model-Driven Engineering

Model-driven Engineering (MDE) is an approach that considers models as first-class citizens (Brambilla et al., 2017). In the MDE paradigm, the approach emphasises treating everything as a model (Gaševic et al., 2009), which means that models are the artefacts used for describing and developing a system. MDE depends on three key characteristics described in the following section: (a) a model that requires languages for its description, (b) model transformations which define rules and their specification for the purpose of describing the way in which a particular model can be transformed into other models, and (c) metamodels which are models that are used to describe other models using a modeling language. Conformance is defined as the relationship between metamodels and models. A model is considered to conform to or is an instance of a metamodel when a metamodel identifies each concept that is used in defining a specific model, and the models use the concepts in accordance with the patterns specified by the metamodel (Benoit et al., 2016).

MDE emerged from the concept of Model-Driven Architecture (MDA), and was proposed in 2001 by the Object Management Group[6]. Kent (2002) defined MDE by taking MDA as a base and adding the concept of organising models via the software development process and modelling space. Increasing productivity and reducing the time-to-market are the objectives of MDE, in developing complex systems with the help of concepts defined via models. (Benoit et al., 2016; Selic, 2003).

[6] The Object Management Group (OMG) is an industry association for standardisation within software engineering.

3.2.2.1. Models

In MDE, requirements, architecture and implementation are described in a software project through models. These models are used to handle complexity as they can manipulate the system and provide rationality at the conceptual level. Models are also used for generating code, deriving artefacts and for documentation purposes. For example, documentation of a system can include UML models. According to Bezivin and Gerbe (2001), a model is a simplified version of a system constructed with a predefined objective in mind. As per the MDA guide (Miller et al., 2003), a system's model is a method of describing a system and its environment for a specific purpose.

As per MDA, models can be divided into four categories, code, platform-specific model (PSM), platform-independent model (PIM) and computation-independent model (CIM) (Miller et al., 2003), as shown in Figure 3-1. Specifically, the MDA guide describes a CIM as the computed independent view of a system; details of system structure are not shown in a CIM. CIM uses vocabulary known by domain practitioners and hence it is also called as domain model (e.g. description of user requirements). A PSM is a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform. A Platform refers to any technology-specific code, open or proprietary, including Web Services, .NET, CORBA, J2EE, and others (Miller et al., 2003). A PIM abstracts away technical details and does not depend on platforms.

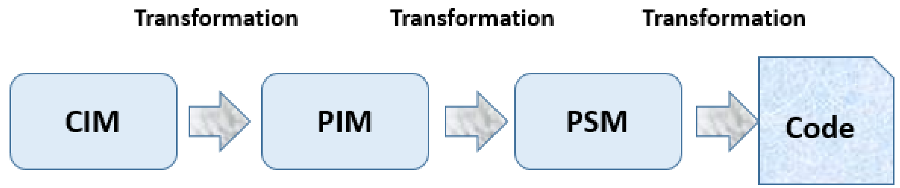


Figure 3-1: The MDA framework (Miller et al., 2003).

3.2.2.2. Metamodel

The process of analysing a specific domain to find concepts, constraints, relationships and rules is called 'metamodelling'. A metamodel is an abstraction that reveals properties of the models themselves. The models are represented as 'instances' of more abstract models. Metamodels are used to define new languages, or to uncover new features or properties of existing data (metadata) (Brambilla et al., 2017).

Metamodels and language are interchangeable due to the fact that the metamodel is also inferred as a language that reveals aspects of the system and process affiliated to a particular domain. Conformance is defined as the relationship between metamodel and model. When a metamodel identifies each concept that is used in defining a specific model, and the models use the concepts defined by a metamodel in accordance with the patterns specified by the metamodel it is considered conformed to a metamodel (Bézivin, 2005). It can also be said that a model can be considered as an instance of a metamodel, which can be changed according to the elements of the abstract syntax that are known as meta-classes.

The OMG (2014) defines four levels for meta-modelling, as depicted in Figure 3-2. The M0 layer, at the bottom level, represents the real system, and the model represents the system at the M1 layer. The conformation of the model to its metamodel is displayed at level M2, and the conformation of the metamodels to their meta-metamodel is shown on level M3. In addition to this, the meta-metamodel conforms to itself at level M3. MOF was proposed by OMG as a standard to define metamodels; for example, MOF has defined the UML metamodel (Di Ruscio et al., 2012).

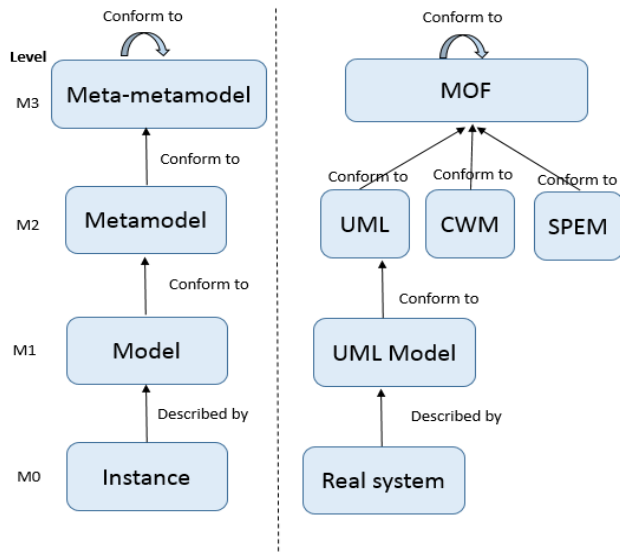


Figure 3-2: The four-layer meta-modelling architecture (Di Ruscio et al., 2012).

3.2.2.3. Model Transformation

According to MDA, in the same system, if one model is converted into another model, this is called model transformation (OMG, 2014). In the same context, Kleppe et al. (2003) define transformation as automatic generation of the model that is targeted from the source model. Transformation rules are used to transform source models into target models. These rules can be written manually by a developer from the ground up, or their definitions can be based on specifications of previously existing rules that have been refined.

As depicted in Figure 3-3, the input is taken by a model transformation program to conform with a given source of a metamodel and generates output as another model, which conforms to the targeted metamodel. The set of rules composed in transformation programs should also be treated as a model; as a result, transformation language depends on the abstract metamodel's definition (Di Ruscio et al., 2012).

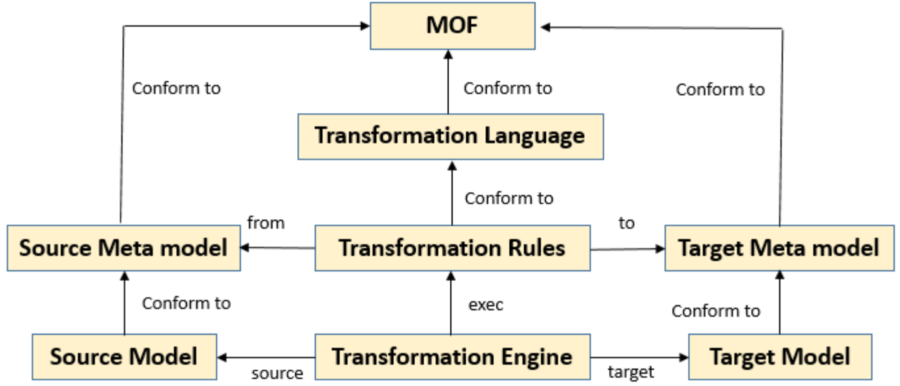


Figure 3-3: Basic concepts of model transformation (Di Ruscio et al., 2012).

3.2.3. Characteristics of Model Transformation Approaches

This section provides a brief account of the key differentiators for model transformation provided in (Czarnecki and Helsen, 2003), which helps in defining the different kinds of method used in model transformation approaches. The primary or core points of variation which are used in model transformation are briefly described in the paragraphs below.

- **Transformation rules:** Transformational rules define the manner through which the elements of source models are translated into the target models. The rule of transformation is classified into two sections, namely the left-hand side (LHS) and right-hand side (RHS). The RHS consists of target model elements, while the LHS consists of source model elements. These can be defined by patterns, variables, queries and logic.

- **Rule application scoping:** Rule application scoping determines the extent of the target model in the transformation, as there is also a restriction for those model parts which are included in the process of transformation. Scoping is vital for the purpose of performance and advance transformation structures.

- **Rule application strategy:** In this approach, it is necessary that the rule is applied within its scope to a specific location. A need for an application strategy has been identified owing to the presence of a large number of matches for the rule in the prescribed source scope, and the application strategy can be further defined as non-deterministic, interactive or deterministic.

- **Rule scheduling:** When the nature of model transformation is complex, the number of rules also increases; therefore, a scheduling method is used to justify the order that is used to apply the specific individual rules. However, in some cases, people have no clear control over the scheduling algorithm. The approaches can be different in terms of the execution of the rules and the techniques used.

- **Rule organisation:** Rules can be shaped or structured in different ways, and model transformation approaches have three types of variation, a re-use mechanism (defining rules based on one or more rules), a modularity mechanism (packaging rules into modules) and an organisational structure (organising rules based on the source or target language or another independent organisation).

- **Traceability links:** The role of transformations is to record links between source and target elements. Such links are valuable for conducting impact analysis (i.e. exploring how alterations to one model affect related models), synchronizing models, debugging models, etc.

- **Directionality:** Directionality is the last item on the list of differentiators of model transformation approaches, and explains that the transformation may be bidirectional or unidirectional. The source model is unidirectional when the transformations occur in a single direction to the target model. Bidirectional transformations are executed in both directions and are beneficial for applying round-trip engineering for the purpose of synchronising the models.

3.2.4. Major Categories of Model Transformation Approach

According to Jilani et al. (2010), transformation approaches are categorised into two major types, as suggested by different authors, the first one is the Model-To-Model transformation (M2M) approach, and the other one is Model-To-Text (M2T) approach. The M2T transformation model is used for transforming the model into the text format or code generation. In contrast to this, the M2M model is used in conditions in which there is a need for transforming the source model into the target model; they are instances of the same or a different metamodel. Detailed outlines of both these models can be presented as follows:

Model-to-text approaches: In these approaches, the PSM transformation technique is generally undertaken for generating codes. Some of the crucial approaches in this category are presented below (Czarnecki and Helsen, 2006):

- **Visitor-based approach**: According to Czarnecki and Helsen (2006), the visitor-based approach is a very basic code generation approach consists in providing some visitor mechanism to traverse the internal representation of a model and write code to a text stream. An example of this approach is Jamda (SourceForge, 2003), which is an object-oriented framework providing a set of classes to represent UML models, an API for manipulating models, and a visitor mechanism (so-called CodeWriters) to generate code.

- **Template-based approach:** This approach is used to generate code by templates which represent the source model, and the templates have rules that are mapped onto the source model. For code generation, the template-based approach is identified as the most precise approach, compared to the visitor-based approach. The result of the code generation is correct and accurate due to the fact that the structure of the template is similar to the code. The tools which are used for the code generation are JET (Popma, 2004) or AndroMDA (AndroMDA.org, 2003).

Model-to-model approaches: In the model-to-model category, Czarnecki and Helsen (2006) distinguish between graph-transformation-based, direct-manipulation, structure-driven, operational, relational and hybrid approaches.

- **Graph-transformation-based approach:** Graph transformation is the most popular technique for the management of transformation. Graph transformation rules involve an LHS graph pattern and an RHS graph pattern. The former often include conditions ancillary to the LHS pattern, such as negative conditions. Some additional logic (such as in string and numeric domains) is needed for the purpose of target attribute values (such as element names). Graph Rewriting and Transformation Language (GReAT) (Agrawal and Aditya, 2003) involves an extended form of patterns with multiplicities on edges and nodes, and is the most common language used by the graph transformation technique.

- **Direct manipulation approach:** In this approach, specific APIs, as well as internal model representation, are offered for the purpose of manipulating models such as JML. This approach is used for the purpose of developing a specific and object-oriented framework. This approach provides a minimal infrastructure, used for organising transformations. For instance, in this category, abstract classes used for facilitating transformation are provided. However, in this category, there is a need for reflecting different crucial perspectives in practice from the beginning, including transformation rules, tracing and scheduling in different programming languages such as Java.

- **Structure-driven approach:** The process of transformation takes place in two distinct phases in this approach. In the first phase, the target model is structured in the form of a hierarchical structure. In the second phase, there are applied several features and references to the target model's hierarchical structure. In this approach, the user is required to pass information related to transformation rules only. Scheduling and application strategies are further determined by the model itself. For instance, OptimalJ is the base example of this approach. In this model approach, the implementation of the

transformation rule is undertaken in the form of a method with an input parameter which defines the type of source and method for returning the Java object, which defines the target model element.

- **Operational approach:** This approach has a similar type of orientation to direct manipulation. Yet, this approach provides extra devoted assistance in model transformation. In this scenario, the extension of meta-modelling formalism is facilitated with expressed computations. For example, the utilisation of a query-based language, such as OCL. QVT operational mappings are considered as the key examples of this approach.

- **Relational approach:** This approach is followed by referring to mathematical relations and source-target relationship, which is a declarative approach (Akehurst and Kent, 2002). QVT relations are considered as the key examples of the approaches of relationship. The core idea is to have constraints of the relationship between the source and target element types. Predicates and constraints are used to describe relationships in mathematics, and complex mathematics is used to construct these relationships. This approach has a feature of supporting backtracking and the rules that are based on the mathematical relationships are bidirectional. The relational approach does not work in the in-place transformation, in comparison with the graph transformation technique.

- **Hybrid approach:** This approach is a combination of different approaches mentioned in the previous headings. In this category, the Transformation Rule Language (TRL) combines imperative and declarative approaches. QVT can be considered an example of the hybrid approach, as it combines three different elements, including operational mappings, core and relations. The Atlas Transformation Language (ATL) is also a hybrid approach, and it has certain similarities to TLR. A transformation rule in ATL may be solely imperative, hybrid or solely declarative.

3.2.5. Model Transformation Languages, Tools and Standards

The presented section has presented an account of the different kinds of model transformational tools and languages.

Query/View/Transformation (QVT): QVT is the standardised language established by the OMG and can be used for model transformation (OMG, 2016). Three types of language defined by QVT for the transformation of the model-to-model are: **QVT Relational**, this high-level declarative language gives support to the specification of bidirectional transformation, which requires information about the direction while being executed. A transformation can be defined as a set of relations between the target and source metamodel. This transformation is used to check for the consistency of these two models. QVT relational, with the implicit help of trace models, supports complex pattern matching by using OCL. **QVT Core** is a low-level and simple transformation language that is classified under the declarative model and can become a foundation stone for the QVT relational language. QVT Core supports pattern matching in line with a pack of variables. The trace models should be explicitly defined. **QVT Operational** is a transformation language that leads to an expansion of the QVT relational language with the constructs of the imperative model. It has been recognised that these transformations use the implicit trace models, and are unidirectional. ModelMorf and SmartQVT are the model transformation engines, which are based on the QVT standard (Biehl, 2010).

ATLAS Transformation Language (ATL): ATL is a hybrid language that transforms on a model-to-model basis and supports both types of construct, which can be either imperative or declarative. The declarative style gives a simple and clean interpretation for simple mapping and hence is mostly preferred over the imperative constructs, which are provided for handling challenging and complicated mappings. The ATL transformation program is a collection of those rules, which are used to develop and set the elements of the target models. ATL is incorporated into the Eclipse development environment for the purpose of handling EMF-based models, along with the UML profiles. ATL does not provide support to incremental model transformation; therefore, it is essential to read a complete source model and create a complete target model. Moreover, it has been determined that the target model is not capable of

preserving manual changes. ATL is supportive of the in-place transformation mode, which is also known as the refining mode; however, it has a drawback in that it cannot be used with various constructs, like lazy rules (Biehl, 2010).

Kermeta: Kermeta is an imperative programming and modelling language, which is used for general purposes, and it can also be helpful in performing the process of transformations. It offers meta-modelling based on EMF, along with constraints and checks. In addition, it is necessary to load and store the models and metamodels explicitly, and to instantiate the target element explicitly in the target model, and this procedure requires more code. Rule scheduling and rule application control must be specified explicitly by the user. Exceptional handling, reflection and aspect-orientation are supported by Kermeta; however, it does not prove to be multi-directional or traceable. It has been analysed that the complete target and source models are being read, created and executed, since incremental model transformation is not supported (Falleri et al., 2006).

XML Stylesheet Language Transformations (XSLT): XSLT a functional language that performs transformation works to manipulate XML data, and the rules followed during the functioning of this language are explicit in nature. Such standards are purely unidirectional, and traceability is also not endorsed. It has been found that the transformation of this language is stateful, and in this regard, no support will be provided to incremental transformation. XSLT transformation descriptions are in the form of XML documents. At the initial stage, XSLT was developed for the purpose of converting XML documents into HTML; however, XSLT has been restricted to a simple transformation process (Bex et al., 2002).

3.2.6. Re-engineering and Reverse Engineering

Re-engineering refers to the process of redesigning a system created with old technologies to increase its maintainability. According to Arnold (1993), any activity conducted for re-using, maintaining or evolving software that enhances the software as a whole, is re-engineering. The need for re-engineering occurs when the quality of a system is degraded through its having been regularly changed, yet the change is

required. Similarly, re-engineering a particular system whose quality is low, but business value is high, is less risky and more economical than replacing the whole system.

There are two phases involved in the process of re-engineering: reverse engineering, which is an understanding phase, and forward engineering, which is a transformation phase. Chikofsky and Cross (1990) provided a widely-accepted definition of reverse engineering, which states that reverse engineering is a process that examines and analyse a subject system to determine its components and their relationship with each other, in order to achieve an abstract level of architecture. The following diagram (Figure 3-4) helps to understand the terminology used in a software lifecycle.

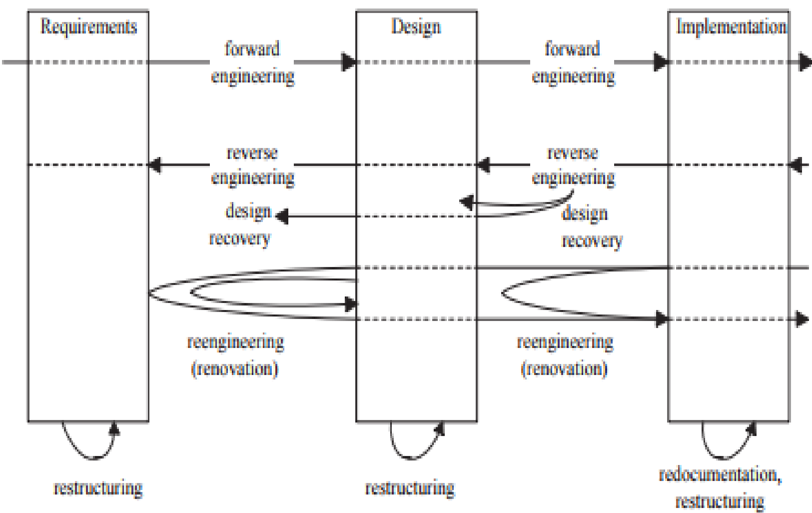


Figure 3-4: Reverse and forward engineering (Chikofsky and Cross, 1990).

The main aim of reverse engineering is the extraction of design artefacts and then the construction of abstractions that are not as dependent on their implementation. It is crucial to focus on gaining an understanding of the system, as system documentation is often not available and source code files may be the only resource for extracting information related to the system (Chikofsky and Cross, 1990).

Forward engineering is concerned with moving towards practical implementations from the perception of high-level requirements and models. Re-engineering is thus a

combination of reverse engineering and forward engineering. The driving force of re-engineering in forward engineering is the necessity of implementing new requirements, and in reverse engineering, re-engineering is conducted for an existing system model, by implementing the changes needed for new requirements, and transmitting the changes that are implemented with the techniques of forwarding engineering.

3.2.7. Software Architecture Recovery

Software engineering has several disciplinary branches, among which is software architecture recovery (SAR), which applies the concept of reverse engineering to existing systems. SAR is the process of retrieving the major components of a piece of software and/or its subsystems, and also dependencies between software and subsystems (Gall et al., 1996). The architecture of a software system consists of compositions of components, the interaction between these components. Retrieving and recovering crucial architectural information from an existing system is the prime objective of SAR.

Various terms are used in the literature to refer to the term SAR, such as 'Reverse Architecting, Architecture Extraction, Mining, Recovery, or Discovery' (Ducasse and Pollet, 2009). The term discovery is specifically used for the top-down process, whereas recovery refers to a process that follows the bottom-up approach, which the present research follows. Ducasse and Pollet (2009) conducted surveys on various methods of SAR and categorised SAR methods on the basis of their goals, techniques used, variety of inputs, types of processes and formats of desired outputs, as depicted in Figure 3-5. In respect to SAR goals, software architecture has various purposes. In the views of Garlan (2000), and the development of these by Ducasse and Pollet (2009), there are six prominent goals of software development fulfilled by software architecture, as follows:

Understanding and re-documentation: A software system is depicted at an optimum level of abstraction by architectural views, which is necessary to explain the design of the software. Architectural views help in taking decisions

considering the constraints of design, principal of designs, quality attributes. For example, this goal is illustrated by the software bookshelf introduced by Finnigan et al. (1997).

Re-use: Architectural views highlight components, frameworks and patterns which can be re-used. SAR is also used for architectural environments that are service-oriented, to identify the components which can be converted from the existing system into new services (O'Brien et al., 2005).

Conformance: Using conceptual architecture is risky in evolving a software application because it is often inaccurate compared with concrete architecture. SAR is a means of checking conformance between concrete and conceptual architecture. Murphy et al. (2001) pointed out that the reflexion model bridges the gap between the system's source code and the system's architecture. Reverse engineers can use SAR to check the conformance of architecture reconstructed against a rule like Symphony (van Deursen et al., 2004).

Co-evolution: Abstraction, architecture and implementation have two levels in their evolution. They evolve at different speeds, so the problem of synchronisation is faced by the software; in order to avoid architectural drift, synchronisation is essential. A method of repairing evolution anomalies between concrete and conceptual architectures was proposed by Tran and Holt (1999), which involved altering either the source code or the conceptual architecture.

Analysis: Quality attribute analysis and dependence analysis are performed on the basis of high-level abstraction of architectural views. Architectural analysis methods like ATAM (Kazman et al., 1998) are supported by SAR environments.

Maintenance and evolution: SAR is usually considered the first step towards maintenance and evolution of software. For example, focus approach (Ding and Medvidovic, 2001) helps to understand the architecture of a software system as well as the evolution of the application.

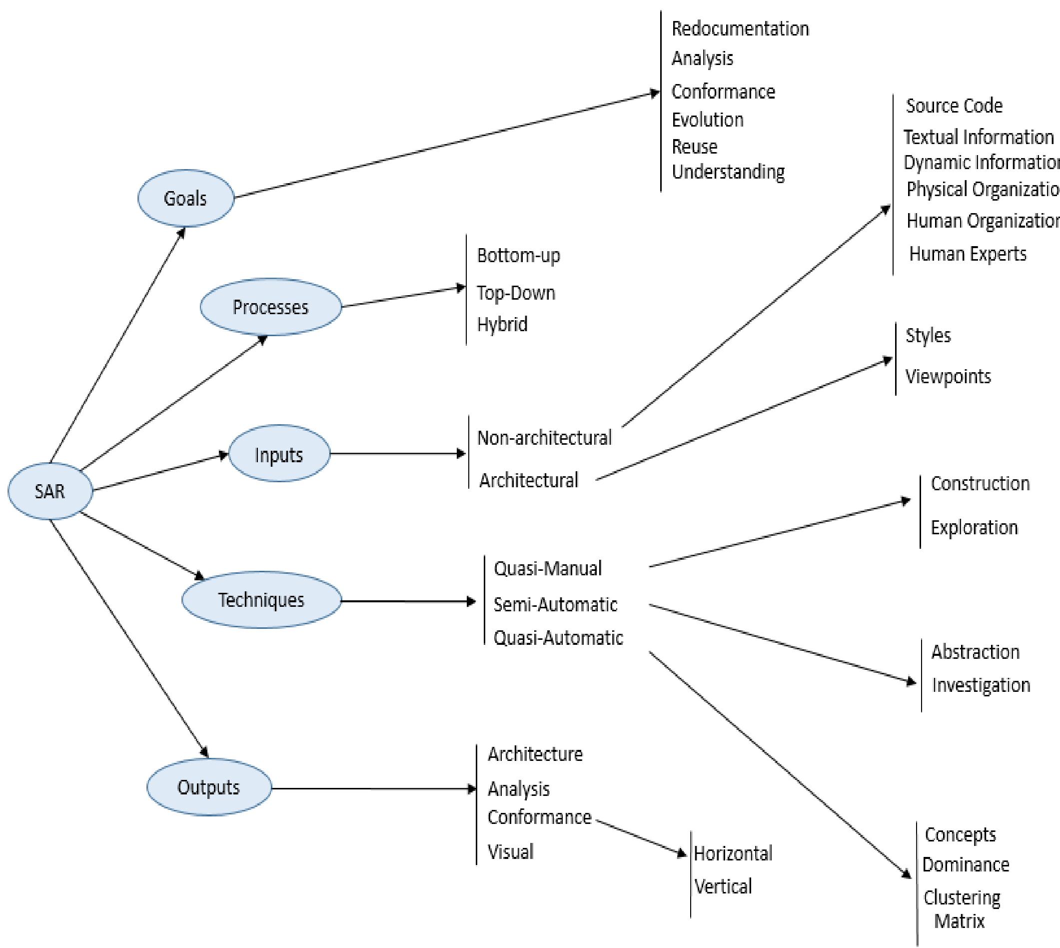


Figure 3-5: A process for SAR (Ducasse and Pollet, 2009, p. 576).

The procedures for software architecture recovery have been classified by Ducasse and Pollet (2009) into three classes, bottom-up, top-down and hybrid approaches.

Bottom-up processes: Often referred to as recovery processes (Figure 3-6), these begin with low-level information such as source codes, documentation and other structured information related to the software. While most include structural information, some include non-structural information such as file paths and ownership; some also include textual information such as the text content of the documentation and comments. The abstraction level of the information is then increasingly raised to achieve a high-level view of the software. A classic example of bottom-up processes is demonstrated by the Dali tool (Guo et al., 1999), an automated and interactive architecture extraction system.

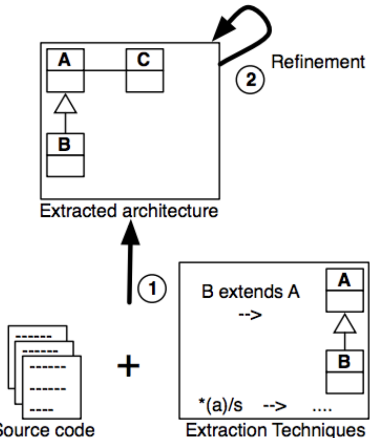


Figure 3-6: A bottom-up process (Ducasse and Pollet, 2009).

Top-down processes: Also known as architecture discovery processes (Figure 3-7), these aim to discover components of the source code that correspond to high-level knowledge description of the architecture, for instance architectural style and its requirements. A top-down scenario is shown in the reflection model (Murphy et al., 2001). Firstly, the user plans a high-level conceptual view, and after that the user starts the process of mapping the concrete view, and the conceptual view of the source code.

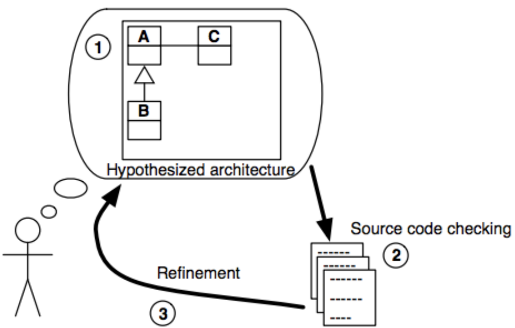


Figure 3-7: A top-down process (Ducasse and Pollet, 2009).

Hybrid processes: As the name entails, hybrid processes combine elements from both bottom-up and top-down processes, as shown in Figure 3-8, and are often used to prevent architectural erosion. The aim of hybrid systems is to use bottom-up techniques for hypothesis recognition in order to aid exploration of architectural hypotheses for top-down processes (Pashov and Riebisch, 2004). Several techniques are used to abstract low-level knowledge, which is then confronted against refined

high-level views. The conceptual and the concrete architectures are then reconciled. For example, the pattern-based recovery system in Sartipi (2003) has a two-phase architecture reconstruction process. In the first phase, the source code is parsed into a graph and divided into cohesive sub-graphs using data mining algorithms, which results in a more abstract representation of the code. The graph can then be queried using an architecture query language (AQL) to find a sub-graph that matches the query using clustering and graph matching techniques in a top-down fashion.

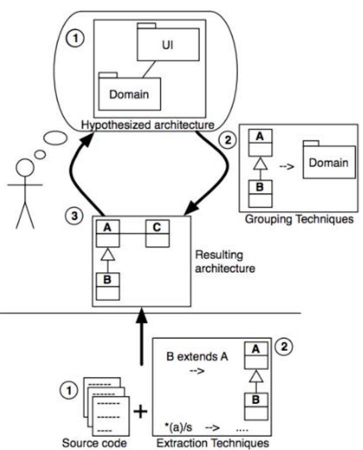


Figure 3-8: A hybrid process (Ducasse and Pollet, 2009).

Approaches undertaken for SAR in regard to inputs are mostly based on source code information or the expertise of humans. Sometimes, however, other means and information sources are utilised, like dynamic and historical information. Different architecture reconstruction techniques have been looked at by the research community; the techniques below were classified by Ducasse and Pollet (2009) based on their level of automation.

- Quasi-manual: using this technique, the reverse engineer has to manually find the architectural elements, and findings of reverse engineering are supported by different studies. Examples of this technique are Focus (Ding and Medvidovic, 2001) and Dali (Guo et al., 1999).
- Semi-automatic: using this technique, the reverse engineer has to manually instruct the tool regarding how to automatically flag any refinements or recover any abstractions. Examples of this technique are Armin (Kazman et al., 2002) and SARTool (Feijs et al., 1998).

- Quasi-automatic: using this technique, the reverse engineer has no control over the tool and mainly steers the iterative recovery process. Examples of this technique are Alborze (Sartipi, 2003) and Bunch (Mitchell and Mancoridis, 2006).

Most of the approaches' outputs aim to provide architectural views or visualisations. However, tools such as Rigi (Müller et al., 1993) use graph representations of the software to visualise the outputs/results.

In order to understand existing software systems, it is necessary to understand both dynamic and static analysis, which are utilised to provide information about software artefacts and their associations.

Static analysis describes static information; it shows the software structure as it is written in the source code, e.g. classes and components. Mendonça & Kramer (2001) provide an approach to static reverse engineering known as X-ray, which can be useful in recovering the architectural runtime information from distributed software artefacts. The driver behind the development of X-ray was the supposition that a lot of information on the possible runtime architecture for a distributed system is accessible from its implementation. In X-ray, this is achieved or accessed by the utilisation of three corresponding patterns based on static analysis techniques: module classification, syntactic pattern matching and structural reachability analysis. Abi-Antoun and Aldrich (2008) developed an additional approach to static analysis in order to extract the runtime architectures from object-oriented programs (OOPs) written in existing languages. In their method of architectural recovery, a developer basically utilises annotations to recover the design from code.

Dynamic analysis describes dynamic information and shows behaviour during runtime, e.g. event trace information. SCED (Koskimies et al., 1998) is a prototype tool that was built in order to support the dynamic modelling of object-oriented applications. This tool was designed to be utilised in the design and analysis phases of the development process of object-oriented software. In this research, this tool is used to reverse engineer the behaviour of Java applications on runtime. The primary user interaction in the SCED tool contains several independent editors, including a state

diagram editor and a scenario diagram editor. In SCED, a scenario diagram is a variation of a sequence diagram in Unified Modelling Language (UML). A notation of the SCED state diagram can be exemplified as a simplified UML statechart diagram notation.

Combined static and dynamic analysis: various attempts have been made to merge dynamic and static analysis. Systä (2000) presents a reverse engineering environment which is known as Shimba, in order to reverse engineer Java software, which combines dynamic and static analysis in an effort to understand the Java software system's behaviour. Static analysis is utilised to select components that need to be analysed later during dynamic analysis. Systä's approach is based on the fact that the software engineer is not required to track the entire system if only a particular part needs to be examined. The dynamic examining is done via the process of generating the event trace information by running the software under the JDK debugger, also known as JDebugger.

Sartipi et al. (2006) also provide helpful architectural information through both dynamic and static analyses. Sartipi et al. (2006) developed an architecture recovery project named Alborz, in order to recover components that are highly cohesive. In the case of static analysis, multiple components are extracted by the tool along with their interactions, where a component is a number of system functions or a number of system files, and the interactions are defined according to or based on the terms of the export and import of entities of the software at the functional level. In dynamic analysis, frequent patterns in execution process traces are utilised to map the individual software's features onto the components of the software.

Generally speaking, the process of recovery is based on the extract-abstract-present paradigm, as shown in Figure 3-9. I can observe from the literature that the recovery phase in most approaches involves the process of an *extraction phase*, which involves the architectural information that is extracted with the help of several artefacts that are identified as the source code of the system, related documentation, history or knowledge regarding the architecture, and storing the architectural information in a repository. It also involves *abstraction*, which helps to describe the operation of filtering and grouping the information in order to gather meaningful information. The

procedure of presentation also provides details related to organising information in such a way that it becomes familiar to the readers who are targeted, such as via graphical and textual representations.

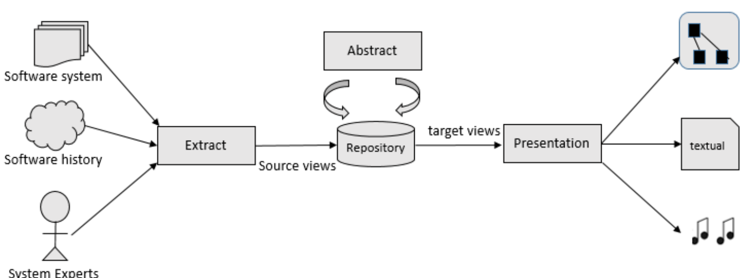


Figure 3-9: Extract-abstract-present paradigm.

3.2.8 Architecture Recovery Frameworks

Several frameworks adopted by researchers have been covered in the literature in the last decade for supporting software architecture recovery. Mendonça and Kramer (1996a) divided them into sub-frameworks, highlighting the advantages and disadvantages of each such as (filtering and clustering, compliance checking, analyser generator, and program understanding frameworks). More recently, machine learning and model-driven engineering frameworks have been used (El Beggar et al., 2013). A categorisation of architecture recovery frameworks is as follows (Mendonça and Kramer, 1996):

Filtering and clustering frameworks: A source model is extracted from the source code, and it is processed through a parser in a database. The filtering and clustering operations are performed based on low-coupling and high-cohesion properties to help identify the system components, as shown in Figure 3-10. Techniques that fall under this category are Rigi (Wong, 1998) and Arch (Schwanke, 1991).

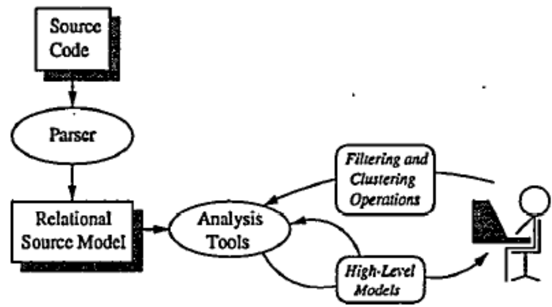


Figure 3-10: The filtering and clustering framework (Mendonça and Kramer, 1996).

Compliance checking frameworks: The process of extraction here follows the same steps as the filtering and clustering frameworks. It differs from the previous framework in the analysis phase, as the analyst defines the projected software's high-level model in a particular form (e.g. interconnection and modules, design pattern, inheritance hierarchy, architectural style), and after that, the tool examines the conformance level between source model and proposed model, as shown in Figure 3-11. An example of this method is the software reflexion model (Murphy et al., 1995; Buckley et al., 2013; Ali et al., 2012; Buckley et al., 2015).

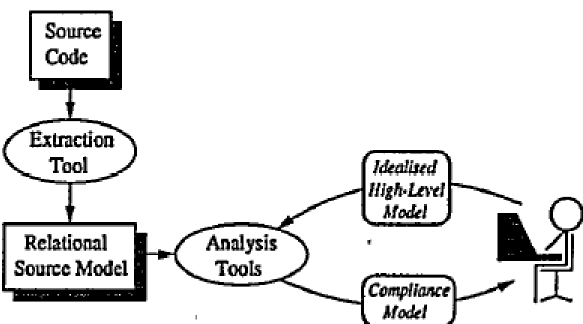


Figure 3-11: The compliance checking framework (Mendonça and Kramer, 1996).

Analyser generator frameworks: An abstract syntax tree is generated and stored by a parser. A query language is utilised to help generate queries. The purpose of those queries is to help analyse the specific properties of a software source model. Those abstract syntax trees are fully dependent on the query language. Refine (Burson et al., 1990) and Genoa (Devanbu, 1992) are examples of this environment.

Program understanding frameworks: In this method, the knowledge of an expert plays a key role in generating the abstract, high-level views of the system's functionality. The knowledge from the expert is taken and stored in a knowledge base, where the source model is placed and represented in an abstract syntax tree. Both the knowledge base (expert's knowledge) and the syntax tree (source model) support the recognition engine in searching for possible matches. The output of this process is a hierarchy of recognised patterns, which are the user-guided views of the system. DECODE is a tool in this category (Quilici and Chin, 1995).

Machine learning frameworks: Among the many techniques adopted by software architecture recovery is 'machine learning'. This technique mainly learns from previous background knowledge and data to help predict and extract future information. It is separated into two major categories: supervised and unsupervised. A supervised machine learning technique divides and classifies data into pre-defined classes, which are later used for training. In the unsupervised technique, data is not classified and is put in classes based on similarity properties (Bibi and Maqbool, 2011).

In machine learning, clustering is considered an unsupervised learning method, i.e. the aim is to learn an underlying pattern that describes the data. Clustering techniques can be adopted to find items that are related to one another in a set. This technique can organise data into groups or clusters, with the help of measuring or by using similarity distance. There are various similarity measures that are adopted under the clustering technique, such as Manhattan distance, Jaccard distance, Euclidean distance, etc. In the majority of cases, clustering requires the construction of a matrix, based on the data for input, and in the next step, algorithms are applied so as to identify the clusters.

Regarding clustering algorithms, two main categories are defined, namely partitional algorithms and hierarchical algorithms. A survey followed by a detailed study are presented by Maqbool and Babri (2007a), explaining the way in which the techniques of hierarchical clustering are implemented for architecture recovery, comparing different hierarchical clustering algorithms and measures, results of research, and

trends and issues. Unlike hierarchical clustering, partitional clustering requires a pre-set number of clusters as an input.

Supervised learning, on the other hand, learns an underlying pattern, or a function, that maps the data to a desired target output or labels, e.g. regression and classification. In Maqbool and Babri (2007b), a Naive Bayes algorithm was applied in software architecture recovery to deal with incomplete or missing documentation. The classifier was trained to classify new software modules into the appropriate subsystems. Bibi and Maqbool (2011) explored the use of supervised learning techniques in specific areas, such as architectural documentation maintenance management, where the researcher applied Bayesian and k-Nearest-Neighbour classifiers.

Model-driven Engineering frameworks: A developing approach in software development is model-driven engineering (MDE) (Schmidt, 2006). The primary concern of MDE is with the reduction of gaps between software implementation and problem domains using systematic transformation between the problem-level concepts and software implementation. MDE is a promising approach which centres around the theory "Everything is a model"(Pires et al., 2018). Models are used to bridge the gaps, and they define the complex systems at multiple abstraction levels through a variety of viewpoints.

In this context, the proposed technique of software architecture recovery can be initially formalised and categorised as a model-driven engineering framework. Contributions are made by this approach to the development of specific applications that are service-based, and they are also appropriate in this case. MDE has started to be recognised in the research community for addressing reverse engineering problems in the last few years (Raibulet et al., 2017). MDE approach supported the separation of concerns as models can be reusable and independent of their graphical notation. Also, an architectural model can be manipulated in other contexts and transformed into other forms. MDE is also supported with languages and plugins that aid the semi-automatic generation and manipulation of models, for example MDA transformation language. This allows the reusability, checking and automation of mapping rules and keeps the traceability between codes and models.

3.3. Related work

In this section, the primary limitations and drawbacks identified in relation to varied aspects of reverse engineering disciplines are discussed. The selected aspects relate to the present work, and the approach outlined in this section is from the architecture recovery perspective. In the first section, I present state-of-the-art research on generalised reverse engineering approaches based on the MDE paradigm, using software static or dynamic analysis. In the second section, I present work related to architecture recovery of microservice architecture. The last section presents a comparison and discussion of related approaches.

3.3.1 Overview of Model-Driven Architecture Recovery Approaches

The aim of this section is to provide a brief glimpse of the current state of generic bottom-up model-driven reverse engineering approaches, which are centred on the core concepts of the model-driven paradigm. The main conceptions considered in the studied approaches are presented in Figure 3-12.

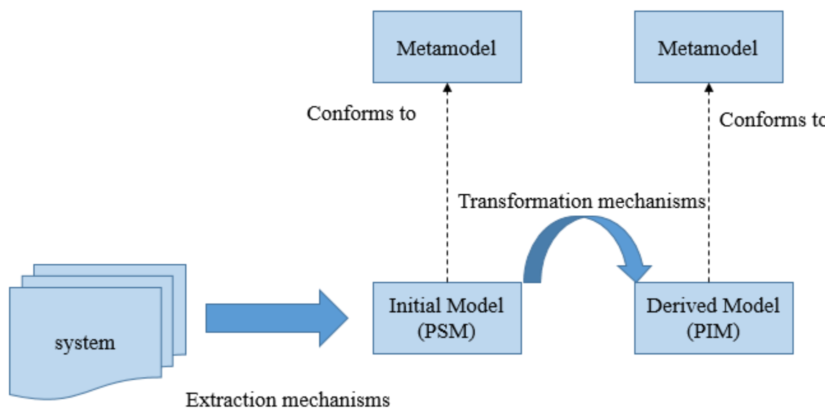


Figure 3-12: The main conception considered in the approaches.

First the extraction mechanisms implement to systems to generate the PSM as in Figure 3-12. With the concept of model transformation, PSM models are abstracted and altered, with the aim of generating the PIM, the targeted model. The transformation of models includes the computation, querying, navigation and

construction of further models (OMG, 2014). This study focuses on the following research questions are presented in Table 3-1:

Table 3-1: Research questions guiding the study.

Q.1: What types of artefact analysis are applied to generate the PSM*?*
Q.2: What are the source artefact extraction mechanisms used to obtain the initial model from the analysed systems?
Q.3: What kind of metamodels can be used by different types of model-driven architecture recovery approaches?
Q.4: What kind of tools can be used for the implementation of different types of model-driven architecture recovery approaches? Can the approaches provide new tools every time for solving different problems or are they using the existing tools?
Q.5: To what extent can automation be applied to the transformation in the specific context of model-driven architecture recovery approaches?
Q.6: What are the mechanisms through which the transformation can be applied to the model in order to have different abstract models?

The following presents the objectives of each of the approaches, the metamodel employed in each approach and the stages of reverse engineering associated with each approach. Author names are used for approaches that have no name.

MoDisco

Brunelière et al. (2014) propose MoDisco, i.e. well recognised as the model-driven reverse engineering approach. The core objective of MoDisco involves assistance in tasks pertaining to legacy systems, including documentation, understanding, quality assurance and modernisation. The architectural design of MoDisco comprises three main layers, namely infrastructure layer, technology layer and use-case layer. In this model, the first layers give general artefacts/components that are independent from any particular legacy technology. Another layer comprises specific components for single legacy technology. The final layer focuses on offering the integration and re-utilisation. For example, elements of two layers can be integrated, and then get reused by customisation for some distinct situations or scenarios.

The MoDisco approach involves two different steps; the first step is related to 'model discovery', in which the identification of the PSM is carried out, and this represents the system's source code. This model is obtained through specific software

components known as discoverers, and conforms to a given metamodel. For instance, metamodels associated with XML, Java and JSP are provided by MoDisco. The second step is 'model understanding', in which an in-depth analysis of the identified model (from the first step) is carried out. The objective of this step is to have clear and effective model transformations, through which OMG ADM standard metamodels are conformed. Some of these metamodels are the Generic Abstract Syntax Tree Metamodel (GASTM), the Knowledge Discovery Metamodel (KDM) and the Software Measurement Metamodel (SMM).

Figure 3-13 shows some of the specific steps performed within model transformations, in which the model is checked and refined through different steps until the desired model is derived. These steps are:

- Navigation through the initial models for the exploration of the system.
- Querying of the model to identify the required information.
- Model computation using the identified information.
- Representation retrieval in the derived models (model building).

Implementation of MoDisco is done in the form of the Eclipse open-source project. MoDisco does not aim at managing particular software such as microservice architecture; instead, it seeks to offer generic components to recover the legacy artefact. Although MoDisco is generic and includes various PSM metamodels, such as JSP, XML and Java, they do not support a 'platform' which supports higher-level programming idioms such as load balancing and support for architectural patterns. At artefact-level, MoDisco offers the potential to retrieve essential data from artefacts of software as well as highlights these artefacts along with the interrelations of them. However, the discover component that generates the PSM in MoDisco is not relying on multiple source artefact types nor integrating multiple source artefacts in the same transformation. At the architecture level, MoDisco defines a formal definition of models or diagrams, the view of architectures and automated identification of such architectural views; however, dependency recovery is limited to internal dependencies within one system, and this framework does not cover external dependencies at the system level (from system to system).

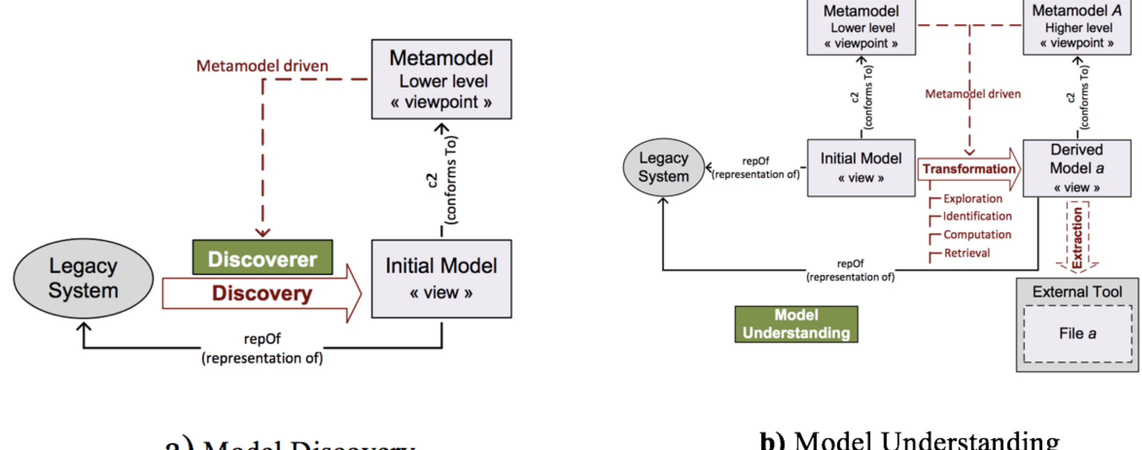


Figure 3-13: General principles of a) model discovery and b) model understanding (Brunelière et al. 2014).

Cosentino et al.

Cosentino et al. (2012) present an approach that aims to recover business rules from source code in Java by separating the segments of code that is related to the business processes. In this study, the reverse engineering based model-driven framework is exploited. Figure 3-14 shows various steps that are used in this approach, such as model discovery, variable classification, business rule identification and business rule representation as described in the following. These steps follow a chain of model-to-model transformation, and ATL is used to implement these transformations.

- **Model discovery** is enforced with the use of MoDisco (Brunelière et al., 2014) in the framework, and model discovery is added to the source code as an input in Java applications and when generating Java models; this Java model is referred to as platform-specific model.

- **Variable classification** recognises the variables provided by a domain together with their containing classes. A PSM is provided as the input for this operation, as well as obtained outcome reflects the modelling that directs every domain class of domain along with the concerned internal variables. The key agenda of

this operation is to determine variables that reflect concepts of the business domain as well as offers suggestions regarding the rules of business.

- **Business rule identification** is used in providing artefacts which represent business rules based on the program slicing technique (Tip, 1995). A PSM and domain variable model are taken as inputs. Two models are derived from this particular operation: one model comprises an internal representation of the business rules while the other model is based on a global domain model that conforms to the business rule metamodel, which comprises classes, attributes and methods.

- **Business rule representation** uses artefacts (text, graphs, etc.) that are understandable to humans in representing the extracted business rules.

This approach currently considers only Java software, as well as developing the Java PSM. The PSM is not applicable to the extraction of crucial information (such as classes, libraries, annotations and methods) from the various artefacts. The discovered models do not really describe a PIM since their works do not perform any distinction between platform-independent and platform-dependent technology concepts.

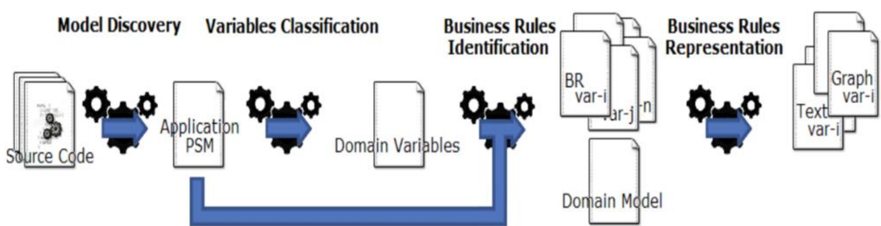


Figure 3-14: Overall approach (Cosentino et al., 2012).

MARBLE

Perez-Castillo et al. introduced the semi-automated approach to recover the business processes from Legacy Systems (Pérez-Castillo, De Guzmán, et al., 2011; Pérez-Castillo, Fernández-Ropero, et al., 2011). Their recovery procedure is based on their framework called MARBLE (Modernisation Approach for Recovering Business

processes from Legacy systems). This method is employed as an Eclipse plug-in. MARBLE supports the standard metamodel, Knowledge Discovery Metamodel (KDM), which is proposed by the Architecture-Driven Modernization (ADM) for retrieving business processes from a legacy system. With the use of MARBLE, communication with experts in between the extraction process is enhanced, and due to this, this method is also called a semi-automated approach.

In MARBLE, there are four clear abstraction levels, which show three different transformations as shown in Figure 3-15. MARBLE's initial level reflects and determines potential information system of legacy within the actual world. The first model is constructed by the legacy system, in which the analysis of the source is performed in a static and dynamic model. The second level includes various models, such as one model for each different software artefact, i.e. user interface, source code and database. At this level, different reverse engineering techniques, such as program slicing, log file generation within the static or dynamic evaluation, are mainly utilised for obtaining data from the software for forming a PSM, that conforms to metamodels e.g. a Java metamodel, SQL metamodel, etc. At the third level, a PIM is found, which is a cohesive representation of all the PSMs, and is generated in this approach by QVT relation. PIM is defined based on the KDM metamodel. The fourth level of MARBLE represents a CIM (Computational-Independent Model) which identifies the different business process models, that conform to business process modelling and notation (BPMN).

MARBLE has been applied to six case studies to recover business processes from systems. The use of case studies helped in the improvement and refinement of the MARBLE tool and technique. The efficiency and effectiveness of MARBLE are measured in these case studies. These measures are computed in regard to retrieved business process elements. Measurement of effectiveness is also conducted with the help of recall and precision. In precision, the correctness of the recovered business process is examined, while in recall, the completeness of the recovered business process is examined. Efficiency is calculated on the basis of the time required for the recovery of relevant information. The results of these case studies varied. Values for precision and recall varied between systems, though recall tended to be higher than

precision. Thus MARBLE retrieves a high number of business activities, but some of them may be erroneous.

MARBLE presents different abstraction levels and different models, such as PSM, PIM and CIM. Therefore, the last transformation of the fourth layer which represents the CIM should be maintained by the manual intervention of experts of business in order to refine the business processes. The fourth layer is beyond the scope of this thesis.

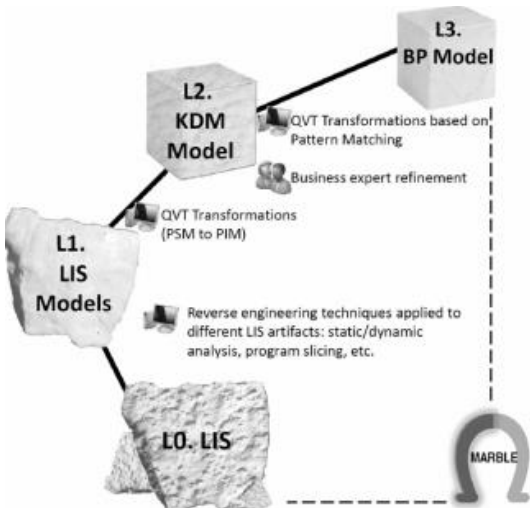


Figure 3-15: MARBLE framework (Pérez-Castillo, De Guzmán, et al., 2011).

El Beggar et al.

An approach has been suggested by El Beggar et al. (El Beggar et al., 2013) for the reverse engineering process for recovering objects from COBOL legacy systems. There are three steps overall that are considered for the identification of objects. Regarding model-driven architecture recovery solutions, El Beggar et al. suggest that the initial step is directed towards analysing and examining the source code required for the development of the PSMs that conform to the COBOL file description metamodel, as depicted by Figure 3-16. After performing this task, the obtained PSMs are integrated in a collective manner for the generation of the common model. This unified model is named as the Merge Model of File Descriptors, abbreviate referred to as (MMFD).

In the last step, a common model is transferred from PSM: MMFD to the PIM: Domain Class Diagram (DCD), which is, of course, a form of domain class diagram. The study mentions that for the transformation of the model they define mapping rules written in a natural language, as depicted in Table 3-2, and then they implement these via an ATL-based language.

Furthermore, they critically compare and evaluate their model-driven architecture recovery approach with a clustering approach in order to reveal which approach is more accurate. For this purpose, three specific evaluation metrics, recall, precision and F-measure, were taken into consideration. These metrics are used to obtain information regarding the degree of closeness of approaches of the correct extracted classes to the eventual classes produced by the human experts, which can be termed the expected classes. As per the results of the comparison, it is revealed that model-driven architecture recovery approaches are more appropriate in comparison to clustering approaches, due to the existence of high values of precision, recall and F-measure in comparison to those in the clustering approach. The main limitation of this approach and from a reusability point of view, is that it can only be utilised for a specific system such as COBOL legacy system.

Table 3-2: The main mapping rules for transformation (written in a natural language) (El Beggar et al., 2013).

Rule	From	TO	Mapping Description
1	Record	Class	Each record in the MMFD model gives place to one class.
2	Simple field	Attribute	Field is mapped to an attribute its type is transformed into a DataType as follows : Picture 9 or S9 → int Picture X or A→string Picture 9V9 or S9V9→float
3	Group field	Attribute with an equivalent type	Group field will be transformed to an attribute but its type and size will be deducted from types corresponding to the sub fields existing in the Group.
4	Record key	Attribute *isId=true, isUnique=true,*	The record's key becomes an attribute which features *isId* and *isUnique* equal to true.
5	Array	Class with ass multiplicities : lower=1,upper =1	An Array Field becomes in DCD a new class associated with the Class corresponding to the record that contains the array

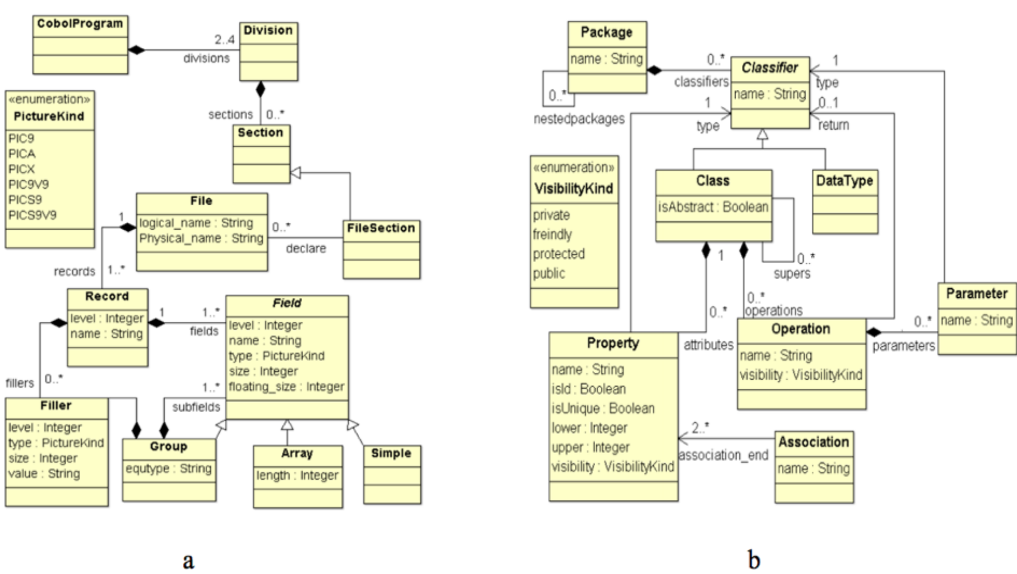


Figure 3-16: a) The PSM metamodel. b) The DCD metamodel (El Beggar et al., 2013).

Fleurey et al.

A study by Fleurey et al. (2007) proposed a model-driven reverse engineering approach that is based on the semi-automatic and round-trip model, which is assistive towards in the process of migration of industrial software. The clear intention behind this proposal raises from the need of complete re-development of the legacy application. The process of model-driven migration that was developed is illustrated in Figure 3-17.

There are four steps in this process. Within the depicted procedure, the first stage is focussed on attaining parsing of the existing source codes of the legacy application in an automatic form, which is helpful in creating abstract syntax tree for the information regarding legacy based on the source code, along with parsing, to obtain a code model that assists in the representation of the PSM that is in conform to the legacy programming language metamodel. Secondly, this process involves reverse engineering from the PSM (denoted as L-figure 3-17) to the PIM (pivot metamodel) via model transformations. The pivot metamodel (ANT), shows algorithms, static data

structure, Graphical User Interface (GUI), application navigator and widgets. Thirdly, the ANT model is specifically transformed into PSMs of the application (UML model). Finally, the generation of code is the last step in the process for developing new applications from the PSM.

A tool suite is provided by the authors for this process; it also named as Model-In-Action (MIA). It helps in the execution of round-trip engineering to transform and generate the code. These transformations are highlighted for the input and output metamodels. There are three elements for each rule, namely context, query and action, wherein the context is identified to signify a collection of the declared variables and the parameters. A query is defined as programming expression that is helpful in analysing the model elements that are processed with the help of rules, while an action can be considered as creation, deletion or modification of the model elements, which are performed in each model and returned as a query. This approach is valid on an existing case study based on the COBOL language. The case study describes the migration from the mainframe to the J2EE of a large banking system.

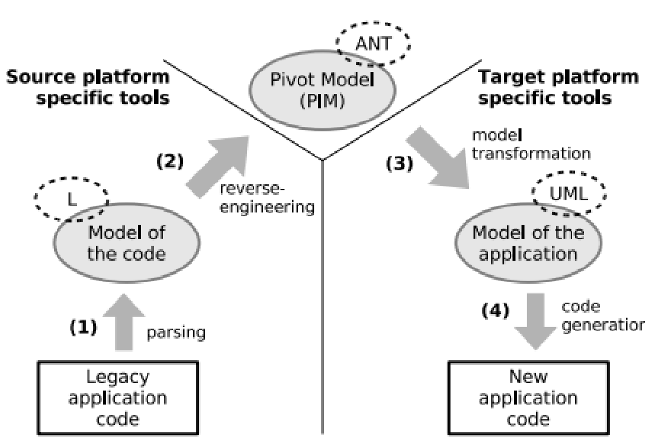


Figure 3-17: Fleurey et al.'s (2007) reverse engineering process.

Akkiraju et al.

For Service-Oriented Architecture (SOA), Akkiraju et al. (2012) clearly indicate a reverse engineering approach. This reverse engineering approach is directed to obtaining a model from PSMs in the process of developing PIMs. This model originates from an application developed on a specific platform. The authors'

approach applies forward engineering to the smooth translation of PIM to PSMs onto the target platform. Regarding metamodels, the creation of these platform metamodels can be handled manually or automatically by exemplar. Different vendor tools, including IBM's Rational Software Architect (RSA), provide different exemplar analysis tools. This metamodel is undertaken in a model generator module for the purpose of developing the PSM.

Model-driven transformations are generally used in the derivation of PIMs from PSMs. In this process, first of all, the authors developed the transformation rules manually. After this, IBM's RSA transformation authoring tool is used for codifying mapping rules. Table 3-3 presents a pictorial presentation of transformation mapping rules applied between a PSM, i.e. SAP NetWeaver composite application framework (CAF), and a PIM metamodel. In an SOA environment, the elements which will be valued at the platform-independent level are extracted by performing rationalisation and filtering mechanisms, for instance data structure, applications service, service operations and business objects. Then from the PSM, they extract different services such as security services, process services, infrastructure services and information services. This leads to the development of service dependency information and a service model, which together form the PIM.

Even though this work focuses on service-oriented architecture rather than the microservice architecture, there are many similarities in the MDE approach undertaken. Their approach focuses on service-level components rather than the class level of software design. However, their reverse engineering is difficult to achieve in microservice architecture. Microservice architecture has specific conceptual characteristics of architectural elements at different abstraction levels different than SOA (Rademacher et al., 2018).

Table 3-3: Transformation mappings between the PIM and PSM metamodels (Akkiraju et al., 2012).

Source: Platform Independent Model (PIM) artefacts	**Target:** SAP NetWeaver artefacts
Operation	Operation
Message	InputOperationMessage, FaultOperationMessage, OutputOperationMessage
ServiceComponent	Service
Entity	BusinessObject
FunctionalComponent	BusinessObject

3.3.2 Microservice Architecture Recovery Approaches

The following literature review investigates the architecture recovery approaches related to microservice architecture. The study of the topic of microservice architecture recovery is limited.

MicroART

Granchelli, Cardarelli and Francesco et al. (2017) propose a microservice architecture recovery approach called MicroART, based on MDE principles. MicroART has two main phases: architecture recovery and architecture refinement. The phase of architecture recovery focuses on the recovery of the system's initial architecture (physical model). The architecture refinement phase focuses mainly on refining the obtained architecture. The activities considered in the phase of architecture recovery are dynamic analysis to extract container information and communication logs, and static analysis to extract information from source artefacts (such as service descriptors, system name and developers).

Architecture recovery also includes activities that are performed in order to abstract information while utilising mapping techniques, as shown in Table 3-4, which map the information collected to the architectural concepts automatically according to (MicroART-DSL) metamodel. In the phase of architecture refinement, the authors

practiced the process of refinement in a semi-automatic manner in order to get the ultimate model of a microservice architecture, which is named the logical model.

Granchelli, Cardarelli, Francesco et al. (2017) propose a microservice architecture metamodel that is made up of seven meta-classes, as depicted in Figure 3-18, in which the root concept to be considered for the system to be designed is a *product*. *Microservices* represent the system, the main attributes of which are types (either functional or infrastructural) and the host (assigned IP address). The *interface* represents the endpoint of communication and attaches it to a particular microservice. The *link* represents the communication between them. The *team* consists of developers. *Developer*'s meta-class depicts the developer of the software that takes part in the system's development. A *cluster* represents a logical abstraction, which is used to group microservices of a specific type.

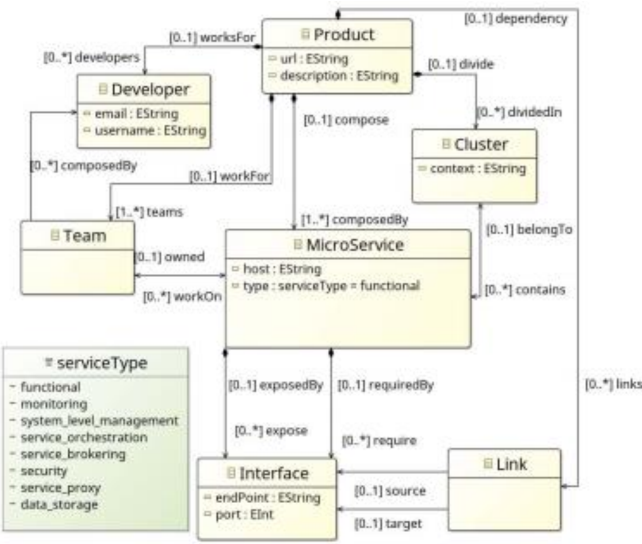


Figure 3-18: MicroART metamodel for microservice-based systems (Granchelli, Cardarelli, Francesco, et al., 2017).

The main function of MicroART is to gather information from the repository, and after taking all necessary information, it produces the system's architecture. After preparing the model, it is further polished by MicroART, and a refined model of architecture is produced by the implementation of service discovery resolution as a refinement process. The Eclipse Modeling Platform (EMF) was utilised in the development of the MicroART tool (Granchelli, Cardarelli, Di Francesco, et al., 2017). The MicroART

tool primarily consists of four components, GitHub Analyzer, Docker Analyzer, Log Analyzer and Model Log Analyzer.

GitHub Analyzer uses the web URL as an input of the source code of the system repository. It copies the repository and gathers information that is related to the name and description of the system and developer. The Docker Analyzer queries a runtime environment of Dockers and takes services' IP addresses and the interface of the network. The function of the Log Analyzer is to investigate the log files that are prepared by monitoring tools dynamically. It can also track the communication between different services. The main function of the Model Log Analyzer is to consider the architecture of the physical model as an input, and on this basis it identifies and discovers services and utilises the information to filter the log file properly. MicroART has been implemented to Acme Air, which is a microservice-based system for an airline website.

The main limitation of the MicroART approach is their metamodel, which is very simple and has few concepts and concerns to represent actual complex microservice architecture. MicroART's DSL metamodel does not define the asynchronous communication of the microservice. The generated models do not really describe a PSM though their works adopted MDE framework, the authors do not perform any distinction between platform-independent and platform-dependent concepts.

Table 3-4: Mapping of the extracted information and the MicroART-DSL(Granchelli, Cardarelli, Francesco, et al., 2017).

Extracted information			DSL concept
Analysis type	**Information**	**Concept**	**Metaclass**
Static analysis	GitHub metadata	System Name	Product (name)
		Developers	Developer (Name, Username, Email)
	Service Descriptor	Service name	Team (Name)
		Container name	Microservice (Name)
		Input ports	Interface (Port)
		Output ports	Interface (Port)
		Build path	Microservice (Build)
Dynamic analysis	Containers	Identifiers	Microservice
		IP address	Microservice (Host)
	Communication Logs	IP source address	Link
		IP target address	Link
		URL	Interface (EndPoint)

Microlyze

Another architecture recovery approach is Microlyze, by Kleehaus et al. (2018). This approach integrates the static and runtime data in order to recover IT infrastructures that are supported by microservice architecture. The layers like hardware, business, applications and their interrelationships are comprised in the recovery process. Microlyze correlates the reconstruct model, which is based on the infrastructures of the microservice, with the Enterprise Architecture (EA) model, which is utilised by several EA frameworks.

Thereafter, it is further divided into three abstraction layers. The first layer is recognised as the technological layer because it contains and defines the aspects related to technology, such as hardware, network and other physical components. The second is the application layer, which encompasses the software elements that are running on the first layer, such as the services and instances of service. The last layer is the business layer, which functions on top of the layers mentioned above. This layer characterises the facets that are associated with business, such as processes and activities of the business that are operated by microservices. With the help of analysis of monitoring data, the first two layers, the technology layer and application layer, are reconstructed automatically. However, additional knowledge of the domain as well as manual input is required to recover the business layer.

This approach was prototyped and evaluated in a microservice-based system called TUM Living Lab Connected Mobility (TUM LLCM). Microlyze does not adopt a model-driven recovery approach. Instead, it utilises a distributed tracing component that dynamically monitors simulated user requests. Metamodels and mapping rules are not dealt with in this approach.

Mayer and Weinreich

Mayer and Weinreich's (2018) study is mainly focused on the architecture extraction approach, so that it can continuously extract REST-based architecture from microservice software systems, in which the service communication is synchronous based on HTTP. This approach is a combination of static information (such as API descriptions and services) and dynamic information (communication relationships captured at runtime).

The architectural information that is mostly presented on the basis of that data model, given in Figure 3-19, consists of three main sections, infrastructure, service and interaction. The service part contains the service information, which is mostly obtained from the static analysis. The service element consists of the version, title and brief of the service; this represents a microservice. The contact element is connected to the services element that represents a person who is responsible for various services and domain nodes in the organisation. The method element explains the HTTP method. The parameter element is also represented by the method parameters. The possible outcomes to the invocations and methods are represented as the response element, which also includes descriptions of various responses. Parameter and response elements are also connected with the schema element, so that it can explain multiple types of response data and invocation parameters.

The information that is related to the infrastructure is also given on the right side of Figure 3-19. It is assumed that every service is operating in its own container. Region and host elements represent the physical infrastructure. The interaction section is used to demonstrate the microservices communication. Any runtime connection related to the service is shown by the request element linked to the server-side responses. The

data model was evaluated by conducting a combined interview and survey so that significant information use cases can be used for managing microservices.

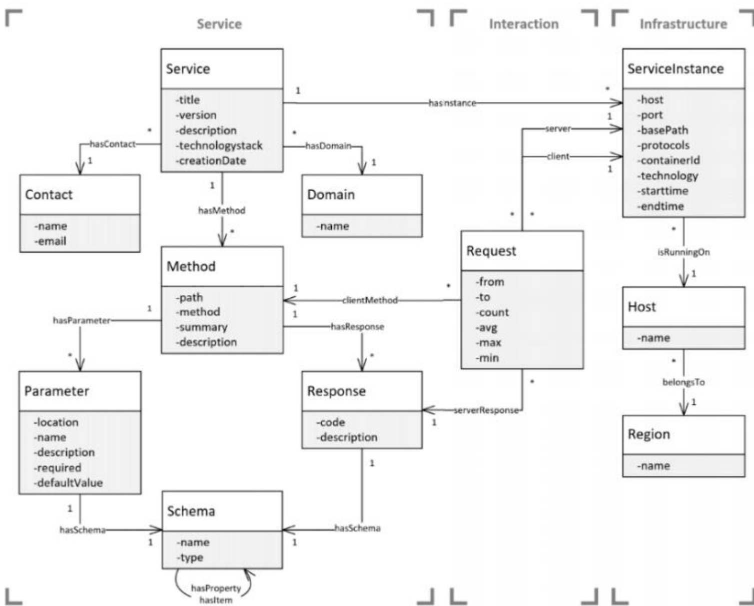


Figure 3-19: Data model (Mayer and Weinreich, 2018).

The approach of Mayer and Weinreich to the extraction of architecture reflects three stages, as shown in Figure 3-20. These stages are 1) data collection, 2) aggregation services and 3) management services. In this process, first of all, static information related to services and infrastructure is obtained and redirected towards central management services. After this, the extraction of service runtime information is frequently carried out and stored in service-specific log files. Eventually, the logged files request is extracted by taking the assistance of aggregation services.

This work uses dynamic analysis, i.e. monitoring of simulated requests at runtime, to recover synchronous REST-based communications in microservice architecture. The approach extracts static information, which starts after a service instance is created and deployed; such architectural information is related to API descriptions, developer and service. The main limitation in Mayer and Weinreich's approach is the restriction to REST-based and synchronous communication between services. They do not expressly represent any other communication types that exist in microservice architecture.

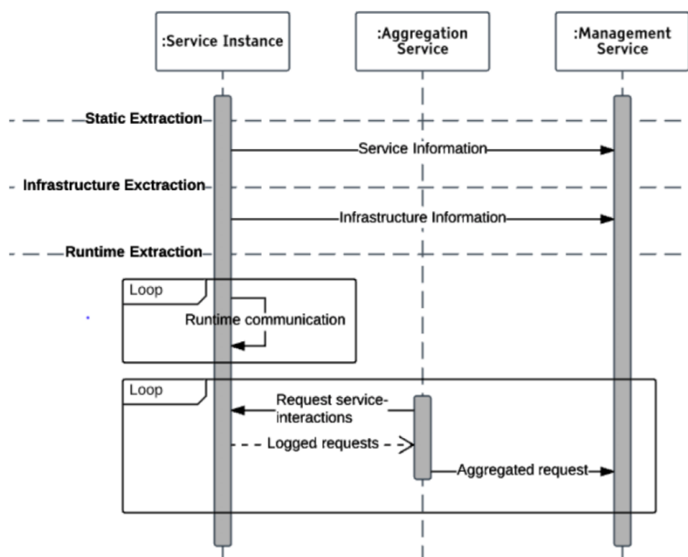


Figure 3-20: The architecture extraction process.

3.3.3 Model-Driven Approaches for Microservices

This section presents model-driven approaches for microservices that have not been used in architecture recovery. The studies discussed here (Düllmann and van Hoorn, 2017; Rademacher, Sorgalla, et al., 2019) present work related to metamodels and languages for microservice architecture, however, they are not oriented towards "architecture recovery". Düllmann and van Hoorn (2017) present a structure of a microservices environment from various viewpoints, such as microservice types, dependencies and deployment, focusing on the area of application performance monitoring as shown in Figure 3-21. However, their proposed metamodel does not consider asynchronous operation or asynchronous dependencies. The structure of business data offered by the services is also not covered. In addition, their metamodel concepts are not comprehensive, do not cover all architectural concepts that exist in the code and are missing some of the concepts that should be considered when recovering models.

Rademacher, Sorgalla, et al., (2019) present a metamodel for model-driven development of microservice architecture. Its basic concepts were deduced from ten existing approaches to SOA modelling. Their metamodel is structured into three distinct viewpoints. They comprise only those concepts relevant to domain-specific Data,

Service and microservice architecture Operation, as shown in Figure 3-22. However, being oriented towards implementation of microservice applications, the proposed metamodels need to be more concise and high-level if they are to be used for the reverse purpose, i.e. recovery of microservice architecture from implementation. The proposed service and operation metamodels do not define concepts for asynchronous data exchange. In addition, their metamodel lacks concepts related to infrastructure microservices.

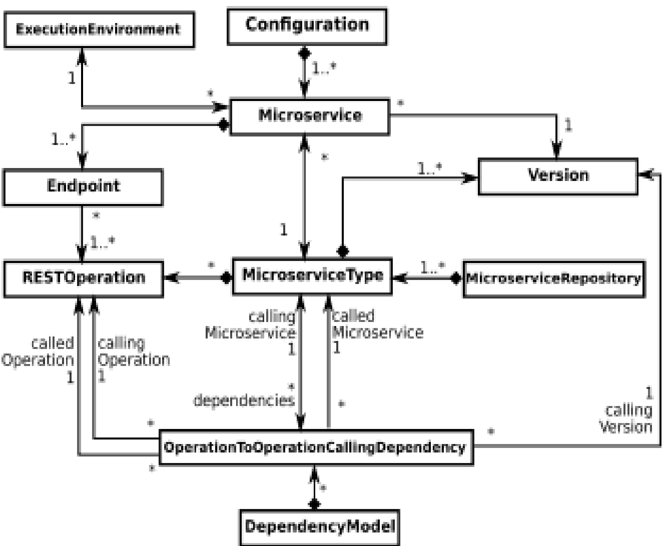
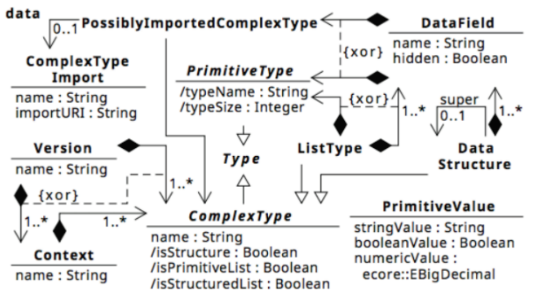
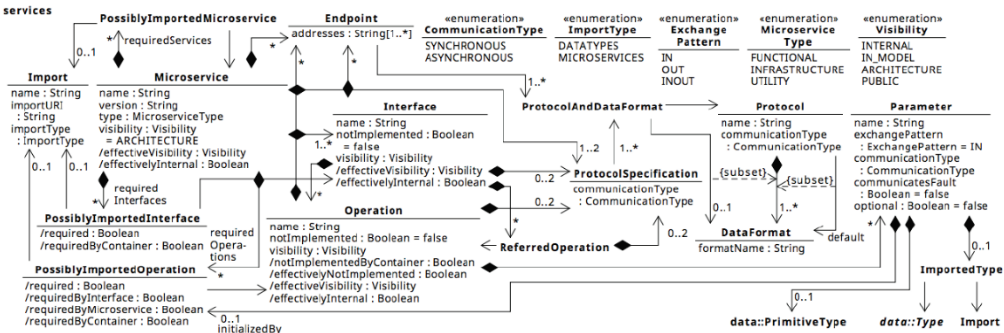


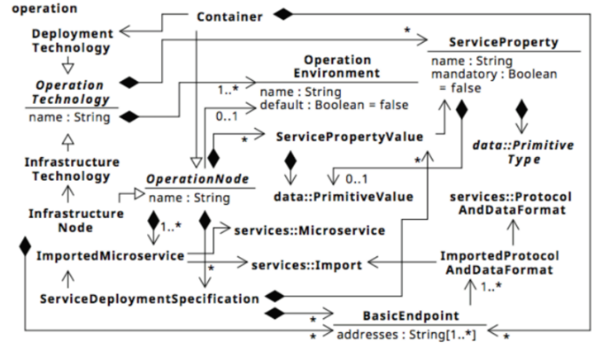
Figure 3-21: Metamodel proposed by Düllmann and van Hoorn (2017).



a) Metamodel of the domain data viewpoint.



b) Metamodel of the service modeling viewpoint.



c) Metamodel of the operation modeling viewpoint.

Figure 3-22: Metamodels a, b and c proposed by Rademacher, Sorgalla, et al., (2019).

Rademacher et al. (2019) present a metamodel of technology modelling language in the context of microservice architecture, implemented with Xcore, as presented in Figure 3-23. The motivation behind their work is to refactor monolithic architecture into microservice architecture. The metamodel they propose captures technological decisions related to microservice development and deployment with the aim of enabling the usage of different technologies in microservice architecture. Figure 3-23

expresses the metamodel concepts as ‘Deployment Technology, Data Format, Infrastructure Technology, Programming Language and Protocol’. In terms of model transformation, they use viewpoints and import mechanisms to reduce abstraction among platforms. The main drawback of this approach is that it introduces a level of complexity in reverse engineering activity, and from the point of view of the source code extraction process, static analysis is challenging in heterogeneous systems, since it essentially requires a suite of parsers to extract the architecture of the whole system.

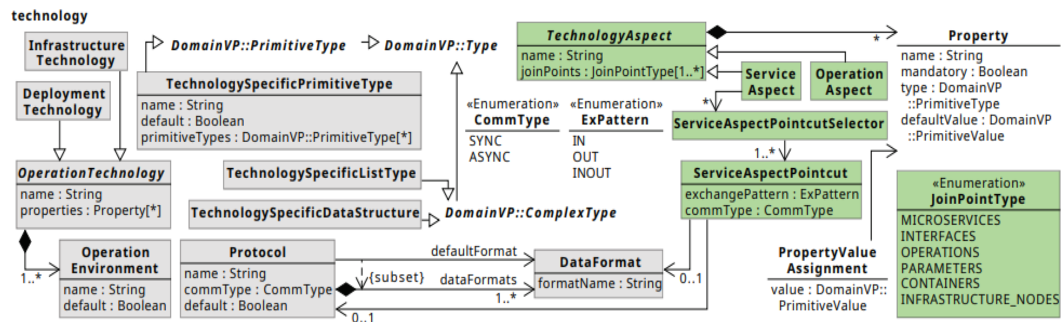


Figure 3-23: Technology Modeling Language defined by Rademacher et al. (2019)

3.3.4 Comparison of the Related Approaches

In order to provide an introductory comparison of the approaches presented above, nine features were considered to answer the questions formulated in Table 3-1: the objective of the approach; the models utilised (PIM and PSM); the extraction method for the generation of the initial PSM; types of system analysis performed on the source artefacts (static, dynamic or both); types of metamodels, which can be either a standard model or a new, innovative metamodel that the researcher may propose; mapping rules mechanism; the automation level – if the architecture recovery process can be partially (semi-automatic) or totally automated (automatic); transformation languages used; and tool support for implementation of the approaches.

It can be observed from Table 3-5, in the metamodel column, that the group of approaches can be divided into ad-hoc models and standardised models such as KDM, and modelling languages such as UML. For example, some approaches define new metamodels, such as those of Cosentino et al., El Beggar et al., Fleury et al.,

MicroART, and Mayer and Weinreich. The exceptions are the approaches of MoDisco and MARBLE, as they extend and re-use the standardised KDM metamodels.

It can be observed on the basis of the system analysis source code analysis column that there are nine approaches. Out of these nine, five approaches complement the static system analysis with the help of dynamic analysis (MoDisco, MARBLE, MicroART, Mayer and Weinrich, and Microlyze). For example, MoDisco's approach generates the legacy system dynamic view using the traces of execution, such as UML sequence diagrams. MARBLE's approach improves the PSM that is obtained statistically with the help of information obtained in log files, which, in turn, is achieved by executing the legacy system. The MicroART approach collects the IP address and network interface of every service, and creates the log files with the help of TCPdump by firing dynamic queries to the Docker runtime environment. The Microlyze approach does this by fetching the data from discovery services. The remaining four approaches make use of just the static system analysis (El Beggar et al., Cosentino et al., Akkiraju et al. and Fleury et al.). None of the discussed approaches use only dynamic analysis for their recovery systems.

It can be observed from the model transformation mechanism column that most of the approaches have associated mapping rules, e.g. El Beggar et al., MicroART, Fleurey et al., MARBLE and Fleurey et al. Two approaches are based on mechanisms such as ModelNavigation, ModelQuerying and ModelComputation (MoDisco) or the variable classification mechanism (Cosentino et al.). Nevertheless, it is important to note that each model transformation mechanism should be dedicated to the project and its specific technical space.

It can be determined from the automation level column that in the process of reverse engineering, partial automation needs human intervention, such as from software engineering experts, for the steps involved in executing the approach. On the other hand, total automation does not require any human intervention in the approach execution. Six approaches are partially automated, as they require human participation for the refinement and enrichment of the model by further information collection. MicroART is an example of one such approach. MicroART needs manual intervention from a software architect in order to resolve service discovery and create the

architecture's logical model. MARBLE's approach needs human intervention during the dynamic analysis of source artefacts. Akkiraju et al.'s approach needs human intervention to derive the metamodel of the platform. Microlyze also requires human intervention to obtain the domain knowledge, as well as the manual input, which is only available to the members of staff.

It can be observed in the tools support column that every one of the approaches incorporates tool support. Three approaches utilise existing tools for their approach implementation (El Beggar et al., Cosentino et al., Akkiraju et al.). Five approaches need to identify new tools for applying and implementing their approaches (MoDisco, MARBLE, Fleury et al., MicroART, Microlyze), while Mayer and Weinreich's approach does not use tools.

3.4. Research Gap

The literature review above confirms that, currently, there is much less research available for architecture recovery in microservices. Among the studies analysed, there are only three recovery mechanisms in the microservice context, namely MicroART, Microlyze, and Mayer and Weinreich, but none of these define a detailed process of MDE which places the metamodel and mapping rules in the core of the different phases of the recovery engineering, and they do not deal with modelling at the PIM and PSM levels, or with the transformation of PSM- and PIM-level models for microservice architecture. Thus, this thesis contributes to the field by providing definite arguments that separate the specific implementation and architecture views shown in the development of microservice applications, and in defining models related to the PSM and PIM levels and mapping rule transformation between them.

The major shortcomings of the current approaches are as follows. (1) Reverse engineering capabilities in regard to extracting a static structure for object-oriented systems are already available. There is a limitation that can be identified in the adoption of these tools. The analysis is conducted at the class level (e.g. interclass-level interaction) instead of the service level (e.g. remote calls between components

for distributed systems). (2) There is a lack of model transformation at different abstraction levels. (3) There is a lack of complete modelling of microservice dependencies, e.g. that specifically incorporate at most two different types of communication protocol, or synchronous and asynchronous communication models. (4) The tools are unified with a particular programming language and cannot really be used to handle microservice architectural concepts such as infrastructure components or lightweight API gateways, which enable easy interaction with external customers. Furthermore, the implementation of microservice logic also includes the technological artefacts that perform an essential function in execution and deployment, e.g. software frameworks and deployment descriptors. (5) Microservice modelling based on MDE is analysed and examined in the literature. Discussions regarding microservice solutions in MDE can be found in Rademacher et al. (2017), who state that few publications about model-driven approaches to microservices in general yet exist. In addition to this, none of them are in relation to the concept of architecture recovery (Ameller et al., 2015). The industries of SOA have proposed various approaches for modelling SOA, both informal and formal model-based standards. OASIS released a reference model in 2006, which hierarchically defined the components of SOA in an abstracted model form (Mackenzie et al., 2006; Kreger and Estefan, 2009), as defined by Open Group and OASIS. In contrast to SOA, neither the Object Management Group (OMG) nor OASIS have defined an approach for microservice architecture. This means that there is a gap in knowledge around microservice architecture concepts and their recovery.

Hence, by following the MDE approach, this research fills this gap in the area of microservices with MDE. A well-designed metamodel and well-developed and tested mapping rules for supporting the architecture recovery of microservice-based systems is proposed, developed and evaluated.

3.5. Summary

In this chapter, the background context relevant to the theme of the thesis was introduced, and studies that address generic model-driven architecture recovery were discussed, with a focus on bottom-up transformation mechanisms. Then architecture recovery methods for microservice architecture were extensively looked into, along with the criteria that have been addressed and the techniques that have been utilised. Model-driven approaches for microservices that have not been used in architecture recovery were presented, with a focus on the metamodel language utilised in such work. The chapter concluded with a discussion of the research gap that this thesis aims to fill.

Table 3-5: Comparison of the related approaches.

Methods studied	Objective of the reverse engineering	Model type		Source code extraction method	Source code analysis	Metamodel			Model transformation mechanism	Transformation languages used	Automation level	Tool support
		PSM level	PIM level			PSM level	Intermediate model	PIM level				
MoDisco (2014)	Modernization of legacy systems	√	√	Discoverers Component	Static, Dynamic	Java, JSP, XML metamodel	-	(KDM, SMM, GASTM) metamodel	ModelNavigation, ModelQuerying, ModelComputation and ModelBuilding	-	Automatic	MoDisco
Cosentino et al. (2012)	Recover business rules model from Java source code	√	-	MoDisco	Static	Java metamodel	Variable classification, Business Object model/ Vocabulary model metamodels	Business Rule metamodel	Variable Classification, Business Rule Identification, Business Rule Representation	ATL	Automatic	MoDisco
El Beggar et al. (2013)	Recover objects from COBOL legacy systems	√	√	-	Static	COBOL metamodel	Merge Model of File Descriptors (MMFD)	Domain Class Diagram (DCD) metamodel	Rule-based transformation	ATL	Automatic	-
Fleurey et al. (2007)	Model-Driven Migration Process	√	√	Parser	Static	Metamodel of the legacy language	AST, UML	ANT metamodel	Rule-based transformation	-	Semi-automatic	Model-InAction (MIA)

MARBLE (2011)	Recover Business processes from Legacy Systems	√	√	Parser (Java)	Static, Dynamic	Java, SQL metamodel	-	KDM metamodel	Rule-based transformation	QVT relation	Semi-automatic	MARBLE ™
Akkiraju et al. (2012)	Recover Service-Oriented Architecture	√	-	Model Generator Module	Static	Create a metamodel by exemplar	-	Derive metamodels	Rule-based transformation	-	Semi-automatic	IBM RSA tool
MicroART (2017)	Microservice architecture recovery	-	√	Github analyser, Docker-analyser, TcpDump.Log analyser,	Static, Dynamic	-	-	MicroART-DSL	Rule-based transformation	-	Semi-automatic	MicroART
Microlyze (2018)	Microservice architecture recovery	-	-	-	Static, Dynamic	-	-	-	-	-	Semi-automatic	-
Mayer and Weinreich (2018)	Microservice architecture recovery	-	√	Swagger Data Collection Library	Static, Dynamic	-	-	Data model	-	-	Semi-automatic	-

Chapter 4

Research Methodology

4.1. Introduction

This chapter describes the research methodology used to accomplish the objectives of this thesis. The systematic mapping study, as discussed in Chapter 2, identified the area in which the new research was to be conducted. This involved collecting available publications in the field to discover any specific areas of microservice architecture that have not yet been explored, and outlined the background and principles of the study, as discussed in Chapter 3. While the subsequent chapters have their own methodologies, this chapter discusses the general methodology used to conduct the research covered in this thesis as a whole. The chapter is structured as follows: first, I provide an overview of the research methodology followed in general and how I developed the Microservice Architecture Recovery (MiSAR) approach. Next, I briefly discuss the selected systems for the different studies conducted in this thesis.

4.2. Research Methodology

The primary objective of this research is to provide architectural recovery support for the emerging microservice architectural software style which addresses the problem of understanding the complexity of microservice architecture. In order to accomplish this objective, an appropriate research methodology design is required. The research used empirical studies to build and evaluate the approach from empirical data. I follow the 'design science' methodology. This addresses the concept of design from a scientific perspective, and has been widely adopted by the information system research community, to the point where it is now considered an equal alternative to natural and behavioural research (Hevner, 2007). Essentially, it is a problem-solving approach based on actions that create and evaluate artefacts for specific problems. The design science paradigm is an ideal methodology for software engineering research due to the synthetic nature of the field (Hevner et al., 2004).

The objectives of the proposed research in architecture recovery and Model-Driven Engineering (MDE) in microservice systems are synthetic; therefore, the design science paradigm is an ideal research methodology for the proposed topic. Figure 4-1 illustrates the overall research framework of information system artefacts, which is centred around the ‘build and evaluate’ process, the cornerstone of the design science paradigm. Note from Figure 4-1 that the relevant problems are identified by the contextual environment, while the relevant works and knowledge gaps are determined by the knowledge base. Appropriate metrics are devised after an artefact is built for a particular problem, in order to evaluate the artefact’s performance and its effects in solving the target task (Hevner et al., 2004). In the context of this research project, the developed artefact will be architectural metamodels, a mapping rules between the architectural metamodel and microservice systems, and a recovery process that includes both the latter two artefacts.

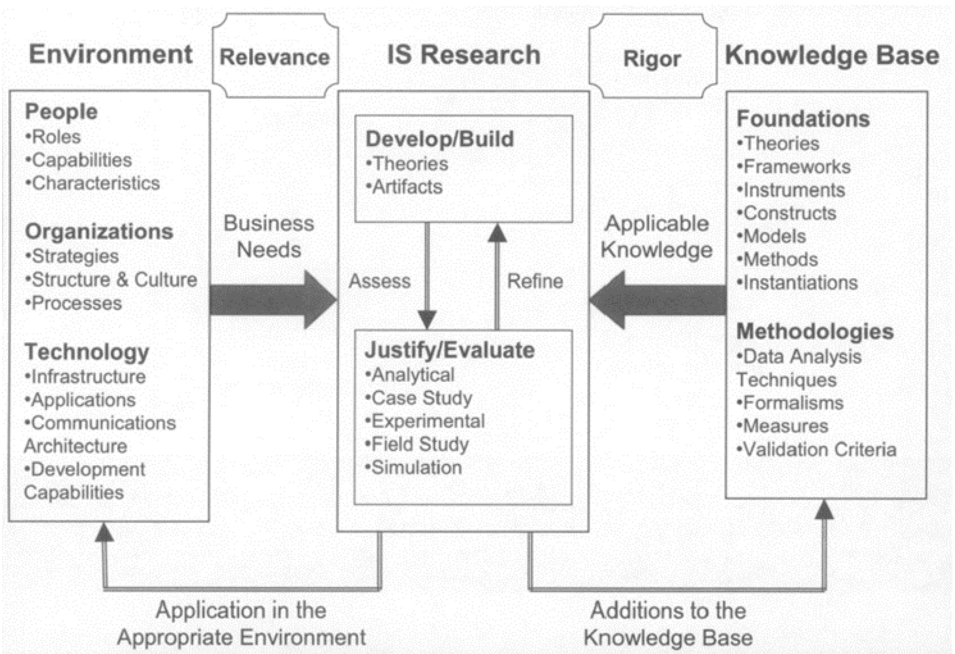


Figure 4-1: Research framework (Hevner et al., 2004, p. 80).

The following are the steps that constitute the design science research methodology (Hevner et al., 2004):

1. Identification and description of relevant problems
2. Validating that no solutions currently exist for the problems identified in the first step
3. Proposing a novel artefact to address the relevant problems
4. Evaluating the utility of the proposal
5. Examining the added value provided by the artefact to the knowledge base
6. Explanation of the practical and developmental aspects of the solution.

The relevant problems addressed by this study in software architecture recovery in microservice systems were determined by examining the literature and conducting a systematic literature review (steps 1 and 2), as presented in Chapter 2. The study used two qualitative and quantitative synthesis methods.

A first major objective of this thesis was to identify and build the artefacts of MiSAR. Therefore, I designed and conducted a first empirical study, as presented in Chapter 5 (see Figure 4-2), which included a manual and iterative recovery process based on eight open-source microservice projects from the GitHub repository (step 3). This study included two main phases: Recovery Design (RD) and Recovery Execution (RE), which are iterative and incremental. The first phase attempts to plan the recovery by defining the architectural concepts along with the mapping rules. In the second phase, I executed the plan for validation purposes, and applied the metamodel and mapping rules defined in the first phase to create architectural models manually. The aim of the first study was to identify the concepts and elements needed to build a metamodel of a microservice-based system, and to develop mapping rules that derive a target model from the source model.

To achieve this, three research questions which the study needed to address were defined:

RQ5-1*: What are the microservice architectural elements/concepts that are identified from the source code?*

RQ5-2*: What are the mapping rules between the source code of microservice implementations and the architectural model?*

RQ5-3*: What kind of software analysis is needed to capture the microservice architecture?*

The outcomes of this empirical study included initial artefacts, the metamodel and the mapping rules of the MiSAR approach, which are artefacts that are used to recover architectures of microservice systems manually.

After that, I conducted a second empirical study on nine open-source microservice projects as presented in Chapter 6. This study focuses on validating and enhancing MiSAR artefacts incrementally and achieve improved artefacts; each artefact of the proposal is to be tested and validated by a medium-scale open-source system in an iterative evaluation loop fashion (Step 4). This enhancement and refinement are essential as part of the 'build and evaluate' loop (Hevner et al., 2004). I manually apply the initial MiSAR artefacts to a set of microservice open projects, which are implemented in Java, Docker and Spring Cloud frameworks. The design of the study includes four activities, as depicted in Figure 4-2. Activities 1 (Application to metamodels) and 2 (Application to mapping rules) to enhance and refine MiSAR in increments. Activity 3 includes implementing the MiSAR artefacts and 4 includes recover an architectural model represented in a diagram. To achieve this, three research questions which the study needed to address were defined:

RQ6-1*: What are the enhancements that have to be performed to the existing MiSAR metamodel to represent more richly recovered architectural models of microservice systems?*

RQ6-2*: What enhancements have to be applied to the current MiSAR mapping rules that map microservice Java and Spring Cloud systems into architectural models?*

RQ6-3: *Can an enhanced MiSAR approach recover architectural models?*

The outcome of this empirical study was a final version of the MiSAR artefacts, including the PIM metamodel, PSM metamodel and mapping rules, which are artefacts that are used to recover architectures of microservice systems in an automatic manner.

Finally, in the last study as presented in Chapter 8, I applied the final version of the MiSAR artefacts via a large-scale microservice system, involving a case study in an industry setting, to show the usefulness of the MiSAR elements and evaluate the recovery approach. This study focuses on the integration of all MiSAR artefacts and

applies them to obtain an architectural model. Along with this, the case study has been adopted to answer the following research questions:

RQ8-1: *What is the degree of completeness of the recovered microservice architecture model?*

RQ8-2: *What is the degree of correctness of the recovered microservice architecture model?*

RQ8-3: *Is the execution time of MiSAR transformations by QVT efficient or not?*

Most case studies in the field of software engineering research use examples from Figure 4-1 to evaluate the artefact design (Myers, 1997). Quantitative methods were used to evaluate this approach e.g., identifying inconsistencies. Finally, the conclusions on the quality and the effectiveness of the proposal were based on the outcome of the evaluations of the empirical studies (steps 5 and 6).

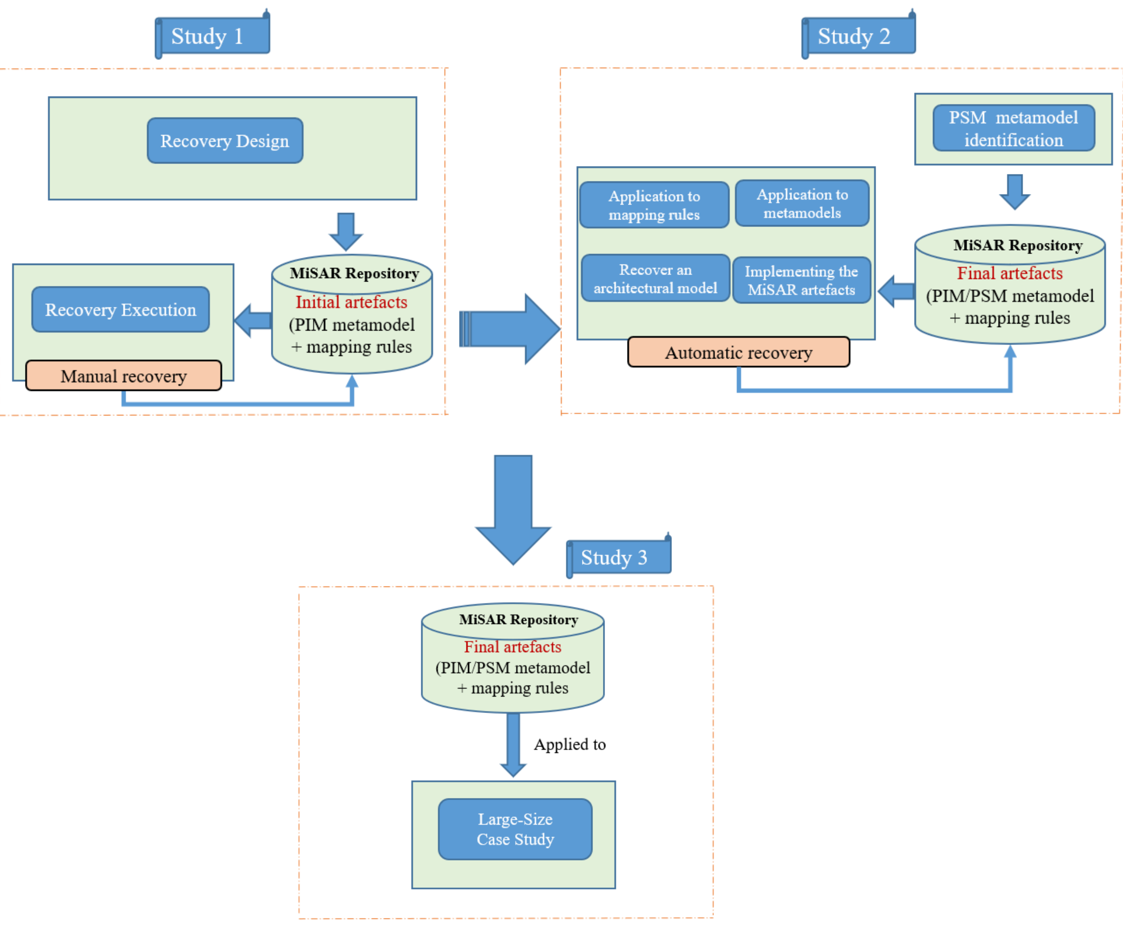


Figure 4-2: MiSAR methodological approach.

4.3. System Studies

Empirical studies were conducted on systems, based on open-source projects from the GitHub repository[7] that employed microservice architecture. The most significant reasons why my approach was based on the GitHub repository analysis are:

(i) GitHub is the most famous open-source code repository, and Spring Cloud has published its framework in GitHub for everyone in the world to reuse.

(ii) The repository that I selected demonstrates best practices from Spring Cloud/Boot framework based microservices, which is widely regarded as the most famous microservices framework in the industry.

(iii) The repository that I selected contained the configurations, code, documentation and best practice integration amongst various components in a typical microservices architecture.

(iv) The code can be obtained by anyone from GitHub and automation can be done to perform automated analysis, to validate the findings based on historical, current or future releases of the code.

The selected case studies demonstrate microservice architecture patterns using Spring Boot and Spring Cloud technologies. The popularity of the Spring Cloud framework in microservice architecture was the reason for selecting this framework for this thesis. Microservices with various tools from Netflix OSS, such as Eureka, Ribbon and Hystrix, support the Spring Cloud framework efficiently. The framework is being adopted by the industry for system development, which is evident in that as of November 2016 it was downloaded 10.2 million times (LONG, 2016). In comparison to the download figures for 2015, this signified growth of 425%. Developers are provided with various tools by the Spring Cloud framework that help them in making common patterns promptly for distributed systems. For the purpose of running and constructing microservices, the Spring Cloud framework provides the most suitable environment (Ibryam, 2016; Woods, 2015).

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

4.4. Summary

This chapter has described the general methodology used to address the questions which this thesis attempts to answer. The proposed methods that were used for conducting the research studies have been briefly described. Detailed and designated techniques that are based on the research methodology are presented in the following chapter.

Chapter 5

Microservice Architecture Recovery (MiSAR)

5.1. Introduction

This chapter presents the Microservice Architecture Recovery (MiSAR) method, MiSAR is an approach which follows a Model-Driven Engineering (MDE) framework. The approach aims to recover the architecture of microservice-based systems from the implementation level to the architecture level. In order to formalise the approach, MiSAR was developed from empirical data to define metamodels of the underlying platform for different aspects of microservice environments and the mapping rules that support the architectural recovery of a microservice system. The basic goals and high-level description of the approach are discussed first, before a detailed explanation of each step in the study. The MiSAR artefacts which are the results of the empirical study are then presented; these are a metamodel and mapping rules. Finally, the chapter outlines how these two components allow one to manually recover the architecture model of a microservice-based system.

5.2. Overview of MiSAR

The complexity of the microservice architectural style makes the task of understanding its many artefacts very difficult, as applications consist of many small components, interfaces and dependencies. The ideal way to comprehend these complexities is to model the artefacts themselves as accurately as possible. MiSAR follows MDE (Brambilla et al., 2017; Kent, 2002) to recover architectural models of microservice-based systems, by developing bottom-up, model-driven transformations for obtaining architectural models from the implementation level. MDE is particularly suited to the distributed, fine-grained nature of microservice architecture systems. In addition, one of its competitive advantages is that modelling occurs at multiple abstraction levels, which helps elucidate a model-driven transformation for a more holistic approach to architecture.

MiSAR focuses on the Platform-Independent Model (PIM) alongside the Platform-Specific Model (PSM) abstraction levels in relation to the modelling of microservice architecture platforms. These models are critical in order to better understand the core of reverse engineering, where the PIM supports the architectural model recovered and the PSM supports the technology of the implemented microservice system. The MiSAR process is based on the transformation from code to PSM to PIM, as shown in Figure 5-1. This implementation pathway has a process that includes code, XML, YAML files, schema, run-times, etc., that are converted into the PIM. This is achieved by providing mapping rules from which these models can be derived. Two key components of MiSAR are a metamodel, which abstracts the concepts of a microservice architecture in a technology-independent manner, and mapping rules, which map an implemented microservice-based system into an architectural model which instantiates the metamodel. The MiSAR approach generates architectural models of microservice-based systems. The following section presents a study based on a systematic analysis which allowed the definition of MiSAR's metamodel and mapping rules based on empirical data.

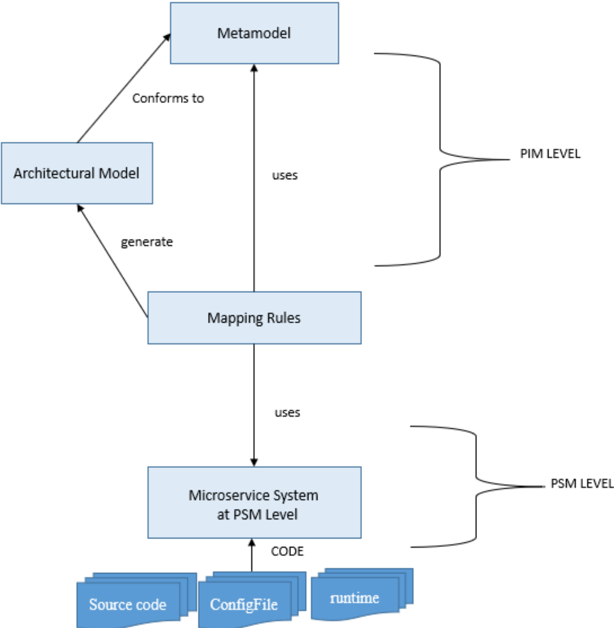


Figure 5-1: Approach overview.

5.3. Microservice Application Platform-Specific Model

Platform-specific models in microservices can be conceptualised into two areas, runtime platform and development technology. The runtime model provides information that helps understand the connectivity and orchestration. Microservices can be packaged into runnable images, which can be a Docker-based container or an open VM format. It would then become important to recover information related to ports that different microservices communicate with, and IP addresses that are assigned to individual running instances of a microservice. It is also important to understand the routing and load balancing that is employed. Even though it may not give insights into how a microservice is internally structured.

The other area of focus in a PSM for microservices is the development technology or the frameworks which are used to accelerate the development of the microservices. There are opinionated configurable frameworks, such as Spring Boot, which provide a range of design pattern implementations to make microservice development rapid for a developer. These patterns include service discovery, client-side load balancing, etc. Employing a framework like Spring Boot would require the PSM to provide details such as a Spring Boot configuration YML file, boot version and some of the auto-configuration features that Spring Boot provides.

The PSM would include traditional implementation aspects such as classes, interfaces and packages that form a microservice business domain model. As for boot runtime, it may include boot-based services that are used in the architecture, such as the Netflix Eureka discovery service, Spring Cloud Config server, Netflix Hystrix for circuit breaking, etc. Such a PSM would provide a comprehensive business and runtime model of a microservice system. Figure 5.2 depicts a mapping of the microservice concept at the PIM level with the different files at the PSM level.

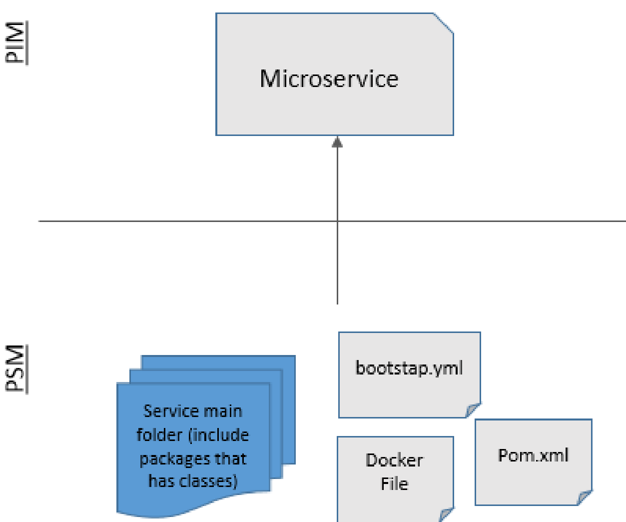


Figure 5-2: Representation of microservice concept at PSM/PIM layer.

5.4. Empirical study to define MiSAR

As described in section 5.2, MiSAR follows an MDE approach, and thus needs to define a metamodel and mapping rules that allow the architectural recovery of a microservice system. To be able to define these MDE elements, a study was designed that feeds into these elements. As the objective of the study is architecture recovery, it was designed as a manual architecture recovery process. I customised the process presented in van Deursen et al. (2004), which includes two main phases: Recovery Design (RD) and Recovery Execution (RE), as depicted in Figure 5-3. Typically, the two phases are iterative and incremental; the metamodel and mapping rules evolve and are refined throughout the process. The first phase attempts to plan the recovery by defining the architectural concepts along with the mapping rules. The second phase involves executing the plan for validation purposes and applying the metamodel and mapping rules defined in the first phase to create the architectural models. The outcome of the validation may lead to the steps being repeated, by refining the metamodel and mapping rules, and re-validating.

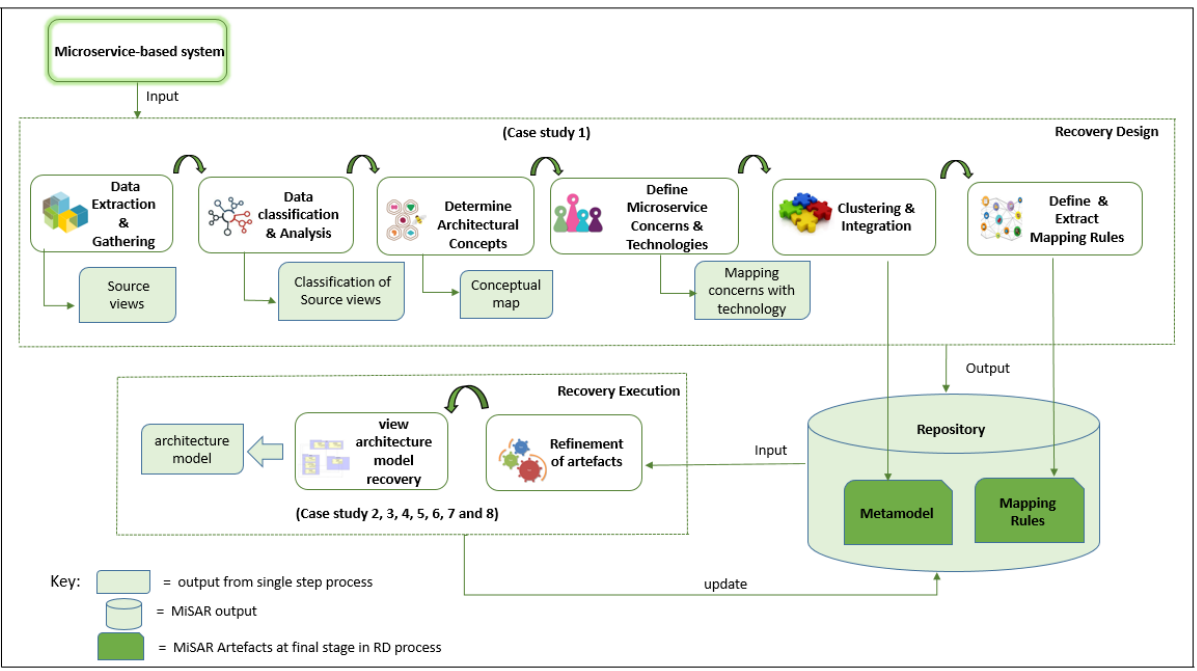


Figure 5-3: Study steps.

This study aimed to develop MiSAR from empirical data. To achieve this, three research questions which the study needed to address were defined (see Table 5-1). The answers to these can be successfully obtained by defining the metamodel and mapping rules, which correspond to the artefacts of the MDE approach (RQ1, RQ2). As the study is reverse engineering from a software system, the kind of software analysis to be conducted for extracting the microservice architecture needs to be classified as either static or dynamic (RQ3).

Table 5-1: The research questions and their motivation

Research Question	Motivation
RQ1: What are the microservice architectural elements/concepts that are identified from the source code?	The aim is to identify the concepts and elements needed to build a metamodel and a specific-purpose abstraction of the microservice-based system.
RQ2: What are the mapping rules between the source code of the microservice implementations and the architectural model?	The aim is to develop mapping rules that derive a target model from the source model.
RQ3: What kind of software analysis is needed to capture the microservice architecture?	The aim is to evaluate and assess the needs of static and dynamic analysis in the process of system recovery within the microservice framework.

5.4.1. Selection of Systems to Study

I selected open-source projects from the GitHub repository that employed microservice architecture. I began by performing a search on the repository facility using the terms “microservice”, “micro service”, “micro-service” and “micro service architecture”. Specific criteria were applied to support project relevance, as stated in Table 5-2. The study was limited to eight systems, as listed in Table 5-3.

Table 5-2: The selection criteria

	Criteria
Inclusion	• Projects that Implemented with Spring Boot/Spring Cloud framework in java language including any technology that integrates with the framework (e.g. Netflix OSS). • Projects that each of its modules run in a single process (Docker technology). • Projects which demonstrate usage of the microservice architectural style (determined by asking developers and reviewing documentation). • Projects that consist of an accumulation of independent individual services. • Projects that implement business functionality.
Exclusion	• Projects that use two or fewer Spring Cloud components (Projects that use fewer Spring Cloud libraries decreases the probability of the project being a microservice). • Projects that not use Spring Boot/Spring Cloud framework. • Projects that include only infrastructure microservice (e.g. development tools, operation frameworks). • Projects that use less than two functional microservices. • Projects that do not revolve around an accumulation of independent individual services.

Table 5-3: Studies selected for analysis

ID	Project Name	Project Repository URL	Microservice Count	LOC[8] Size	No. of Developers	Project Timeline	Documentation	Architecture Diagram
1	piggymetrics	https://github.com/sqshq/PiggyMetrics	13	3309	5	Mar 29, 2015 – Aug 17, 2017	Available	Available
2	microservice-blog	https://github.com/3PillarGlobal/microservice-blog/tree/part4/step3	7	474	1	Aug 30, 2015 – Jan 4, 2018	Not available	Not available
3	spmia-chapter10	https://github.com/carnellj/spmia-chapter10	7	2261	1	Jun 30, 2017 – May 13, 2017	Not available	Not available
4	microservice-consul	https://github.com/ewolff/microservice-consul	11	2434	3	Jun 19, 2016 – Jan 4, 2018	Available	Not available
5	spring-cloud-consul-example	https://github.com/yidongnan/spring-cloud-consul-example	7	286	1	Jun 5, 2016 – Jan 4, 2018	Available	Available
6	spring-cloud-netflix-example	https://github.com/yidongnan/spring-cloud-netflix-example	9	328	1	Jun 5, 2016 – Jan 10, 2018	Available	Available
7	microservices-sidecar-example	https://github.com/xetys/microservices-sidecar-example	5	2434	1	Dec 20, 2015 – Jan 4, 2018	Not available	Not available
8	blog-microservices	https://github.com/callistaenterprise/blog-microservices	14	2093	1	Mar 1, 2015 – Jan 4, 2018	Not available	Not available

[8] Line Of Code

5.4.2. Research Design

The study has two main phases: recovery design and recovery execution. In the RD phase, the study analysed case study (ID=1, Table 5-3); this case study was chosen due to the availability of its architecture documentation and supporting diagrams with illustrations, which can be used to compare the results of this phase with the documentation. Case studies (ID=2-to-ID=8, Table 5-3) were used in the second phase for refining and enhancing purposes. The steps, techniques and tools taken in each phase are described in the subsequent sections.

5.4.2.1 Recovery Design Phase

During RD, the microservice architectural concepts that build the system were determined, and the mapping rules for the code and the architectural concepts were identified. Within the RD phase, PiggyMetrics (ID=1, Table 5-3) was analysed in steps in order to define the metamodel and mapping rules. These steps are separated into the following:

Step 1 – Data Extraction and Gathering: This step involves the collection of artefacts (source code and other documents) and reviewing them to search for information about the system. The artefacts are gathered in an effort to build the knowledge base for the software system. Next, data extraction is employed in order to understand and gather the required data from the software system for the recovery of microservice architecture. Data extracted from artefacts should include the most indicative elements and lines in the source code, configuration files, descriptive files, etc., which are then collected and stored within a data repository as a PSM model.

Technique: Data was extracted from the following artefact files:

- **Docker Compose files:** These are YAML files used for defining and building multi-container applications, hence they provide a complete view of the microservice repository of the architecture system.
- **Dockerfile files:** These are script documents that contain all the commands a user could call on the command line to assemble an image in order to run a container and/or a service. One Dockerfile can identify one microservice in the

architecture, since every microservice application will be containerised into one Docker container by running one Dockerfile.

- **Maven POM files:** These are XML files that include information regarding the modules of each multi-module project as well as details regarding the components and libraries utilised by Maven in order to build each module project. Adopting the Spring Cloud style in developing microservice architecture systems, the multi-module application corresponds to the entire architecture system, while the module project corresponds to a microservice. The Netflix OSS libraries attached to a module application's POM represent the static infrastructure components of a microservice.
- **Gradle Build and Gradle Settings files:** These are script files that are equivalent to Maven POM files in functionality and representations for a microservice architecture system.
- **Spring Configuration files:** Configuration files define various runtime properties and settings that are used by the Spring Boot framework to initialise various components in the execution environment. Local configuration files exist in the project folder, while centralised configuration files are stored in a remote shared location, as defined by the configuration infrastructure microservice.
- **Java source files:** This is the actual source code of the program that contains the business logic for the microservice. In particular, the microservice's role, service endpoints and inter-service communication implied by the source code are essential to the recovery of microservice architecture behaviour. Important PSM concepts in Spring Java source files include class annotations, controller's method declarations, RESTful request calls and POJO class declarations.
- **Documentation:** This includes textual and graphical descriptions of the architecture, in addition to instructions on how to compile, deploy, operate, integrate and use the microservice-based application. Usually, this is the basis for recovery validation.

Output: The outcome of this stage when executed on case study 1 (PiggyMetrics) is a repository which contains data on the source files, as shown in Appendix A, 1.

Step 2 – Data classification and analysis: Different kinds of analysis, static and dynamic, contribute different kinds of information to the data flow.

Technique: Two kinds of iterating analysis were conducted for the extracted data: static analysis and dynamic analysis.

- **Static analysis:** Static analysis is performed by observing only the artefacts of the system. In microservice architecture, services run in isolated environments, similar to Docker containers. Static analysis can extract services' descriptors from these environments from the Dockerfile or the Docker-compose.yml files. These descriptors, written in declarative languages like YAML, describe the properties and configurations of each service during its build and run stages (build path, middleware configurations, service name, ports, ports mapping, etc.). Information extracted from these files behaves similarly in any environment (development, testing and production environments, etc.). Examples of service descriptors can be found in source code, software infrastructure and files like Docker, RKT, Vagrant files. To extract a static view of the system for the Java source code file, two reverse engineering tools were used: Enterprise architect[9] and visual paradigm.[10] These tools were applied to the Java source code to generate UML class diagrams, as shown in Figure 5-4.

[9] http://www.sparxsystems.com.au/products/ea.
[10] https://www.visual-paradigm.com.

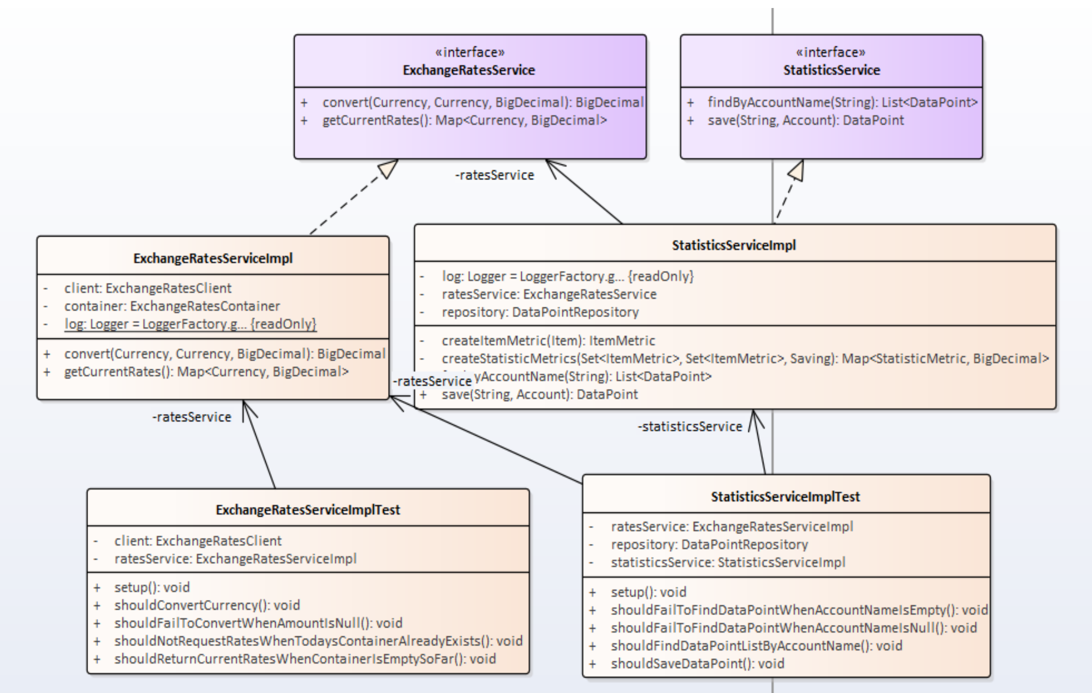


Figure 5-4: Packages and classes extracted from source code.

- **Dynamic analysis:** Dynamic analysis is performed by observing the system during execution, and aims to extract information from the running code at a production runtime. The Zipkin server was implemented and enabled with case study 1,[11] then the Zipkin tool[12] was used to trace communication between microservices, as in Figure 5-5, so that a call graph from one microservice to another could be built. Zipkin captured all the calls and dependencies between different microservices, as shown in Figure 5-6. TCPDump provided low-level TCP protocol connectivity and communication, which provided information about the latencies between different components of the system.

Output: The outcome of this stage is a fusion of extracted information from both static and dynamic analysis, as shown in Appendix A, 2.

[11] https://github.com/nuha77/piggymetric-with-Zipkin.
[12] https://zipkin.io/.

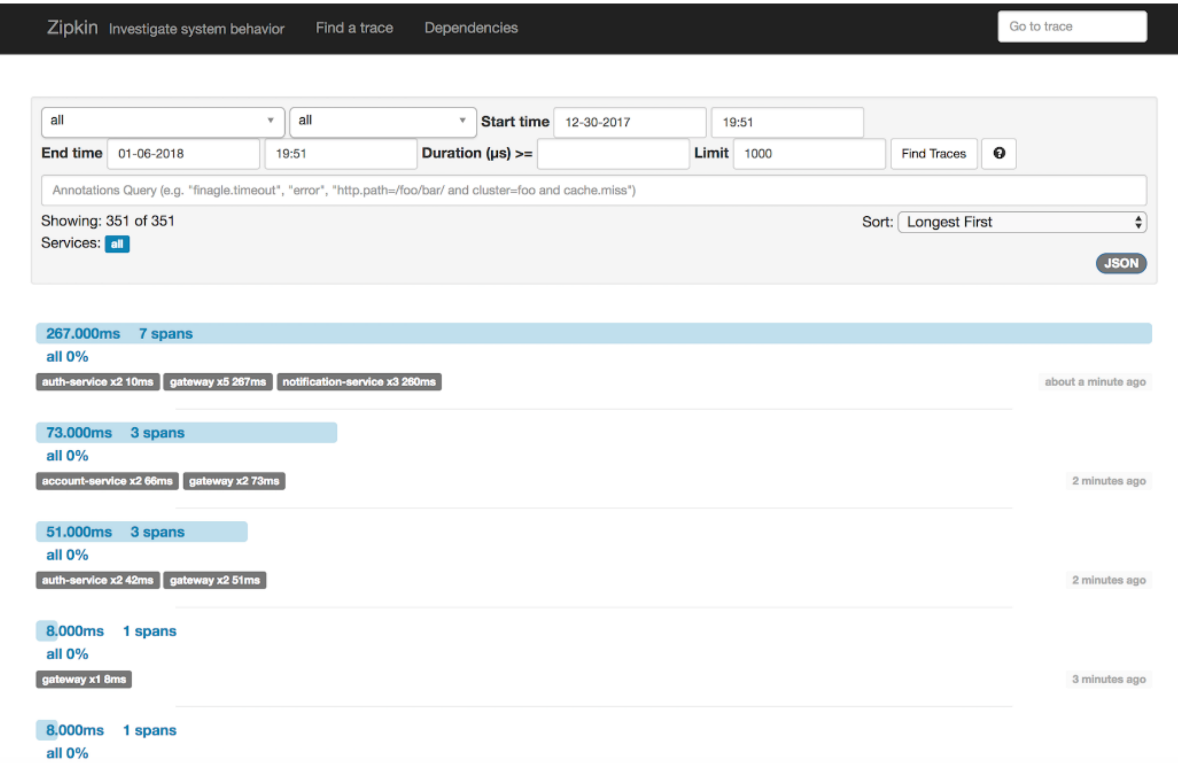


Figure 5-5: Using Zipkin to trace transactions.

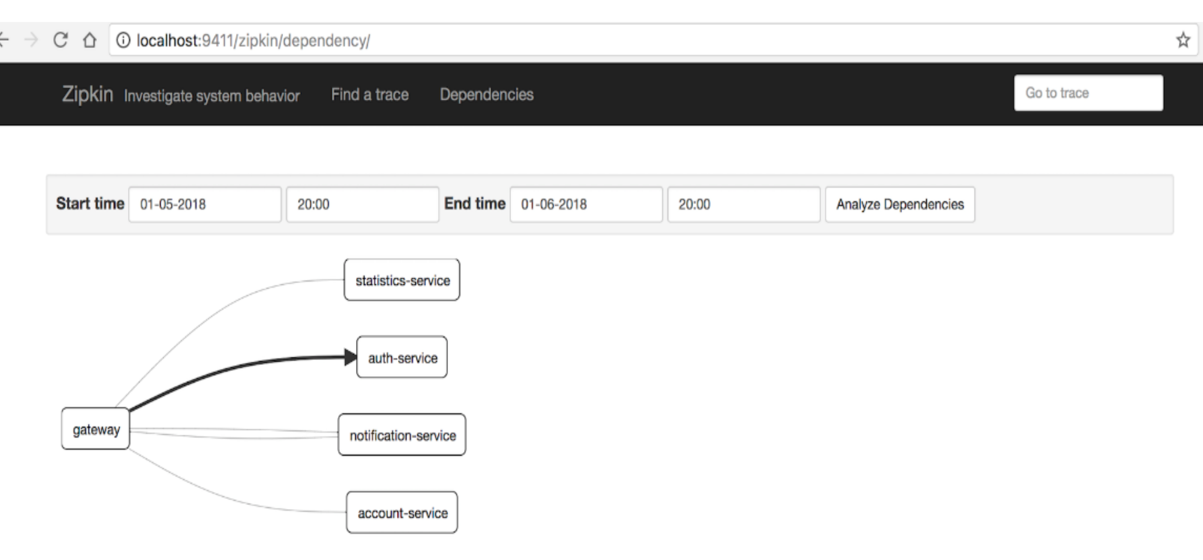


Figure 5-6: Dynamic analysis using Zipkin.

Step 3 – Determine Architectural Concepts: Architecture dimensions or abstracts, known as 'concepts', are considered the first building blocks of the recovery process (Riva, 2000). When determining architectural concepts, information about the relevant architecture that was used to build the system is obtained. The architecture concepts represent the terminology of the recovery process and the elements at the architecture level. Therefore, the heart of this work lies in the following question:

"Is the identified concept considered an architectural element or not?"

In order to select the proper architectural concepts, the architectural concepts which are relevant to microservice architecture and that overlap with those of any system built using traditional (e.g. monolithic) architecture were evaluated. A microservices style architecture is structurally very different from a traditional architecture system, in that the system is usually composed of many microservices. Moreover, the concepts must be independent of any technology and hide the details of any particular platform.

Technique: Both bottom-up (code-to-model) and top-down (model-to-code) techniques were used to understand and determine the microservice architectural concepts. The mechanism used for the bottom-up technique is to start from the level of code (program layer) and analyse the source artefacts (e.g. source code, configuration file, etc.). The concepts are discovered after abstracting and evaluating their relevance to architectural elements (concept layer), as illustrated in Figure 5-7, and via the more concrete example in Figure 5-8. On the other hand, the top-down (model-to-code) technique focuses on literature, allowing the identification of several concepts and supporting the definition of the underlying features and behaviour of microservice-based systems. Microservice architectural concepts were determined that focus on using microservice patterns (Richardson, 2018). According to the service-type classification, Richardson (2018) discussed that the classification of services is of two types, infrastructural and functional.

Output: The outcome of this stage is a conceptual map which contains the identified concepts from both techniques that are relevant to the analysis, as shown in Figure 5-9.

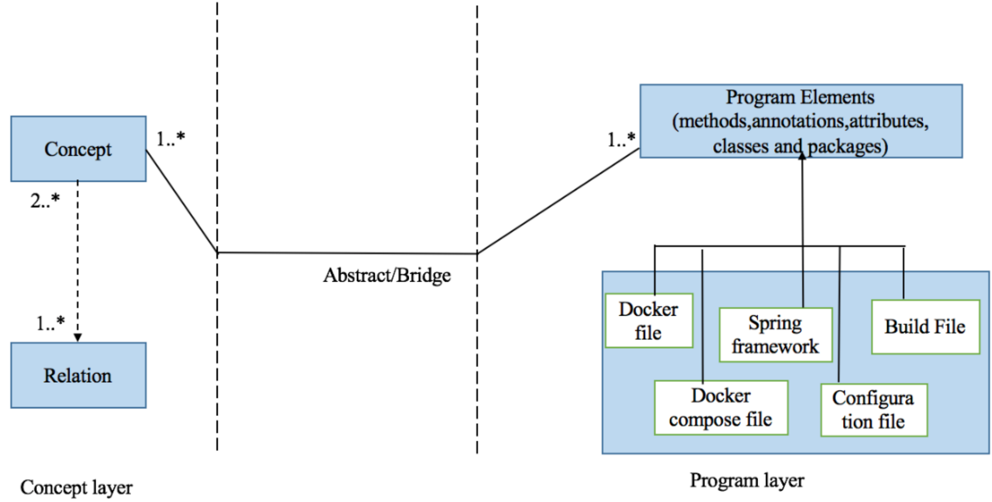


Figure 5-7: The bridge concept.

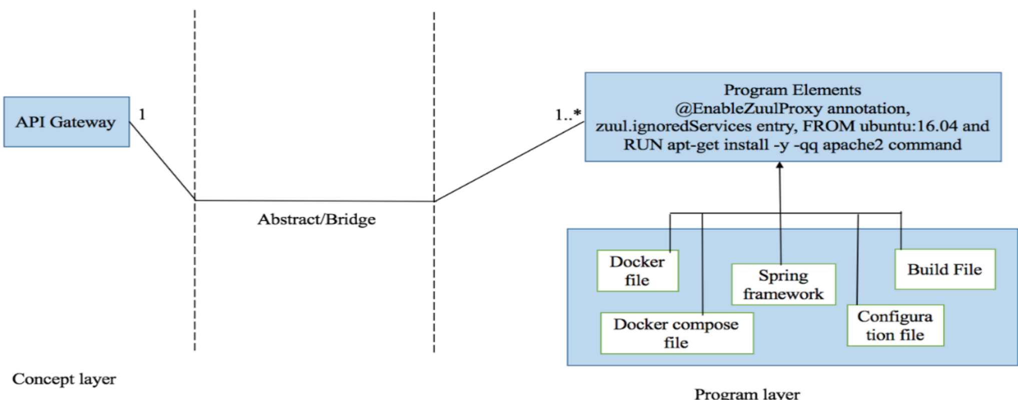


Figure 5-8: Concrete example for API gateway concept.

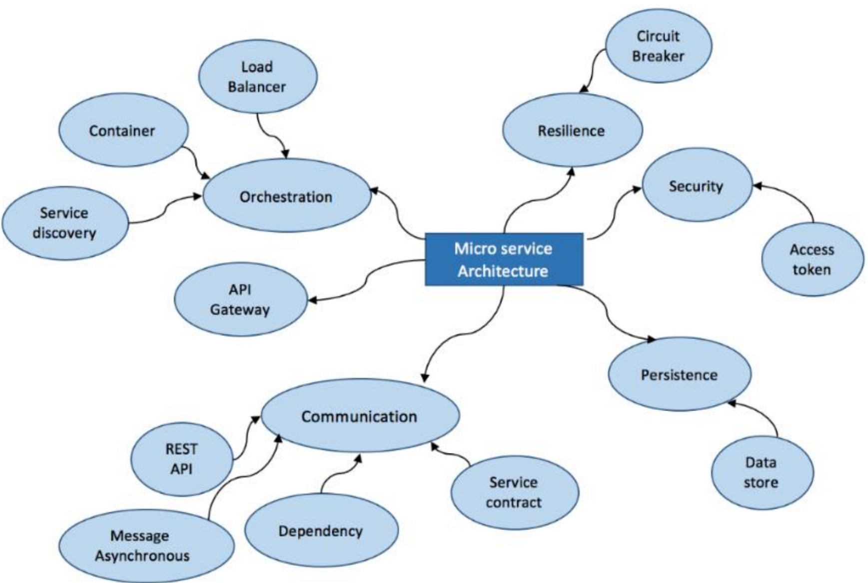


Figure 5-9: Microservice conceptual map.

Step 4 – Define Microservice Architectural Concerns and Technologies: Concerns are common characteristics of microservice architecture, which can be commonly implemented across multiple microservices. These can be related to making microservices fault-tolerant, ease their deployment and discovery. The focus of this study is the most common technical concerns, inspired by Ibryam (2016), presented in Figure 5.10; non-technical concerns such as organisation structure, culture and so on are excluded.

Technique: Several concerns appeared in Step 3 as shown in the conceptual map (Figure 5-9). Concerns are difficult to identify from the code, as in Ibryam (2016) the literature was used to review the most common ones that have to be considered in microservice systems. The technologies that are commonly used to implement these concerns were then identified in a Spring Cloud OSS-based microservices environment, as shown in Table 5-4. Recovering these technologies can help determine whether a given microservice implements a specific concern. This will help in identifying and building the relations between various platform-specific services, and the functional or business services in the PIM.

Output: The outcome of this stage is a list of concerns to be taken into account in the microservice architecture.

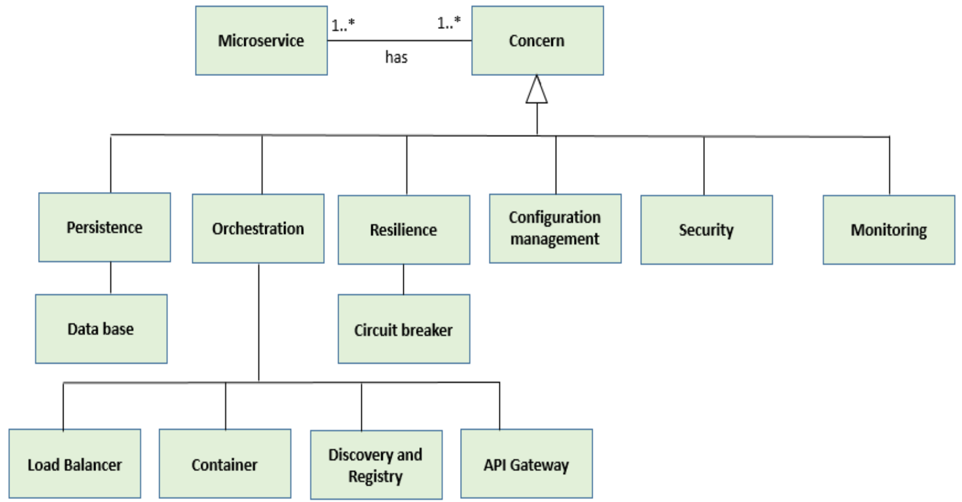


Figure 5-10: Microservice architecture concerns.

Table 5-4: Technology mapping to microservice concerns

Microservice concern	Technologies (Spring Cloud)
Service discovery	Netflix Eureka, Hashicorp Consul, etc, zookeeper
Load balancing	Netflix Ribbon
API gateway	Netflix Zuul, APIARY, APIGEE
Configuration management	Config server, consul, Netflix archaius
Service security	Spring Cloud security
Distributed tracing	Spring Cloud sleuth, Zipkin
Monitoring & resilience	Netflix Hystrix, Turbine & Ribbon
Centralised logging	ELK stack, elasticsearch, elasticserach logstash, Kibana Splunk
Centralised metrics	Netflix spectator & Atlas, APM
Deployment & scheduling	Spring Boot, container orchestration tool (Kubernetes, Docker swarm, Cloud Foundry, Mesos)
Data store	MongoDB, PostgreSQL, H2
Containerisation	Docker

Step 5 – Clustering and Integration: After identifying the various common architectural concepts in microservices, they were clustered together based on high-level related concerns that had been identified in step 4.

Technique: The technique represents the concepts as meta-classes by grouping related architectural concepts together in one cluster based on their microservice concerns, as shown in Figure 5-11. An association in the metamodel is added for meta-classes that are related. Finally, the concepts and their relationships were integrated and abstracted.

Output: The outcome of this stage is a metamodel for microservice architectures, illustrated in section 5.4.3.

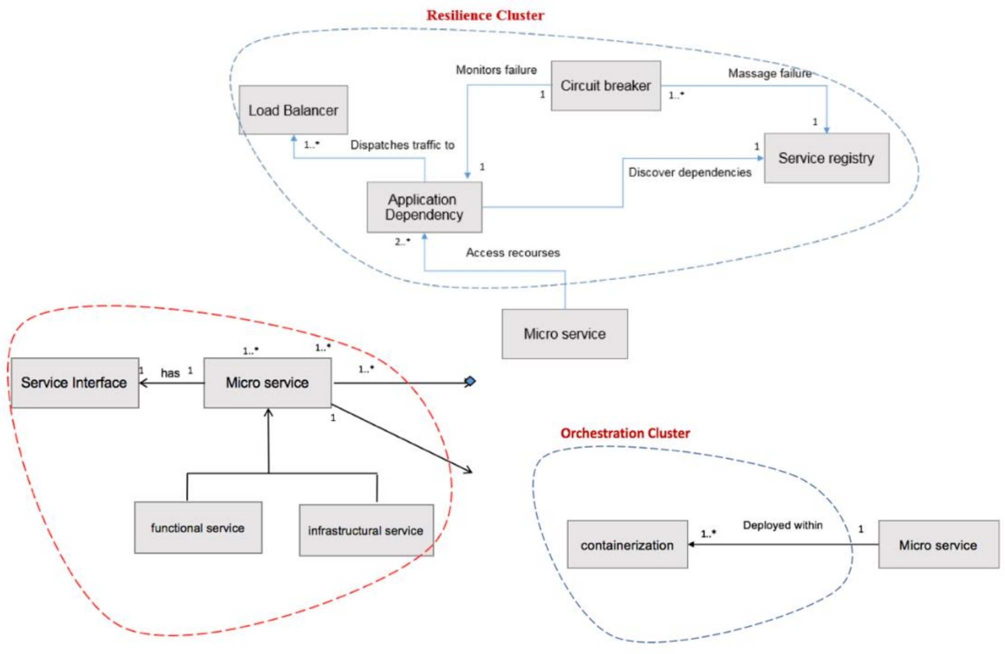


Figure 5-11: Sketch of the related architectural concepts under one cluster.

Step 6 – Define Mapping Rules: The purpose of this step is to identify how the architectural concepts are represented in the source code and how they are mapped in the implementation.

Technique: To define the mapping rules, the system was manually inspected and examined by analysing the source files available in the project directory. Then, for each concept in the metamodel, the extracted files of that concept were analysed to define the mapping rules which map architecture concepts with implementation artefacts. The mapping rule extraction process was performed at two analysis levels: microservice system level and microservice level. The microservice system level involves analysing the Docker compose files, and Multi-Module project build files, generated either by the Maven build tool (e.g. pom.xml) or by Gradle (e.g. settings.gradle). The microservice level involves analysing the Module project build, Spring configuration, Java source and the Docker files. Information collected for each mapping rule included the input PSM (artefact) being studied (e.g. container orchestration file, project build file, source code file, etc.), and then mapping architecture concepts (e.g. microservice, service dependency, service interface, registry and discovery, API gateway, etc.) into implementation artefacts. Mapping rules were then classified and grouped based on the output architectural element they mapped to.

Output: The outcome of this stage when executed on case study 1 (PiggyMetrics) is a set of mapping rules for each concept in the metamodel, illustrated in section 5.4.3.

5.4.2.2. Recovery Execution Phase

A validation for the results obtained from the RD phase was conducted, using case studies 2 to 8, as listed in Table 5-3. The RE phase includes two steps, as follows:
Step 7 – Refinement of artefacts: The metamodel and mapping rules obtained in the RD phase are applied and validated against the seven case studies for enhancement and validation purposes.

Technique: The seven system implementations were analysed manually, and the mapping rules and metamodel were then applied. Based on the success of this analysis, the mapping rules and architectural elements were amended and enhanced. The case studies analysis can be found in Appendix A, 3.

Output: The outcome of this stage is an updated MiSAR repository with refined mapping rules and the metamodel as illustrated in section 5.4.3.

Step 8 – View Architecture Model Recovery: After refining the MiSAR repository in the previous step, a manual architecture recovery process was formulated. Notations to visualise the recovered model graphically are also defined in this step. The process and notations are confirmed by the application to case study 1, previously analysed, in order to easily ensure the consistency of the results with the documentation. Later, new and more complex case studies are considered to evaluate the process (discussed in Chapter 6).

Technique: The architecture model for case study 1 (PiggyMetrics) was recovered by applying the manual steps: Setup, Recovery and Visualisation as illustrated in section 5.4.3.

Output: The output of this stage is the instance diagram equivalent to a UML object diagram and architecture diagram conforms to the PIM metamodel for the recovered model diagram of case study 1 as illustrated in section 5.4.3.

5.4.3. Results

This section presents the resulting metamodel and mapping rules after the analysis.

- **RQ1: Microservice metamodel**

Regarding RQ1, there are various architectural elements which are fundamental to any system. Therefore, they appear across all the selected case studies. Figure 5-12 shows the architectural concepts discovered, and the case studies (indicated in numbers) from which they were identified. It can also be observed that various architectural elements appear only in few cases due to various contextual demands of the projects. The context of these cases was analysed, and the need for these elements to be used by the designer of the system was determined.

It can be observed that **Containerisation** appears across all the projects. This is due to the fact that the initial selection criteria for the case studies included the usage of Docker. Docker is fundamentally a containerization technology, hence all the case studies in Table 5-3 use containerization as an architectural element. Docker is the most commonly used containerization technology hence most microservices use Docker as the container image format of choice. **Configuration** is also a fundamental architectural element which happened to have been used across all case studies except studies 2, 5 and 7. This is probably due to the size of these projects. It contains few microservices and the project's objective is a proof-of-concept for microservice development.

API gateway is present in most projects, suggesting that the use of API gateways is very common in microservices. This is due to the fact that the API gateway allows architects to configure cross-functional elements such as security, logging and authorization. This relieves individual services to handle these architectural elements within their code. **Registry and Discovery** were discovered in most projects. However, each project uses different technologies to implement this concept. For example, 5 case studies used Netflix Eureka, while Consul was used in two studies (4 and 5). Again, as with the configuration element, due to the small size of system 2, registry and discovery are not used.

As shown in Figure 5-12, it was found that some concepts are essential in a microservice architecture, and are found in all systems, while others are not. Based on these counts, the metamodel shown in Figure 5-13 was defined. For instance, **Containerisation**, **Microservice (Functional and Infrastructural)**, **Service Interface**, **Load Balancer**, **Endpoint**, **Service Operation** and **Service Dependency** are found in all analysed systems, so when defining the metamodel this would be represented with mandatory associations: one-to-one or one-to-many multiplicities. For example, one microservice should run independently in one container and have at least one communication endpoint. However, **Security** was implemented in only three systems, even though it is an essential concern of a microservice system, and so its association is not mandatory.

It can also be seen from Figure 5-12 that most metamodel elements were discovered in Phase 1, as case study 1 was used in this phase. The metamodel was refined in Phase 2 with two new elements **Message Bus** and **Cache Store**, which were discovered in case study 3. The following section describes the concepts of the metamodel.

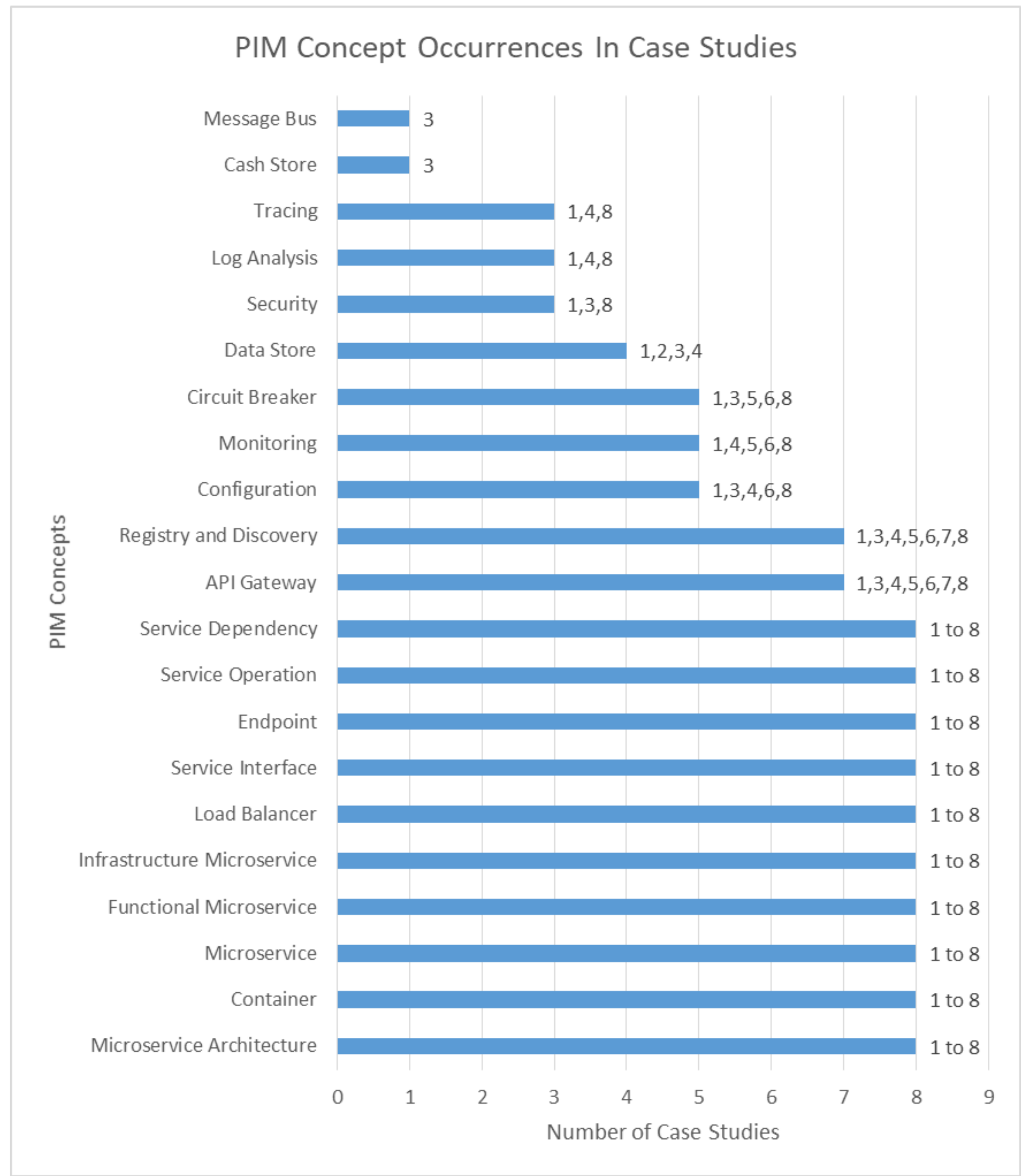


Figure 5-12: Architectural concepts, counts and system references.

5.4.3.1. MiSAR Metamodel: Platform-Independent Metamodel

This section introduces the proposal for a microservice architecture metamodel that can be used to define the basic architectural elements of any microservice system at the PIM level, depicted in Figure 5-13. The following describes the concepts of the metamodel.

Microservice Architecture is the logical repository of microservices. It contains one or many microservice instances, along with the components implementing them. A **Microservice** is the central and main building block of the metamodel, and is generally a software application that offers a complete independent service. In a microservice architecture, there might be multiple instances of the same microservice type, as well as different types, depending on the domain of the microservice system. Since the metamodel is based on static analysis, multiple instances of the same Microservice is out of scope. Microservices are broadly classified as follows. **Functional microservice** types realise a system's business capabilities as well as a set of **Infrastructure microservice** types, which implements an infrastructure pattern/component addressing a particular concern of microservice architecture. **Infrastructure Microservice** types include **API Gateway**, **Configuration**, **Discovery** and **Registry**, **Security**, **Log Analysis**, **Monitoring**, and **Tracing**.

Although the implementation of microservices differs, every microservice instance is defined by at least one service interface. The **Service Interface** element aggregates all **Service Operations** as well as exposed **Endpoints** of a microservice. While an **Endpoint** is the service URI that can be called by remote consumers; it is defined by the path and HTTP method, e.g. GET/POST/PUT, etc. A **Service Operation** reflects the main procedure/function that is directly executed by calling a corresponding endpoint. Unless the microservice instance is stateless, one service operation could interact with a **Data Store** to preserve microservice's state. Alternatively, service operation could communicate with a **Cache Store** to preserve repeatedly requested data from remote microservices in order to decrease the number of future requests. This element contributes to improving the response time of the microservice, especially if the data at the remote microservice does not change often. An asynchronous **Message Bus** could be used by a **Service Operation** in order to write

data to and/or read data from remote microservices in a non-blocking fashion as opposed to the synchronous request-response blocking manner.

The deployment concern of microservice architecture model is represented by **Ambient** and **Container** elements. They describe in which architectural context the microservices are to be deployed. An **Ambient** element is the boundary of a microservice (Hassan et al., 2017). A **Container** is a kind of ambient element. Each microservice instance will be running in exactly one container. A container is an execution environment used to isolate each microservice within one virtual machine, leveraging the host's hardware and operating system capabilities while enabling each microservice to appear as a completely stand-alone software artefact that is running externally (Vaughan-Nichols, 2017).

The **Service Dependency** element describes the communication between one consumer microservice and one provider microservice. One microservice (whether it is a consumer or a provider) can have several dependency instances. This communication takes place as one consumer's service operation invokes one provider's service operation per one dependency instance. It occurs either in a synchronous request-based manner or in an asynchronous message-based manner. A dependency can occur between two different instances of functional microservices, two different instances of infrastructural microservices, or between an instance of an infrastructural microservice and another instance of a functional microservice.

In such an environment, that is rich in communication taking place among multiple instances of microservices, resilience and load balancing requirements are necessary to maintain a healthy execution environment for the microservices. The **Circuit Breaker** pattern/component supports client's resiliency by monitoring requests of a microservice and breaking them if they are experiencing faults or forever waiting. Each service operation can be monitored by one circuit breaker. The role of a **Load Balancer** pattern/component is to periodically fetch addresses of all active instances of remote microservices from discovery server and then caches them locally in a microservice. As a result, this microservice will not need to request discovery server every time an address of a remote microservice is needed and it will also receive fast responses when its requests to the same microservice are balanced over multiple

instances. Like a circuit breaker, a load balancer is optional for any microservice instance, such that one microservice instance may use at most one load balancer.

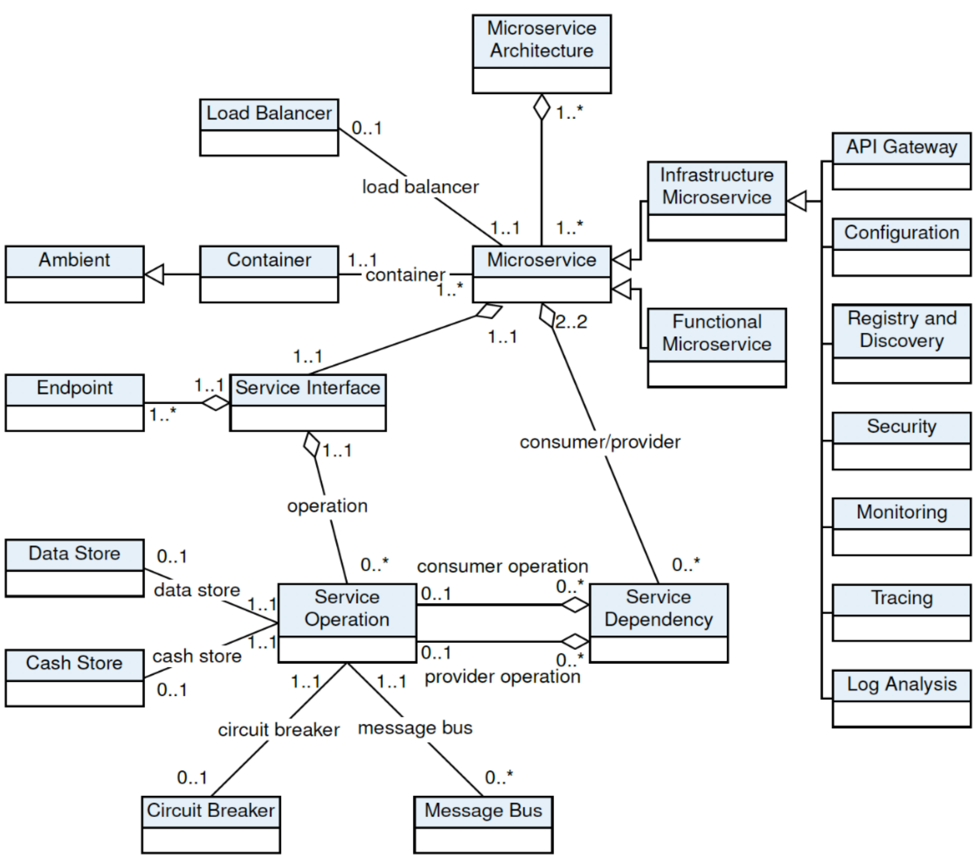


Figure 5-13:Microservice architecture metamodel at the PIM level.

RQ2: Mapping Rules

Regarding RQ2, Figure 5-14 demonstrates the number of mapping rules identified in each case study. It can be noticed that the lion's share of mapping rules was captured in Phase 1 (see Appendix A, 4). In phase 2, the count of mapping rules started to decrease as the number of case studies analysed increased, except for case study 2 which was the smallest in size. The reason for the decreasing trend is that all of the studies were developed using the Spring Boot/Cloud and Netflix OSS frameworks, hence they share common architectural elements. The few rules added were related to specific technologies implemented for specific contexts.

Figure 5-15 shows the number of related mapping rules per architectural concept. A great number of mapping rules reflect the variety in technologies, implementation styles and the artefacts expressing the same architectural elements. To illustrate, the **Service Dependency** concept can occur between functional-to-functional, infrastructural-to-functional and infrastructural-to-infrastructural microservices. It can also be synchronous or asynchronous, as it may require authentication. From the artefacts point of view, **Service Dependency** can be expressed in Configuration and Java Source artefacts using a wide range of properties and Java methods, respectively. The **Data Store** concept, for another instance, was implemented differently, as MongoDB in case studies 1 and 2, as PostgreSQL in case study 3, and as HSQLDB in case study 4.

On the other hand, the **Container** concept was always implemented as a Docker and the **Microservice** concept as a Spring Boot application, which explains their smaller counts of mapping rules. The effect of this was noticeable in the recovery process, where concepts with a standard implementation were faster to recover. In the RD stage where case study 1 was used, 104 rules were identified. In the RE stage, 164 new rules were identified, and 47 rules previously identified in the RD were refined (see Appendix A, 5).

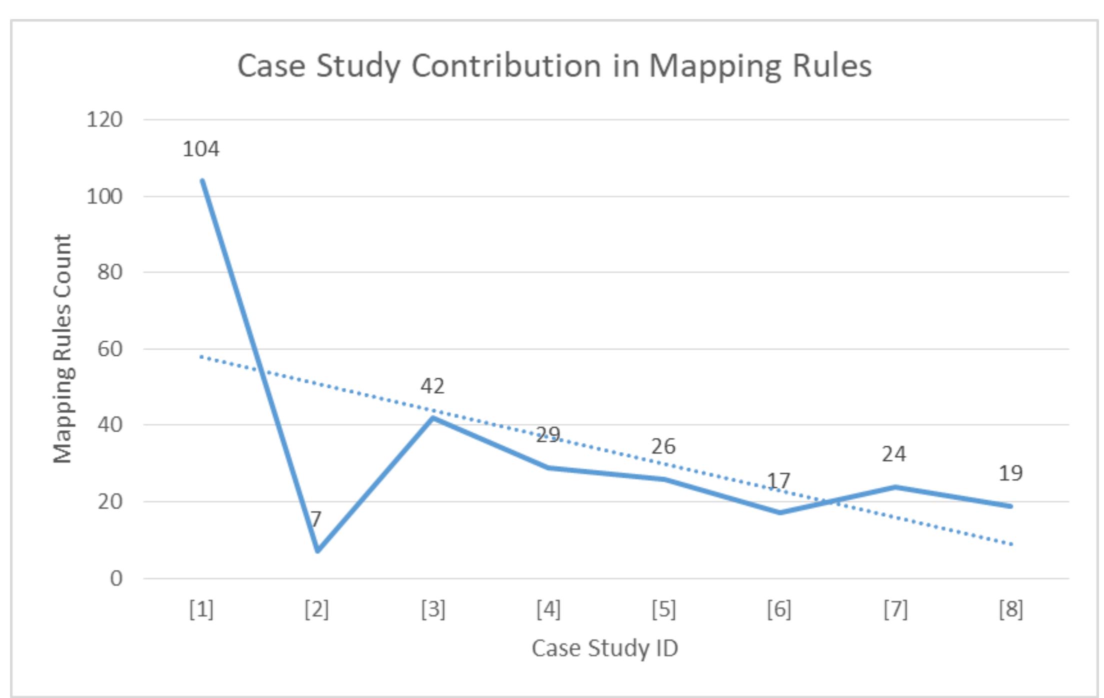


Figure 5-14: Number of new mapping rules extracted from each case study.

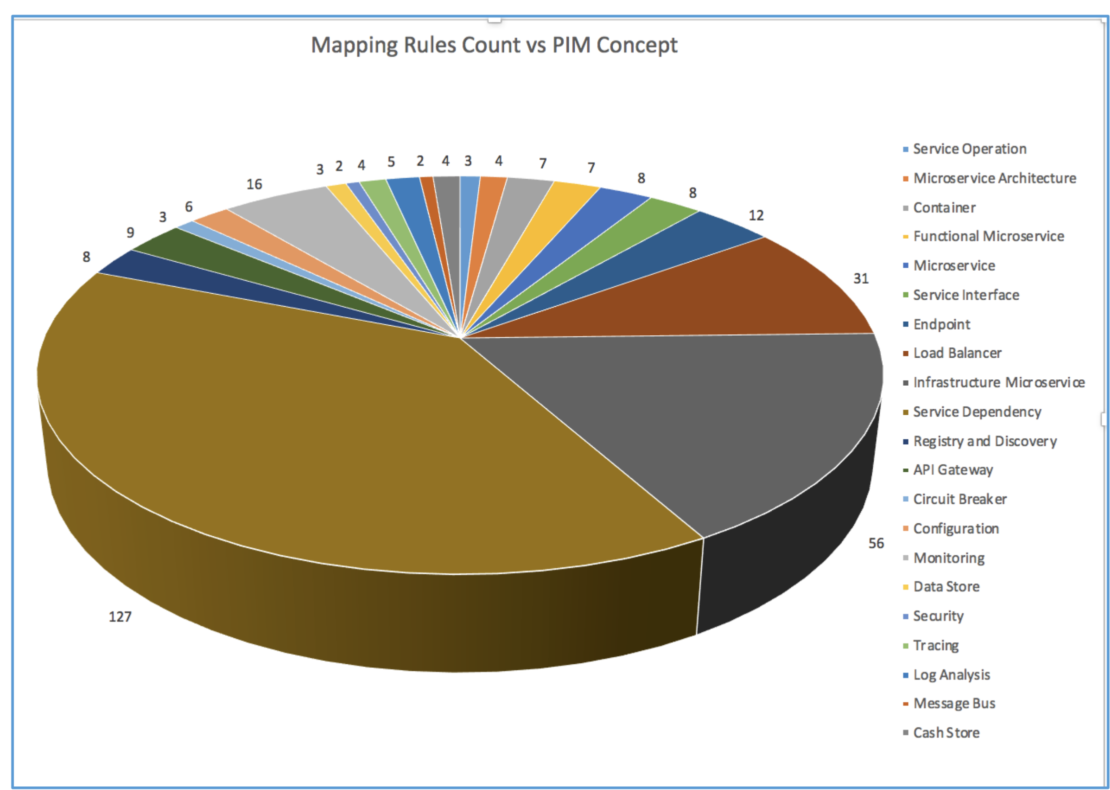


Figure 5-15: Number of mapping rules per concept.

5.4.3.2. MiSAR Mapping Rules: Initial Mapping Rule Artefacts

Mapping rules are the rules that map between architectural concepts and the implementation of these concepts. Once the metamodel concepts that were considered in section 5.4.3.1 have been defined, the remaining steps involve mapping them and their implementation. The purpose of this section is to identify how the architectural concepts are represented in the source code and how they are mapped in its implementation.

Two hundred and sixty-eight mapping rules were identified from the source files, as shown in Appendix A, 5. Each architecture concept was mapped with implementation artefacts. The mapping rules were defined used natural language, which map the PSM to the PIM. These mapping rules are preliminary, informal and may not be used directly to derive an executable operation. For example, a total of nine rules are defined for the **API Gateway** concept, as shown in Table 5-5. Table 5-6 shows all seven mapping rules for **Containerisation** concept.

As can be seen from Table 5-5 and Table 5-6, the mapping rules map between artefact types (or PSM concepts) and architectural elements (or PIM concepts). There are two types of mapping rules; one type is the PIM Concept Identification Rule, which identifies the implementation of corresponding architecture element type, i.e. at PIM concept (source). The other type is the PIM Dependency Identification Rule, which indicates the association between two PIM concepts, source and destination. For example, in Table 5-5 all mapping rules are considered as PIM Concept Identification Rules, since they map to a PIM concept (source). On the other hand, in Table 5-6 all mapping rules identify a dependency from one **Microservice** PIM concept, i.e. source, to another **Container** PIM concept, i.e. destination.

Table 5-5: Mapping rules to identify API Gateway.

Artefact Type	PIM Concept (Source)	Mapping rules
Build File	API Gateway	An API Gateway concept with the technology of 'Netflix Zuul' is indicated by a <project><dependencies><dependency><artifactId> key with the value 'spring-cloud-starter-netflix-zuul' in the Build File of the microservice project.
Build File	API Gateway	An API Gateway concept with the technology of 'Netflix Zuul' is indicated by a <project><dependencies><dependency><artifactId> key with the value 'spring-cloud-starter-zuul' in the Build File of the microservice project.
Build File	API Gateway	An API Gateway concept with the technology of 'Netflix Zuul' is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-zuul' in the Build File of the microservice project.
Build File	API Gateway	A 'Netflix Sidecar' API Gateway is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-netflix-sidecar' in the Build File of the microservice project.
Configurations File	API Gateway	An API Gateway concept with the technology of 'Netflix Zuul' is indicated by the property name that starts with 'zuul.routes.' in the Configurations File of the microservice project.
Configurations File	API Gateway	A 'Netflix Sidecar' API Gateway is indicated by the property 'sidecar.port:' and/or 'sidecar.healthUri:' in the Configurations File of the microservice project.
Source Code File	API Gateway	A 'Netflix Zuul' API Gateway is indicated by a Java Class with the '@EnableZuulProxy' annotation in the Source Code File of the microservice project.
Source Code File	API Gateway	A 'Netflix Sidecar' API Gateway is indicated by a Java Class with the '@EnableSidecar' annotation in the Source Code File of the microservice project.
Container Build File	API Gateway	A 'Apache HTTP' API Gateway concept is indicated by a 'RUN' command with an argument value that contains 'apache2', 'proxy_http' and 'proxy_balancer' in the Container Build File of the microservice project.

Table 5-6: Mapping rules to identify containerisation.

Artefact Type	PIM Concept (Source)	PIM Concept (Destination)	Mapping rules
Build File	Microservice	Container	The name of this Container concept is indicated by the value of the <project><modules><module> key in the Build File of the application project.
Build File	Microservice	Container	The name of this Container concept is indicated by the value of the <project><artifactId> key in the Build File of the microservice project.
Configurations File	Microservice	Container	The name of this Container concept is indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice project.
Source Code File	Microservice	Container	The name of this Container concept is indicated by the last section of the package name of a Java Class with the '@SpringBootApplication' annotation in the Source Code File of the microservice project.
Source Code File	Microservice	Container	The name of this Container concept is indicated by the last section of the package name of a Java Class with a Java Method with the identifier 'main' that invokes another Java Method with the identifier 'SpringApplication.run' in the Source Code File of the microservice project.
Container Build File	Microservice	Container	The name of this Container concept is indicated by the JAR application name argument of the 'ADD' command in the Container Build File of the microservice project.
Container Orchestration File	Microservice	Container	The name of this Container concept is indicated by the key name of the service container definition in the Container Orchestration File of the application project.

- **RQ3: Methods of system analysis**

Mapping rules related to all architecture concepts in the proposed metamodel were extracted using static analysis, but that was mainly possible due to the presence of a container orchestration file (Docker-compose.yml). Without it, the dynamic analysis would have been required in order to inspect the execution context of the microservice architecture including the integration of non-JVM applications and external backing services needed at runtime. These aspects cannot be checked statically as they are sometimes wrapped in Spring annotations and default configurations. Several mapping rules could be identified by using both static and dynamic analysis. For example, the port of a microservice can be identified using the Docker-compose.yml and/or the Dockerfile, and at the same time, this can be confirmed by running software like TCPDump or tracing the requests that the service sends/receives at runtime.

Appendix A, 2 shows extracted information using static analysis and dynamic analysis.

View Architecture Model Recovery

The architecture model for case study 1 (PiggyMetrics) was recovered by applying the following manual steps utilising the functionalities of MS Excel spreadsheets. The steps are divided into Setup, Recovery and Visualisation steps.

a) Setup: the aim of the two setup steps is to prepare the table of mapping rules for the facilitation of executing the recovery steps.

1- A list of all microservices in the PiggyMetrics system is generated by application of mapping rule 4 and rule 97 (a list of mapping rules is given in Appendix A, 4). These generic rules are applicable to most microservice-based systems.

2- For every microservice generated in 1, a new column with the name of the microservice is added to the mapping rule table. To illustrate, a new column of the microservice *registry* is added, as in Table 5-7.

Table 5-7: Mapping rules applied for *registry* microservice in case study 1 (PiggyMetrics).

SEQ	PIM Concept (Source)	PIM Concept (Destination)	Mapping Rule	registry
1	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the name of the root GitHub Repository which contains all artefacts of the application's project.	1
2	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the value of <project><artifactId> key in the Build File of the application's project.	1
3	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the value of <project><parent><artifactId> key in the Build File of the microservice's project.	1
4	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of <project><modules><module> key in the Build File of the application's project.	1
5	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of	1

			<project><artifactId> key in the Build File of the microservice's project.	
6	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice's project.	1
7	Microservice Architecture	Microservice	The name of Microservice concept is indicated by last section of package name of a Java Class with '@SpringBootApplication' annotation in the Source Code File of the microservice's project.	1
8	Microservice Architecture	Microservice	The name of Microservice concept is indicated by last section of package name of a Java Class with Java Method with identifier of 'main' that invockes another Java Method with identifier of 'SpringApplication.run' in the Source Code File of the microservice's project.	1
9	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the JAR application name argument of 'ADD' command in the Container Build File of the microservice's project.	1
10	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
11	Microservice	Container	The name of Container concept is indicated by the value of <project><modules><module> key in the Build File of the application's project.	1
12	Microservice	Container	The name of Container concept is indicated by the value of <project><artifactId> key in the Build File of the microservice's project.	1
13	Microservice	Container	The name of Container concept is indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice's project.	1
14	Microservice	Container	The name of Container concept is indicated by last section of package name of a Java Class with '@SpringBootApplication' annotation in the Source Code File of the microservice's project.	1
15	Microservice	Container	The name of Container concept is indicated by last section of package name of a Java Class with Java Method with identifier of 'main' that invockes another Java Method with identifier of 'SpringApplication.run' in the Source Code File of the microservice's project.	1
16	Microservice	Container	The name of Container concept is indicated by the JAR application name argument of 'ADD' command in the	1

			Container Build File of the microservice's project.	
17	Microservice	Container	The name of Container concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
18	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-server' in the Build File of the microservice's project.	1
19	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.	1
20	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the two properties 'eureka.client.registerWithEureka: false' and 'eureka.client.fetchRegistry: false' in the Configurations File of the microservice's project.	1
21	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'eureka.server.waitTimeInMsWhenSyncEmpty:' in the Configurations File of the microservice's project.	1
22	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a Java Class with '@EnableEurekaServer' annotation in the Source Code File of the microservice's project.	1
23	Microservice	Service Interface	The server path of Service Interface concept is indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice's project.	1
24	Microservice	Service Interface	The server path of Service Interface concept is indicated by last section of package name of a Java Class with '@SpringBootApplication' annotation in the Source Code File of the microservice's project.	1
25	Microservice	Service Interface	The server path of Service Interface concept is indicated by last section of package name of a Java Class with Java Method with identifier of 'main' that invockes another Java Method with identifier of 'SpringApplication.run' in the Source Code File of the microservice's project.	1
26	Microservice	Service Interface	The server path of Service Interface concept is indicated by the JAR application name argument of 'ADD'	1

			command in the Container Build File of the microservice's project.	
27	Microservice	Service Interface	The server path of Service Interface concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
28	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-config' in the Build File of the microservice's project.	1
29	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-client' in the Build File of the microservice's project.	6
30	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by the hostname section of the url value of the property 'spring.cloud.config.uri:' or 'spring.cloud.config.failFast: true' in the Configurations File of the microservice's project.	1
31	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by the hostname section of the URL value of the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.	6
32	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by the hostname section of the url value of the property 'security.oauth2.resource.userInfoUri:' or 'security.oauth2.client.accessTokenUri:' in the Configurations File of the microservice's project.	1
33	Microservice	Service Dependency	A 'RabbitMQ' Infrastructure Microservice provider to a Microservice is indicated by the value of the property 'spring.rabbitmq.host:' in the Configurations File of the microservice's project.	1
34	Microservice	Service Dependency	A Registry and Discovery provider to a Microservice is indicated by a Java Class with '@EnableDiscoveryClient' annotation in the Source Code File of the microservice's project.	6
35	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the service container name of 'depends_on' or 'links' key in the Container Orchestration File of the application's project.	1

36	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-server' in the Build File of the microservice's project.	1
37	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by the two properties 'eureka.client.registerWithEureka: false' and 'eureka.client.fetchRegistry: false' in the Configurations File of the microservice's project.	1
38	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by the property 'eureka.server.waitTimeInMsWhenSyncEmpty:' in the Configurations File of the microservice's project.	1
39	Registry and Discovery	-	A Registry and Discovery concept with the technology of 'Netflix Eureka' is indicated by a Java Class with '@EnableEurekaServer' annotation in the Source Code File of the microservice's project.	1
40	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-client' in the Build File of the microservice's project.	6
41	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.	6
42	Registry and Discovery	Service Dependency	A Microservice provider to a Registry and Discovery is indicated by a Java Class with '@EnableDiscoveryClient' annotation in the Source Code File of the microservice's project.	6
43	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by the non-zero property 'hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:' or the property 'feign.hystrix.enabled: true' in the Configurations File of the microservice's project.	1
44	Log Analysis	Service Dependency	A Microservice provider to a Log Analysis is indicated by the property name that starts with 'logging.level.' in the Configurations File of the microservice's project.	1

45	Log Analysis	Service Dependency	A Microservice provider to a Log Analysis is indicated by the key name of service container definition that has 'logging' or 'log_opt' key in the Container Orchestration File of the application's project.	1
46	Service Operation	Circuit Breaker	A 'Netflix Hystrix' Circuit Breaker to Service Operation is indicated by the non-zero property 'hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:' or the property 'feign.hystrix.enabled: true' in the Configurations File of the microservice's project.	1

b) Recovery: The following steps were conducted in a microservice-wise manner. To illustrate some steps, the *registry* microservice is considered here.

1- The entire mapping rule table is sorted by the *Artefact Type* column in the following order of values (this list is inserted into Excel as the basis for the sorting):

- *GitHub Repository*
- *Container Orchestration File*
- *Container Build File*
- *Build File*
- *Configurations File*
- *Source Code File*

This order facilitates the recovery since it is equivalent to the depth level of each artefact in the project's directory/file tree. To illustrate, the build file of a project is located just under the root directory, while the source files are deep inside.

2- The artefact corresponding to each type is checked and the group of mapping rules related to this particular artefact type is manually examined in a rule-wise manner.

3- If a rule applies ***n*** times, then the non-zero value of ***n*** is inserted in the cell located at the rule-microservice intersection, otherwise the value **0** is inserted. To illustrate, rules 29, 31, 34 and 40 to 42 were applied to the *registry*

microservice six times (see Table 5-7) because, as a service registry and discovery server, the URL of the *registry* microservice is configured for six client microservices in order to register themselves, and the registry microservice itself needs to frequently request the */health* of its six clients to check their running state.

4- To facilitate the subsequent visualisation steps, the mapping rules table is re-sorted by two columns: *PIM Concept (Source)* then *PIM Concept (Destination)*, in the following order of values (these lists are inserted into Excel as the basis for the sorting):

PIM Concept (Source) Sorting

- *Microservice Architecture*
- *Container*
- *Microservice*
- *Functional Microservice*
- *Infrastructure Microservice*
- *API Gateway*
- *Configuration*
- *Registry and Discovery*
- *Security*
- *Monitoring*
- *Tracing*
- *Log Analysis*
- *Service Interface*
- *Service Operation*

PIM Concept (Destination) Sorting

- *(Blank)*
- *Microservice*
- *Container*
- *Load Balancer*
- *Service Interface*
- *Endpoint*
- *Service Operation*
- *Data Store*
- *Cash Store*
- *Circuit Breaker*
- *Message Bus*
- *Service Dependency*

This order is equivalent to the depth level of each concept in the PIM metamodel illustrated in Figure 5-13. To illustrate, the *Microservice* and its infrastructure subtypes are listed after the *Microservice Architecture* and before the *Service Interface*.

5- The last step is to filter the current mapping rules table by non-zero values (the **0** value is simply unchecked). Table 5-7 is the results table for the example *registry* microservice after filtering.

c) Visualisation: The final recovered model is visualised in two diagrams: the instance diagram and the architecture diagram. The instance diagram is equivalent to a UML object diagram that conforms to the PIM metamodel in Figure 5-13.

In developing the architecture-level diagram, I was inspired by Sam Newman's (2019) graphical notations for representing high-level abstraction; for example, he uses a hexagon to represent the service and a link to represent the association between two services, as outlined in Appendix B-1.

The following steps are a continuation of the Recovery steps. They are conducted for the particular microservice being recovered. To illustrate some steps, the *registry* microservice is considered here.

- **6-** Draw every concept appearing in the *PIM Concept (Source)* column and connect it to the concept appearing next to it in the *PIM Concept (Destination)* column, except for the *Microservice Architecture* and *Service Dependency* concepts because they require references to already recovered microservices. The instance diagram that includes the *registry* microservice only after this step is provided in Figure 5-16. The objects with the dashed line are the objects to be added later after completion of recovery for all concepts in all microservices, while an object with a red outline indicates an issue in the recovery. This issue is discussed in Chapter 6.
- **7-** Repeat the recovery steps (steps 1 to 5) and visualisation process (step 6) for all the remaining microservices in the list generated in setup step 1.
- **8-** End the visualisation by drawing the *Microservice Architecture* and *Service Dependency* concepts needed for the entire architecture.

Output: The output of this stage is the complete instance diagram and architecture diagram (see Appendix B-2) for the recovered model diagram of case study 1. More elaboration of the manual recovery and visualisation steps is provided in Appendix B-3.

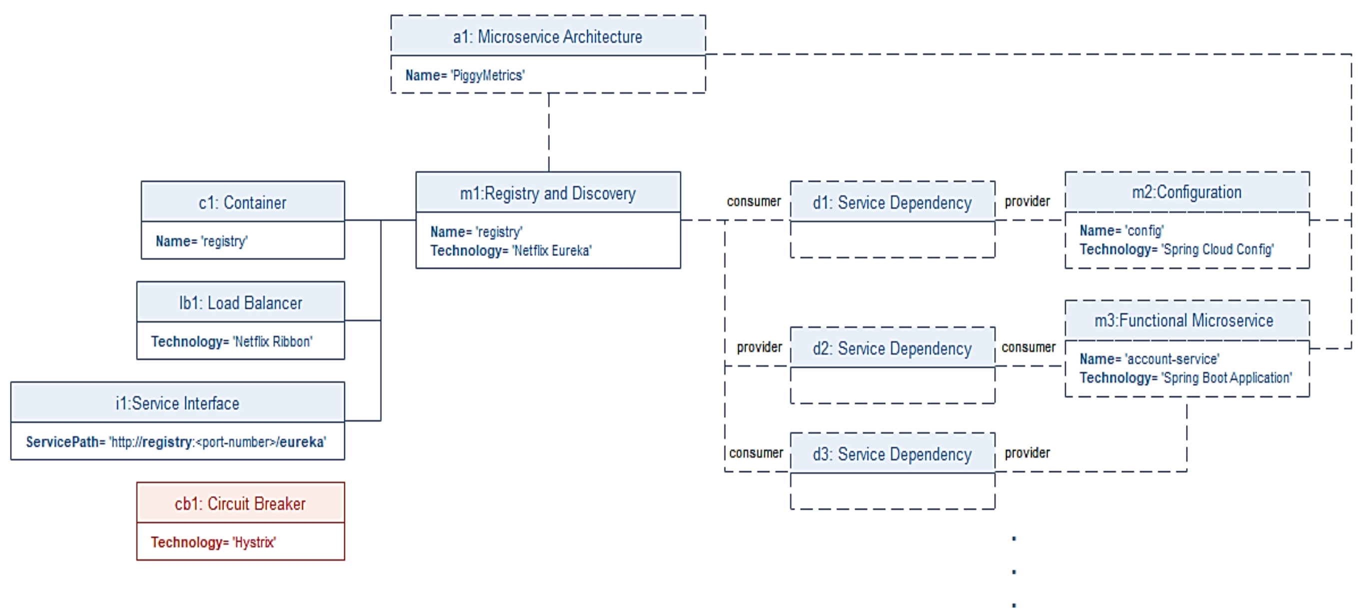


Figure 5-14: The partial instance diagram of case study 1 after the recovery of the registry microservice.

5.5. Summary

This chapter presents the metamodel and mapping rules of the MiSAR approach, which are artefacts that are used to recover the architectures of microservice systems. The metamodel abstracts the concepts of a microservice architecture in a technology-independent manner, and the mapping rules map an implemented microservice-based system into an architectural model which instantiates the metamodel. These two key components of MiSAR offer a well-defined procedure for recovering the architecture of a microservice-based system. To be able to define these MDE artefacts, a study was designed and conducted which included a manual and iterative recovery process. By conducting this study, the approach has considered the key architectural concepts encountered in microservice systems and their mapping rules. In the next chapter, MiSAR repository of metamodel and mapping rules is to be validated further and extended by the analysis of new complex case studies.

Chapter 6

Improving the Initial Artefacts of MiSAR: An Empirical Study

6.1. Introduction

In Chapter 5, I conducted an empirical study to define an initial version of the MiSAR artefacts: the metamodel, which supports the creation of microservice architectural models, and mapping rules, which map microservice source systems into the metamodel by analysing microservice software systems and extracting and clustering architectural concepts. This chapter presents a new empirical study which uses the initial MiSAR artefacts to evaluate and enhance themselves. For this chapter, the aim was to define the final MiSAR artefacts in order to be able to generate architectural models of implemented microservice systems. To achieve this, I designed an empirical study which manually applies the initial MiSAR artefacts to a set of open microservice projects, which are implemented in the Java, Docker and Spring Boot/ Cloud frameworks. The focus is on validating and enhancing MiSAR artefacts incrementally and achieving improved artefacts. I then used the improved artefacts to recover the architectural model of a system.

6.2. MiSAR Abstraction Levels

MiSAR considers elements at three different abstraction levels. As described in Figure 6-1, the L0 level includes the microservice software system in the real world as a set of physical artefacts. It currently considers source code, configuration, Docker, Docker Compose, build and project build files. The L1 level, also known as the Platform-Specific Model (PSM), represents the software artefacts of L0 in more abstract models, which conform to their metamodel, and supports the technology of the implemented microservice system. The L2 level represents the Platform-Independent Model (PIM), which abstracts the concepts of microservice architecture in a technology-independent way. Mapping rules are needed to map an implemented microservice-based system into an architectural model by instantiating the PIM. The

PIM metamodel is presented in Chapter 5, Figure 5-13. In this chapter, I aimed to enrich the existing artefacts of MiSAR with the objective of allowing the latter to recover more expressive architectural models.

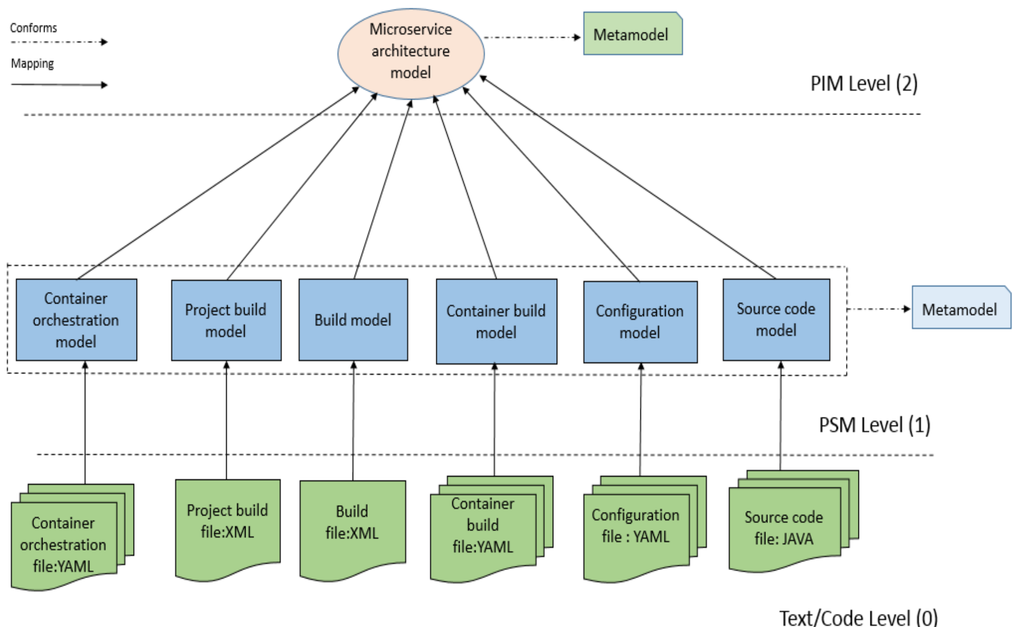


Figure 6-1: MiSAR abstraction levels.

6.3. Identification of PSM metamodel

The PSM layer of microservice applications in this study is composed of six artefacts, as illustrated in Figure 6-1: the XML files containing information about the project, the configuration details used by Maven or Gradle to build the project, the YAML files describing system runtime configuration values, and the Java files that specify the different variable elements of a service. However, a natural question to ask is "*What are the PSM metamodels of information that needs to be extracted from source artefacts?*".

In order to represent the models at the PSM layer, the analysis started early in study 1 (Chapter 5), at the stage of creating the initial mapping rules. In study 1, I noticed patterns in the textual representation of mapping rules, especially in the phrase that describes keywords in the artefacts that begin the transformation, which enabled me

to extract a common structure that contains a **variable/attribute** in the source artefact (e.g. the <dependency> element in the POM file or a particular class-level annotation in the Java source file), followed by a **literal value** (e.g. the dependency library's <artifactId> value or the annotation's name). This common structure facilitated the design of a PSM metamodel to reflect the key variables/attributes and values extracted from the artefact elements. I performed the five steps represented in Figure 6-2 for the identification of the PSM metamodel.

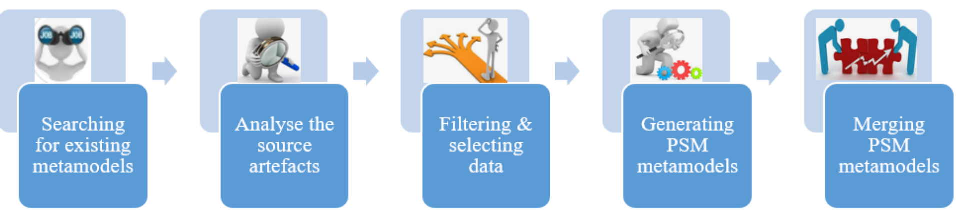


Figure 6-2: The PSM metamodel identification steps.

- **Step 1- Searching for existing metamodel**: For the purposes of this study, I used my own simplified Java metamodel inspired by Paige (2006) and (Dirckze, 2002) rather than the standard fine-grained Java syntax schema. As for other artefacts, I did not find any existing metamodels, which raised the need to identify them from scratch.

- **Step 2- In-depth analysis of source artefacts**: I analysed the artefacts manually for all eight systems listed in Chapter 5 in a depth-first manner, left-to-right order, starting at system project artefacts, followed by microservice project artefacts, as appears in the PSM model tree (Figure 6-3).

- **Step 3- Extracting and selecting a subset of extracted information**: For the artefacts analysed in Figure 6-3, I extracted information. Data selection and filtering which describes the static aspects is necessary to select which data to include in the PSM instance. This selection approach is considered partial coverage as I selected a particular set of variables that is meaningful, identified through the analysis phase, into a separate file to provide the most relevant information for architecture recovery provided in Appendix A, 1 and 2.

- **Step 4- Generating PSM metamodels for each artefact**: The PSM metamodels are obtained from source artefacts. Each artefact element is modelled, building up the element at the PIM architecture level.

- **Step 5- Merging PSM metamodels**: All PSM metamodels are integrated into one metamodel.

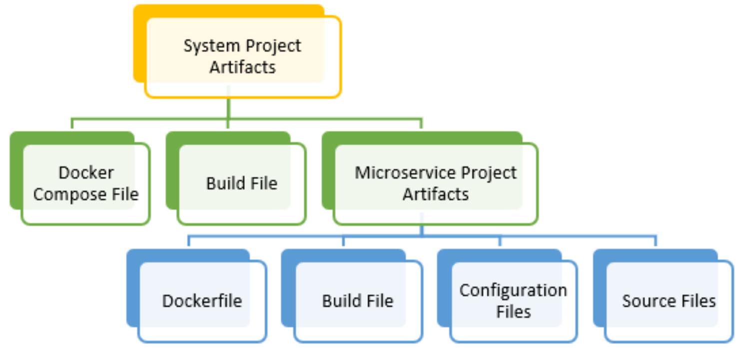


Figure 6-3: PSM model tree.

Figures 6-4 and 6-5 present the PSM metamodel for this study. **DistrubutedApplicationProject** captures the architecture's development artefacts includes multi-module project (**ApplicationProject)** and module projects (**MicroserviceProject**) as well as the runtime artefacts (**Docker containers**). **DistrubutedApplicationProject** is described by application name and its root repository URI. The runtime artefacts are simply the collection of **DockerContainerDefinition** elements involved in the architecture and defined in the Docker Compose as well as Dockerfile files. Each **DockerContainerDefinition** is described by container name, build path, image name, and whether it generates log or not. The build path denotes the path of the module project if the artefacts are locally available; otherwise, the image name denotes the artefacts at the remote Docker Hub. The **DockerContainerPort** and **DockerContainerLink** runtime information are also captured for each Docker container.

The development artefacts are generally represented by the **ApplicationProject** element, which is equivalent to a multi-module project along with its module projects, each represented by a **MicroserviceProject** element. The **MicroserviceProject** element generalises a wide range of project artefacts implemented in any frameworks

or languages, including Java Spring Boot/Cloud. Each **MicroserviceProject** defines a collection of **DependencyLibrary** elements that can be found in project build artefact such as Maven POM.XML or Gradle BUILD.GRADLE. The **DependencyLibrary** elements in a Maven POM.XML file match the '<dependency></dependency>' XML elements, while these match the argument's 'compile' commands in a Gradle BUILD.GRADLE file. They simply list the project's software dependencies which are described by group, library name and scope.

The **JavaSpringWebApplicationProject** element is a subtype of the **MicroserviceProject** element which reflects the specific characteristics of applications built with the Spring Boot/Cloud framework. One such characteristic is that each **JavaSpringWebApplicationProject** defines a collection of settings in YAML or PROPERTIES artefacts. These settings are represented with the **ConfigurationProperty** element, which defines important functionality and execution information. Each **ConfigurationProperty** has a name, value and particular scope (configuration profile).

The **JavaSpringWebApplicationProject** element is extended further to be either a **JavaSpringMVCApplicationProject** or **JavaSpringWebFluxApplicationProject**. The second subtype was recently adopted by the Spring Boot/Cloud team to facilitate building reactive non-blocking Web applications. This recent paradigm led to slight changes in the terms and libraries used for implementation, which have to be reflected in MiSAR's PSM metamodel.

Another characteristic of **JavaSpringWebApplicationProject** is that it aggregates multiple Java classes and/or Java interfaces with a means of annotation into **JavaSpringWebApplicationLayer**s. Each layer represents a specific role in the application. To illustrate, a Java class annotated with '@SpringBootApplication' represent the application's start-up class, while Java classes annotated with '@Controller' represent the declaration and definition of Web services exposed by the application.

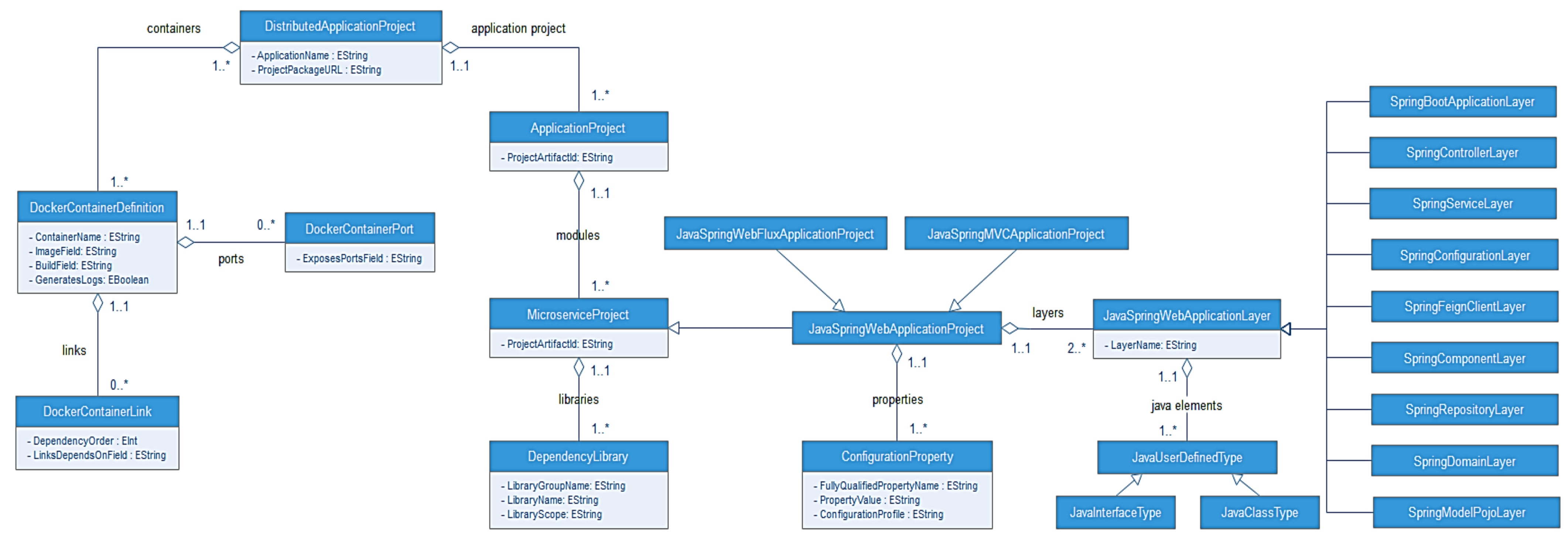


Figure 6-4: PSM metamodel (Java PSM metamodel is reduced).

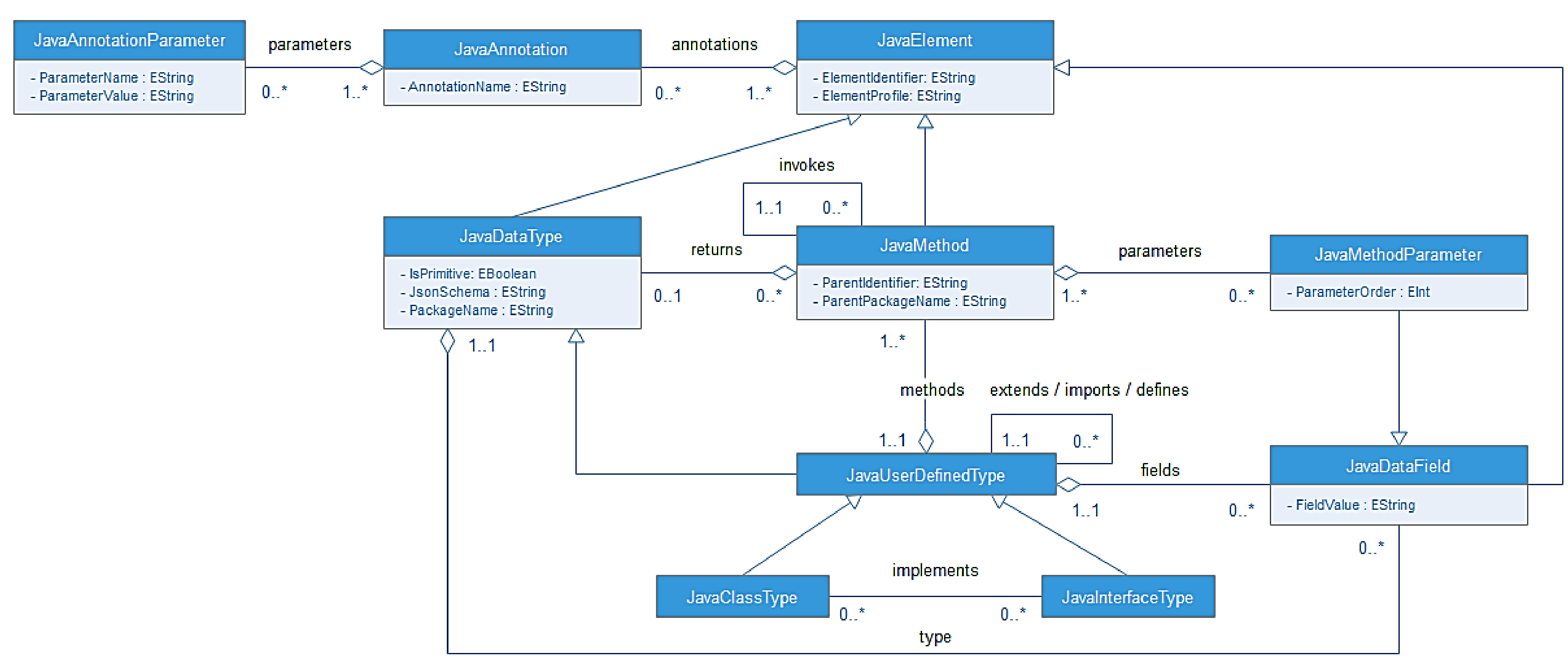


Figure 6-5: Java PSM metamodel.

Each **JavaUserDefinedType** maps to a Java source file created for a microservice Spring Boot application. As noted earlier, and for the purposes of this study, I used my own simplified Java metamodel, illustrated in Figure 6-5. For example, Java control flow statements (if-then and for-loop statements), as well as expressions (arithmetic and logical), are not considered. In addition, class definitions, declarations of local Java fields and method invocations are elaborated no further than needed for this study.

Basically, a Java file is defined as containing many **JavaElement**s; each could either be a **JavaDataType** or **JavaMethod**, as shown in Figure 6-5. A **JavaElement** is defined by its name, i.e. identifier, and execution profile. Spring framework defines the role of each **JavaElement** by using its set of tailored **JavaAnnotations**, which, in turn, may have one or many **JavaAnnotationParameter**s. **JavaDataType** is either primitive (e.g. int, float, char or boolean) or a **JavaUserDefinedType**, which, in turn, can be a **JavaInterfaceType** or a **JavaClassType**. A **JavaDataType** is also the type of any **JavaDataField** element and is defined by JsonSchema, i.e. a JSON representation of type definition, and PackageName. A **JavaUserDefinedType** may extend other **JavaUserDefinedType** elements and contain one or many **JavaMethod**s. A **JavaClassType** can implement a **JavaInterfaceType** and may contain local **JavaDataField** elements in addition to **JavaMethod**s. A **JavaMethod** may take instances of **JavaMethodParameter** as input parameters in its signature and/or as local fields in its body. It may return at most one instance of **JavaDataType** and can invoke other **JavaMethod**s.

6.4. Study Design

6.4.1. Study Aim and Research Questions

This study aims to validate that the existing artefacts of MiSAR can recover an architectural model and to enhance these artefacts if required. To achieve this, I defined three research questions:

RQ1*: What are the enhancements that have to be performed to the existing MiSAR metamodel to represent more richly recovered architectural models of microservice systems?*
RQ2*: What enhancements have to be applied to the current MiSAR mapping rules that map microservice Java and Spring Cloud systems into architectural models?*
RQ3*: Can an enhanced MiSAR approach recover architectural models?*

RQs 1 and 2 focus on enhancement and refinement of the MiSAR elements that were proposed in Chapter 5, based on analysis of microservice systems. RQ3 focuses on the integration of all MiSAR artefacts and applies them to the obtaining of architectural models at the PIM level. To evaluate the recovered architectural model, it is compared with the documentation provided with a particular system.

6.4.2. Selecting the Case Studies

After comprehensive surfing on the GitHub repository, to answer RQ1 and RQ2 I chose nine open-source systems, listed in Table 6-1, as the aim in this study is to address more complex systems which have more services than the ones selected in the previous study (Chapter 5). Specific criteria were applied to support project relevance, as stated in Chapter 5, Table 5-2. Systems were selected that (a) employed microservice architecture (b) are built using the Java Spring framework; (c) use Netflix OSS technologies;[13] and in this study, in addition to the previous points, I considered systems that (d) implement synchronous/asynchronous inter-microservice interaction style; and (e) integrate variation implementations style. To answer RQ3, I selected a new system called Microservices Sample (Vijayendra, 2019) for validation purposes, which is an open-source microservice project. The criteria for the selection of this case study concentrated mainly on (a) the implementation of both synchronous and asynchronous inter-microservice communication styles (i.e. service dependencies), (b) the integration of polyglot technologies (e.g. multiple datastores, reactive programming, etc.), (c) multiple configuration profiles, (d) the availability of its architecture documentation and supporting diagrams with illustrations, in order to

[13] Netflix OSS is a set of frameworks and libraries that Netflix wrote to resolve some issues concerning distributed systems at scale.

ensure that the validity and effectiveness of all enhancements made to MiSAR in this study are evaluated. The selected case study meeting these criteria was the Microservices Sample application, which consists of 14 microservices.

6.4.3. Research Design

The following describes the design of the study, which is divided into four activities, as depicted in Figure 6-6. Activities 1 and 2 both include manual recovery and are always performed in parallel to enhance and refine MiSAR in increments. Activity 3 includes implementing the MiSAR artefacts and activity 4 includes applying the implemented artefacts to recover an architectural model represented in a diagram of a system.

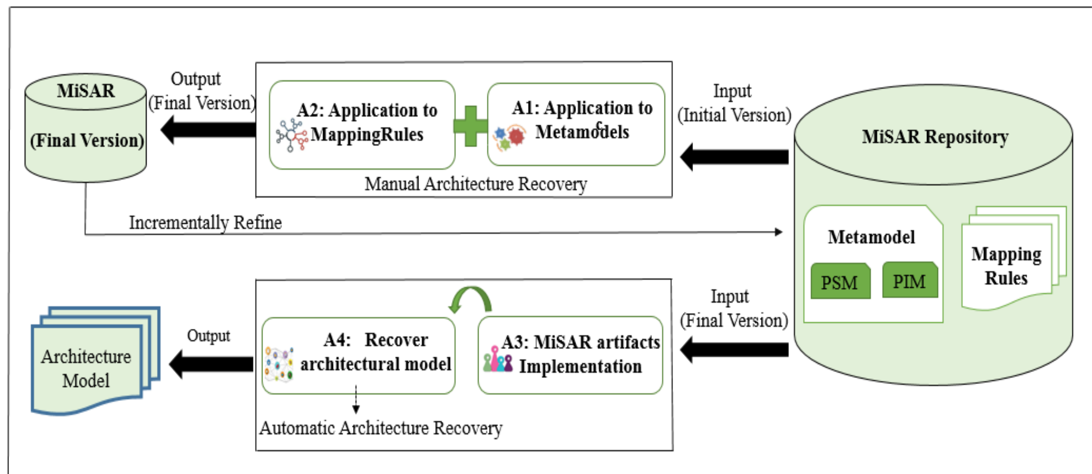


Figure 6-6: Empirical process for enhancing and refining MiSAR.

Activity 1 – Application to metamodels: This activity begins by applying the metamodel presented in Chapter 5 (see Figure 5-13, version 1) to every system in Table 6-1. As a result of this, I incrementally updated and refined the metamodel after analysing the selected case studies. An 'increment' is a change in the metamodel design. The objective of this task is to verify the validity of these metamodel concepts with the new systems that are being analysed.

Activity 2 – Application to mapping rules: The nine systems' implementation was analysed manually in order to refine and enhance the MiSAR repository's mapping rules. In the previous study (Chapter 5), mapping rules were defined using natural

language, informal descriptions and non-executables. In this activity, I wanted to permit existing mapping rules to include more detailed descriptions and to add new ones. For each system analysed, a new mapping rule was added to the MiSAR repository, and for each new metamodel concept added, a mapping rule was added to map this concept as well.

Activity 3 – MiSAR artefacts implementation: After analysing all nine systems, refining and enhancing the MiSAR repository in activities 1 and 2, the mapping rules were implemented using the QVTo Operational QVT transformation language (Barendrecht, 2010), using Eclipse M2M. Metamodels were implemented as Ecore models using the Eclipse Modeling Framework (EMF) (Dave et al., 2008); the details of this activity are presented in Chapter 7.

Activity 4 – Recover architectural model: The ultimate goal of this activity is to check the validity of the mapping rules to create an architectural model that conforms to the PIM. In this activity, the implemented MiSAR artefacts (mapping rules and metamodels) are applied in order to ultimately develop the architecture diagram for validation purposes. The recovery was applied with a different system to those systems listed in Table 6-1. The recovery process approach and its validation can be summarized in the six-step process (presented in section 6-5), which can be attempted at different levels of abstraction in order to extract the architectural model.

Table 6-1: Selected systems

ID	Project Name	Project Repository URL	Microservice Count	LOC[14] Size	No. of Developers	Project Timeline	Documentation	Architecture Diagram
1	spring-netflix-oss-microservices	https://github.com/fernandoabcampos/spring-netflix-oss-microservices	9	714	1	Mar 13, 2016 – Mar 20, 2020	Not Available	Not Available
2	spring-rabbitmq-messaging-microservices	https://github.com/jonashackt/spring-rabbitmq-messaging-microservices	7	932	2	Nov 4, 2018 – Mar 2 , 2020	Available	Available
3	cloud-enabled-microservice	https://github.com/sergeikh/cloud-enabled-microservice	7	1609	1	Mar 5, 2017 – Mar 2, 2020	Available	Not available
4	event-sourcing-microservices-example	https://github.com/kbastani/event-sourcing-microservices-example	10	3483	5	Oct 22, 2017 – Mar 2, 2020	Available	Available
5	spring-cloud-sidecar-polygot	https://github.com/BarathArivazhagan/spring-cloud-sidecar-polygot	7	305	3	Aug 20, 2017 – Mar 2, 2020	Available	Available
6	microservices-basics-spring-boot	https://github.com/anilallewar/microservices-basics-spring-boot	10	2581	3	Apr 23, 2017 – Mar 2, 2020	Available	Available
7	spring-cloud-event-sourcing-example	https://github.com/kbastani/spring-cloud-event-sourcing-example	15	6777	4	Mar 20, 2016 – Mar 2, 2020	Available	Available
8	spring-boot-graph-processing-example	https://github.com/kbastani/spring-boot-graph-processing-example	9	1279	3	Dec 13, 2015 – Mar 2, 2020	Available	Available
9	BookStoreApp-Distributed-Application	https://github.com/devdcores/BookStoreApp-Distributed-Application	14	5291	1	May 12, 2019 – Mar 2, 2020	Available	Available

[14] Line of code.

6.5. Results

This section presents the analysis and results of the empirical study, according to our research questions. As stated in section 6.4.3, activities 1 and 2 were performed in parallel, but they are separated here for presentation purposes.

- **RQ1:** *What enhancements are required to the MiSAR metamodel to represent more richly recovered architectural models of microservice systems?*

This section presents the analysis of the architectural concepts empirically derived from the nine systems in a number of enhancement increments that were applied to the initial version of the MiSAR PIM metamodel presented in Chapter 5 (see Figure 5-13). The following outlines how version 1 was incrementally enhanced to create an updated metamodel which can help to represent recovered architectural models of microservice systems more accurately. The results of these increments present new requirements that need to be fulfilled as enhancements to the MiSAR PIM metamodel (version 1, as in Figure 5-13). These enhancements led to the final version of the MiSAR metamodel.

Increment 1: Supporting Components of Microservice Patterns

Context-1: During the process of manual recovery, the association from the Service Operation concept towards the Data Store, Cash Store, Circuit Breaker and Asynchronous Message Bus concepts in the PIM metamodel (version 1) could not be recovered for many Infrastructure Microservices and these destination concepts were recovered disconnected (such as the Circuit Breaker instance in Figure 5-14 and Figure 6-7). This is because the Service Operations of Infrastructure Microservices are not explicitly implemented in the source artefacts.

For illustration purposes, I will show how I manually recovered the instance diagram of the edge-service microservice in case study 7 (mentioned in Table 6-1). By applying the mapping rules (see Table 6-2), I could manually recover that edge-service (m5) is an instance of an API Gateway concept (applying rules 7-9 in Table 6-2) as shown in

Figure 6-7, which is a subtype of Infrastructure Microservice according to metamodel (version 1). Also, a Circuit Breaker element is recovered (applying rule 11 in Table 6-2). However, it is noticeable that the Service Operation concept was not recovered at all from the edge-service Java source artefacts, and that Circuit Breaker object (cb1) in Figure 6-7 is recovered disconnected with no associations at all. The reason is that infrastructure providers, such as edge-service microservice, when implemented with Spring Boot/Cloud framework, do not need to explicitly implement any Service Operations in their source artefacts. This discussion leads to the following requirement:

Requirement-1.1➔ Infrastructure components such as Circuit Breaker, Data Store, Cash Store and asynchronous Message Bus concepts need to be directly associated with Microservices in order not to result in disconnected instances.

Enhancement-1.1➔ Reposition the association of the Data Store, Cash Store, Circuit Breaker and asynchronous Message Bus concepts from Service Operation to Microservice instead.

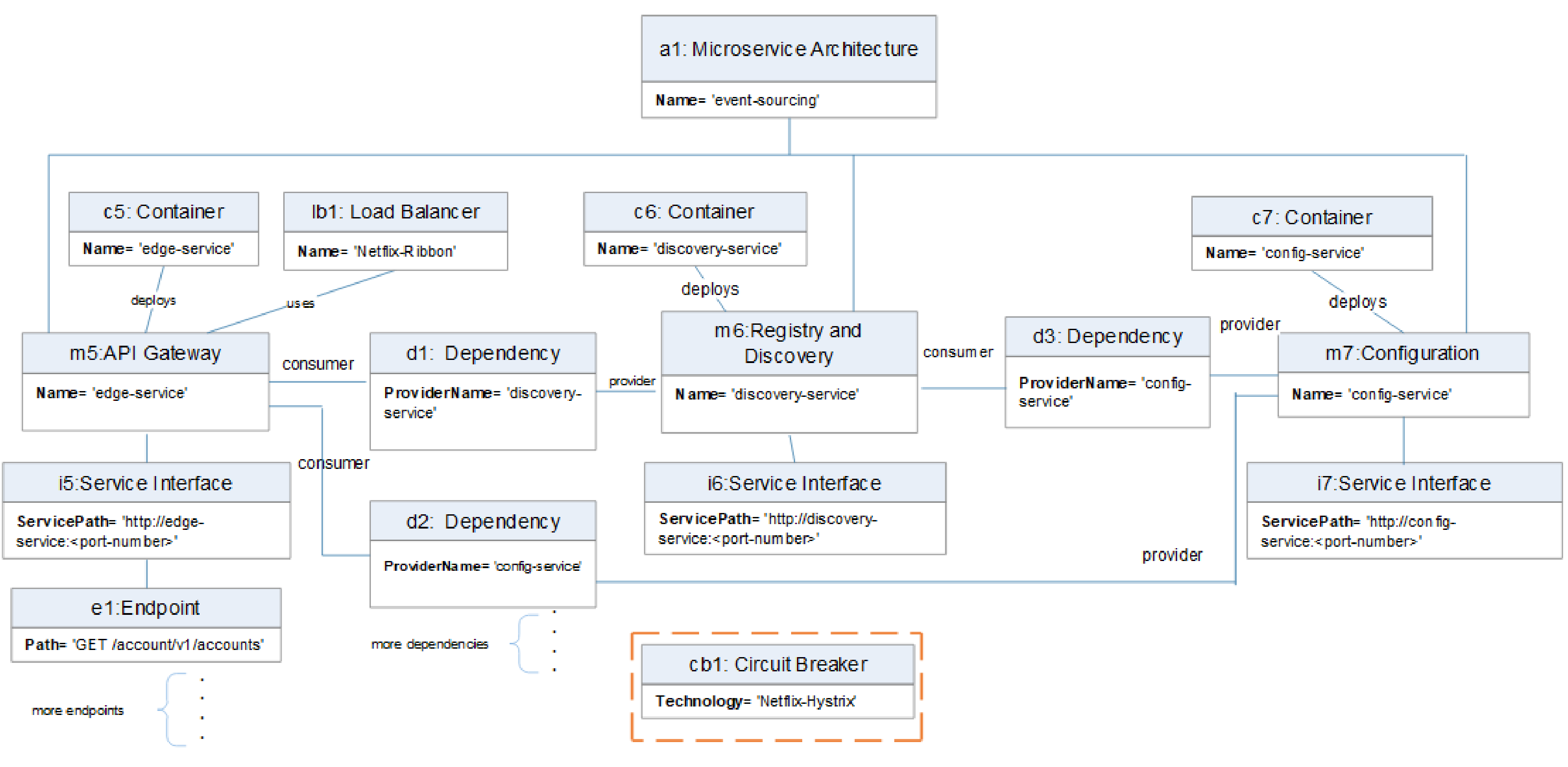


Figure 6-7: PIM instance recovered for edge-service from case study 7 using PIM metamodel (Version 1).

Table 6-2: Mapping rules applied in edge-service in case study 7

SEQ	PIM Concept (Source)	PIM Concept (Destination)	Mapping Rule	edge-service
1	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-config' in the Build File of the microservice's project.	1
2	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-eureka' in the Build File of the microservice's project.	1
3	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by the hostname section of the url value of the property 'spring.cloud.config.uri:' or 'spring.cloud.config.failFast: true' in the Configurations File of the microservice's project.	1
4	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by the hostname section of the url value of the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.	1
5	Microservice	Service Dependency	A Registry and Discovery provider to a Microservice is indicated by a Java Class with '@EnableEurekaClient' annotation in the Source Code File of the microservice's project.	1
6	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the service container name of 'depends_on' or 'links' key in the Container Orchestration File of the application's project.	3
7	API Gateway	-	An API Gateway concept with technology of 'Netflix Zuul' is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-zuul' in the Build File of the microservice's project.	1
8	API Gateway	-	An API Gateway concept with technology of 'Netflix Zuul' is indicated by the property name that starts with 'zuul.routes.' in the Configurations File of the microservice's project.	1
9	API Gateway	-	A 'Netflix Zuul' API Gateway is indicated by a Java Class with '@EnableZuulProxy' annotation in the Source Code File of the microservice's project.	1
10	Service Interface	Endpoint	An Endpoint to Service Interface is indicated by the value of the property that starts with 'zuul.routes.' and ends with the microservice name in the Configurations File of the microservice's project.	7 * N[15]
11	Service Operation	Circuit Breaker	A 'Netflix Hystrix' Circuit Breaker to Service Operation is indicated by the non-zero property 'hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:' or the property 'feign.hystrix.enabled: true' in the Configurations File of the microservice's project.	1

[15] Number of endpoints in each provider microservice.

Context-2: In the initial design of the PIM metamodel (version 1), each infrastructure provider, e.g. API Gateway, Configuration, Discovery and Registry, Security, Log Analysis, Monitoring, and Tracing, is represented as an independent Infrastructure Microservice. For example, it was assumed that an instance of the Discovery and Registry microservice provides only the service registry and discovery functionality. However, this representation encountered certain limitations. The current PIM metamodel (version 1) allows for only one subtype of Infrastructure Microservice at a time. With regard to this issue, the bookstore-consul-discovery microservice from case study 9 is manually recovered as either an instance of Discovery and Registry concept or Configuration concept, subtypes of the Infrastructure Microservice (applying rules 6 and 7 in Table 6-3). However, being an image of Consul Agent provided by HashiCorp[16] the bookstore-consul-discovery microservice provides multiple infrastructure patterns all at once and out-of-the-box, including Configuration and Registry and Discovery.

Requirement-1.2➔ One Infrastructure Microservice can support multiple-infrastructure patterns.

Enhancement-1.2➔ A new Infrastructure Pattern Component concept is introduced. A microservice can aggregate zero to many Infrastructure Pattern Components. A pattern component refers to an architectural element that supports the functionality of a pattern. Infrastructure pattern components have more specific categories. All subtypes of Infrastructure Microservice types in metamodel version 1 become instances of a new enumeration type named Infrastructure Pattern Category, defining the category of one Infrastructure Pattern Component instance.

Table 6-3: Mapping rules applied in bookstore-consul-discovery in case study 9

SEQ	PIM Concept (Source)	PIM Concept (Destination)	Mapping Rule	bookstore-consul-discovery
1	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the name of the root GitHub Repository which contains all artifacts of the application's project.	1
2	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the key name of service container definition in the	1

[16] https://cloud.spring.io/spring-cloud-static/spring-cloud-consul/2.2.0.M1/.

			Container Orchestration File of the application's project.	
3	Microservice	Container	The name of Container concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
4	Microservice	Service Interface	The server path of Service Interface concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
5	Infrastructure Microservice	-	An Infrastructure Microservice concept is indicated by a service container definition that does not have 'build' key in the Container Orchestration File of the application's project.	1
6	Configuration	-	A 'Consul' Configuration concept is indicated by an 'image:' key with value that starts wth 'consul:' in the Container Orchestration File of the application's project.	1
7	Registry and Discovery	-	A 'Consul' Registry and Discovery concept is indicated by an 'image:' key with value that starts wth 'consul:' in the Container Orchestration File of the application's project.	1

Context-3: In the PIM metamodel version 1, it is not straightforward to determine the type of infrastructure pattern requested by a consumer microservice involved in a Service Dependency with an infrastructure provider. Hence, this design requires some improvements. To illustrate, let us look at the edge-service microservice from case study 7, as depicted in Figure 6-7. By applying the mapping rules 2, 4, 5 and 6 in Table 6-2, edge-service (m5) is an instance of the APIGateway microservice, which has a Service Dependency (d1) associating it to a Registry and Discovery instance named discovery-service (m6). The (m6) acts as a Registry and Discovery provider to edge-service (m5), since the edge-service uses this pattern to register its address. The edge-service, by applying rules 1, 3 and 6 in Table 6-2, has another Service Dependency instance (d2) associated with config-server (see Figure 6-7). The (m7) is considered a Configuration provider since edge-service (m5) needs this pattern to pull its centralized configuration properties. The PIM (version 1) does not express literally that edge-service microservice is using both infrastructure patterns: Registry and Discovery and Configuration. Enhanced PIM instances are provided in Figure 6-8.

Requirement-1.3➔ The metamodel design needs to clarify information about infrastructure pattern components that microservices use. Infrastructure pattern components should be divided into two types: ones that provide services to microservices and others that call/request services from microservices.

Enhancement-1.3➔ Infrastructure pattern component is extended by two subtypes: Infrastructure Pattern Server Component and Infrastructure Pattern Client Component. The first represents infrastructure patterns provided by a microservice, i.e. subtypes of infrastructure microservice, while the second represents infrastructure patterns that are used/requested by a microservice, i.e. consumers of remote infrastructure microservices.

Context-4: In PIM (version 1), I previously considered representing the Data Store, Data Cache, Asynchronous Message Bus and Load Balancer as backend infrastructure patterns within the consumer microservice itself that can only be used/requested by client microservices, as explained in context-1. Therefore, the providers of these backend patterns were represented as mere infrastructure microservices, because the subtypes of Infrastructure Microservice did not yet include the Data Store, Data Cache, Asynchronous Message Bus or Load Balancer. To illustrate, the kafka microservice from case study 4 is a provider of asynchronous Message Bus as illustrated in Figure 6-9, however, according to the initial mapping rule 6 in Table 6-4, it is represented as mere Infrastructure Microservice. Similarly, the redis microservice from case study 7 is a Cache Store provider, and the mysql, neo4j and mongo microservices from case study 7 are Data Stores, however, according to the initial mapping rules, they are represented as mere Infrastructure Microservices.

Requirement-1.4➔ Data Store, Cash Store, Load Balancer and Asynchronous Message Bus need to be represented as Infrastructure Pattern Components provided and requested by microservices.
Enhancement-1.4➔ Data Store, Cash Store, Load Balancer and Asynchronous Message Bus are appended to an enumeration type Infrastructure Pattern Category so that they can be represented as Infrastructure Pattern Components.

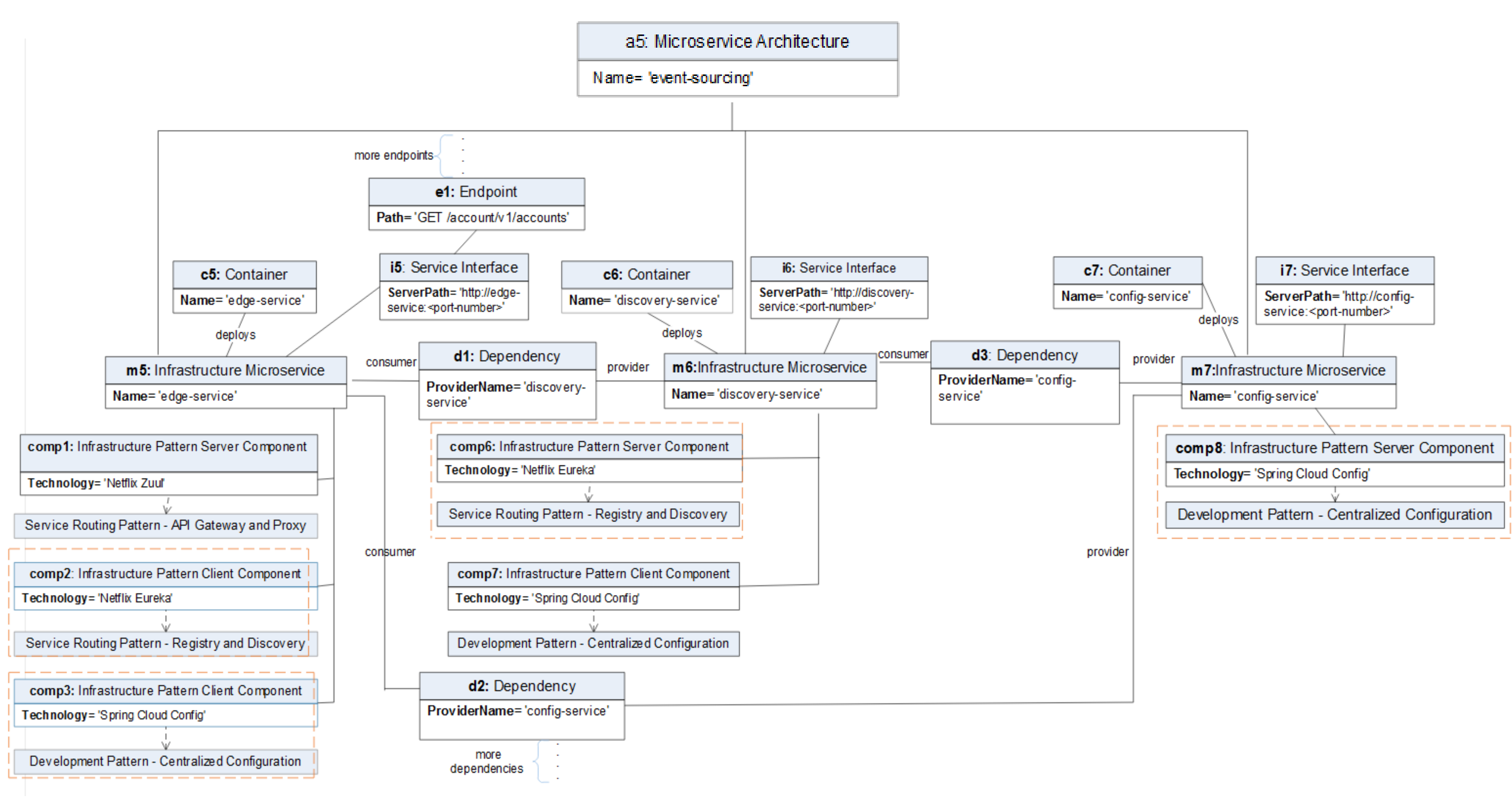


Figure 6-8: Enhanced PIM instance recovered for edge-service, discovery-service and config-service microservice from case study 7.

Table 6-4: Mapping rules applied in Kafka in case study 4

SEQ	PIM Concept (Source)	PIM Concept (Destination)	Mapping Rule	kafka
1	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the name of the root GitHub Repository which contains all artifacts of the application's project.	1
2	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
3	Microservice	Container	The name of Container concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
4	Microservice	Service Interface	The server path of Service Interface concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.	1
5	Infrastructure Microservice	-	An Infrastructure Microservice concept is indicated by a service container definition that does not have 'build' key in the Container Orchestration File of the application's project.	1
6	Infrastructure Microservice	-	A 'Kafka' Infrastructure Microservice concept is indicated by an 'image:' key with value that contains 'spotify/kafka' in the Container Orchestration File of the application's project.	1

The application of the first enhancement increment to the PIM metamodel (version 2), in response to all four requirements aforementioned, is shown in Figure 6-10. Categories of infrastructure patterns that are covered in the case study, in addition to the subtypes of Infrastructure Microservice, include Load Balancer, Circuit Breaker, Data Store, Cash Store and asynchronous Message Bus. These categories are all defined in an enumeration type called **Infrastructure Pattern Category**. The new **Infrastructure Pattern Component** concept enables future pattern categories to be added to the model smoothly by appending the enumeration type. Infrastructure Pattern Component is extended by two subtypes: **Infrastructure Server Component** and **Infrastructure Client Component**. If the used/requested patterns execute locally in the same microservice’s container, such as in-memory Data Store, embedded Cash Store, embedded asynchronous Message Bus, Load Balancer, Circuit Breaker, Log Generation, Log Correlation, etc., they can be represented using the supertype Infrastructure Pattern Component. Examples include the in-memory H2 database in user-service and friend-service microservices (from case study 4), the user-service, shopping-cart-service, payment-service, catalogue-service and account-service microservices (from case study 8), and the embedded Kafka message broker in user-service, recommendation-service and friend-service microservices (from case study 4). The main aim of those embedded components is to perform pre-production testing.

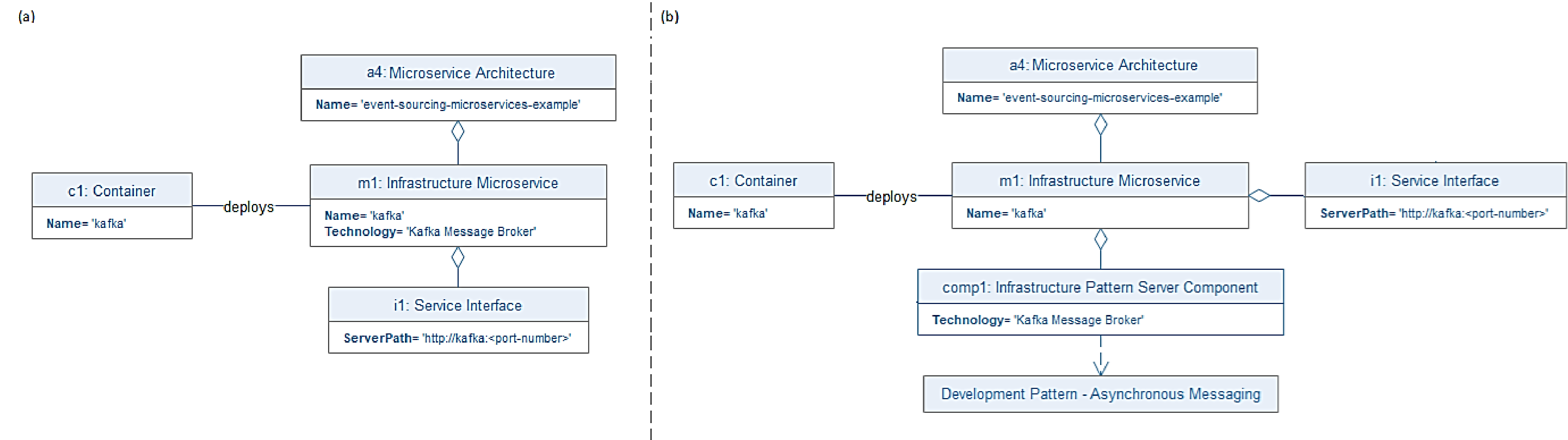


Figure 6-9: (a) PIM instances recovered for Kafka microservice from case study 4 based on PIM metamodel (Version 1). (b) Enhanced PIM instances recovered for *kafka* microservice based on enhanced PIM metamodel (Version 2).

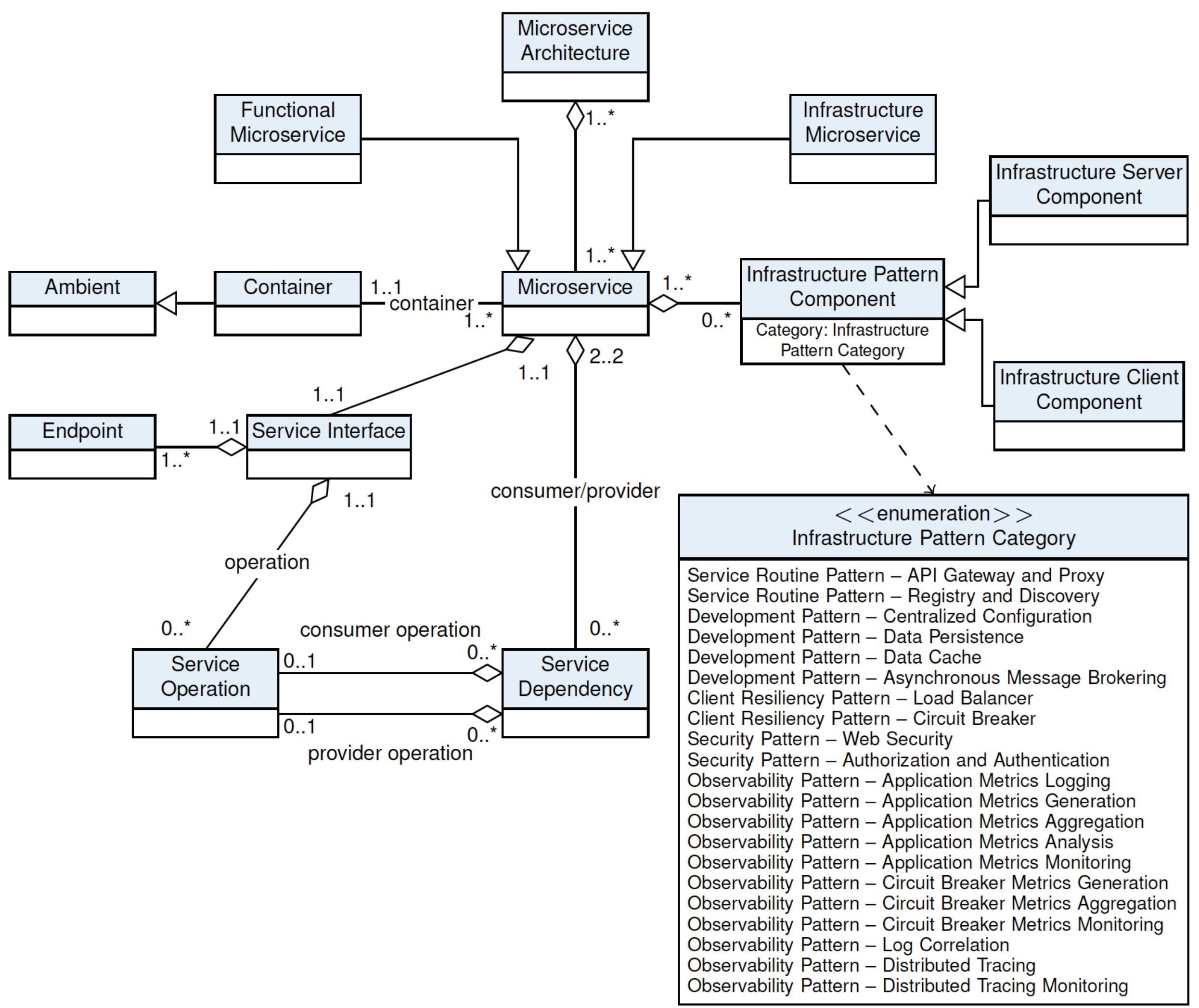


Figure 6-10: PIM metamodel (version 2).

Increment 2: Supporting Synchronous Communication through Endpoints

Context-1: Request-response synchronous inter-service communication is represented in the PIM metamodel (version 2, see Figure 6-10) using the Service Dependency concept, which specifies the Service Operation of the consumer that invokes a remote Service Operation of the provider. However, the request-response inter-service communication in the PIM metamodel (version 2) is based on Service Operation, which is low-level and probably unrecoverable in some cases, and Service Operations are not linked to their exposed Endpoints.

To illustrate, it can be observed from case study 1 that the card-statement-composite consumer microservice sends two consecutive requests: the first is a request to card provider microservice through an endpoint GET api/card/{cardId}, and the second is a request to statement provider microservice through an endpoint GET api/statement?cardId={cardId}, as illustrated in the sequence diagram in Figure 6-11. A PIM instance of the card-statement-composite microservice based on version 2 is shown in Figure 6-12. It states that Service Dependency instance (d3) defines the consumer's Service Operation, named getStatementByCardId, which invokes a remote provider's Service Operation named getCard. Similarly, Dependency instance (d4) indicates that the same Service Operation of the consumer invokes a remote Service Operation named getStatements of another provider statement. However, the invocation to the provider is made first to the Endpoint, which, in turn, maps the request to the Service Operation. One observation on the PIM instance is that it does not demonstrate the mapping of Service Operations by equivalent Endpoints.

Requirement-2.1➔ Service Operations should be linked to their exposed Endpoints.

Enhancement-2.1➔The association of Service Operation is repositioned from Service Interface to Endpoint. This association is an optional association that goes from Endpoint to Service Operation. It is optional support for modelling Infrastructure Microservices that tend to hide, i.e. abstract, implementation of their Service Operations.

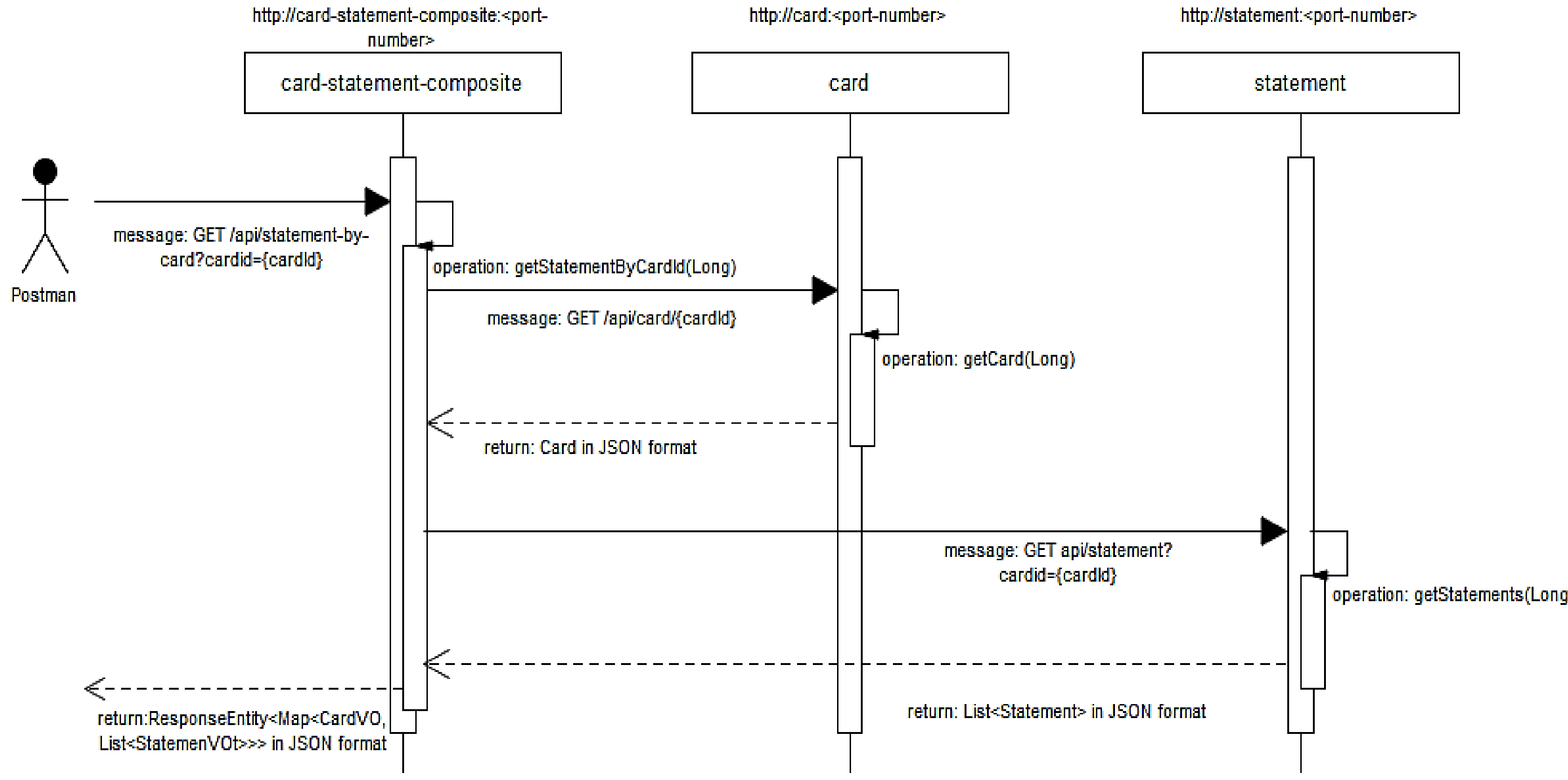


Figure 6-11: Synchronous request-response between card-statement- composite, card and statement microservices from case study 1.

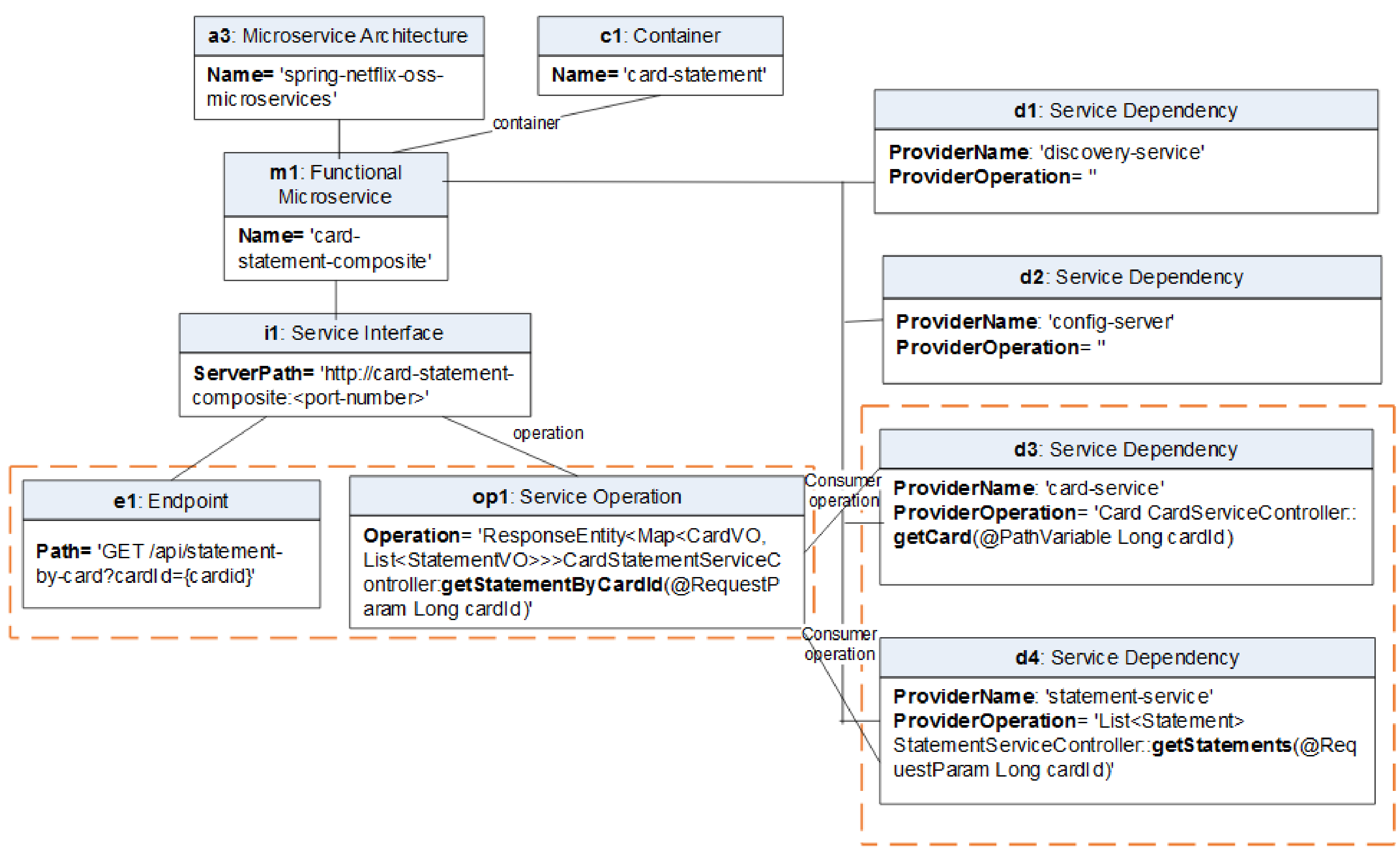


Figure 6-12: PIM instance recovered for card-statement-composite microservice from case study 1 based on PIM metamodel (version 2).

The second observation is that the representation of the request-response inter-service communication as an invocation to the remote provider's Service Operation is considered low-level, and Service Operations can be unrecoverable, especially in the case of modelling Infrastructure Microservices, as explained earlier in Increment-1.

Requirement-2.2 ➔ Service Dependency between two microservices should mainly involve the consumer microservice and provider microservice. Additional information to consider is the consumer's operation and the provider's endpoint if they are available.
Enhancement-2.2 ➔ The two associations between Dependency and Service Operations are replaced with two optional attributes: ConsumerOperation and ProviderEndpoint.

Having shifted the association of the Service Dependency concept to the provider's Endpoint instead of the provider's Service Operation (in Enhancement-2.2), information about the format of the request and response data messages at the provider's endpoint is missing. Service Operation already specifies the format of its request/input message via the data type of the parameter(s), if any, while the response/output message is specified by the data type of the object returned, if any. To illustrate via the Provider Operation value in Service Dependency instance (d3) as in Figure 6-12, we can see that the data type of the return object is Card, which corresponds to the payload of the response message. The data type is eventually converted to some standard data representation format, e.g. XML or JSON. Thus, it is necessary to provide the schema of the message in a standard format in order for it to be followed by the consumer.

Requirement-2.3➔ An Endpoint of a microservice should define the format and type of its data messages, if any.
Enhancement-2.3 ➔ A Service Message concept is introduced. Service Message is associated with Endpoint and it is defined by Type, i.e. request/response/error, Schema and Schema Format, i.e. XML/JSON.

Figure 6-13 shows the enhanced PIM (version 3). Figure 6-14 shows a PIM instance of card-statement-composite microservice based on the enhanced PIM (version 3).

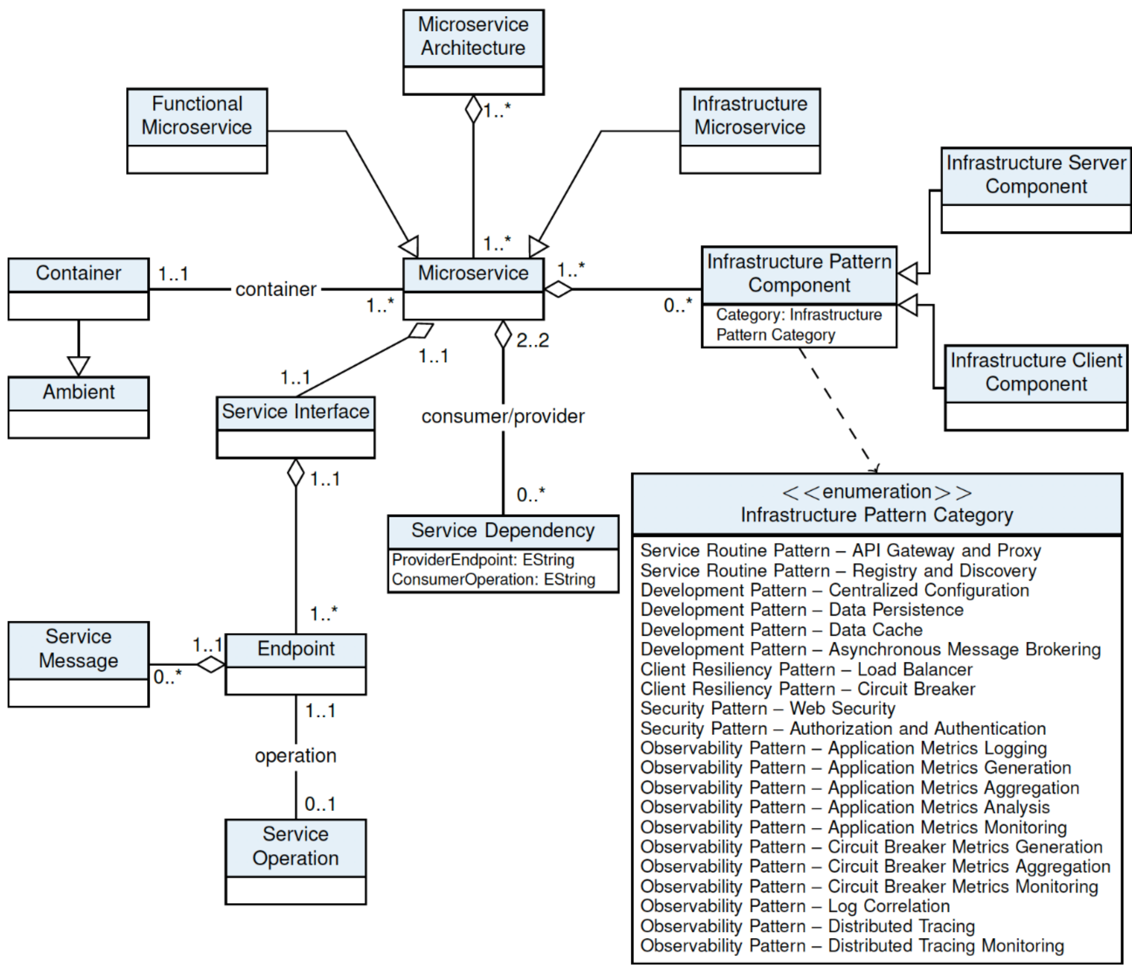


Figure 6-13: PIM metamodel (version 3).

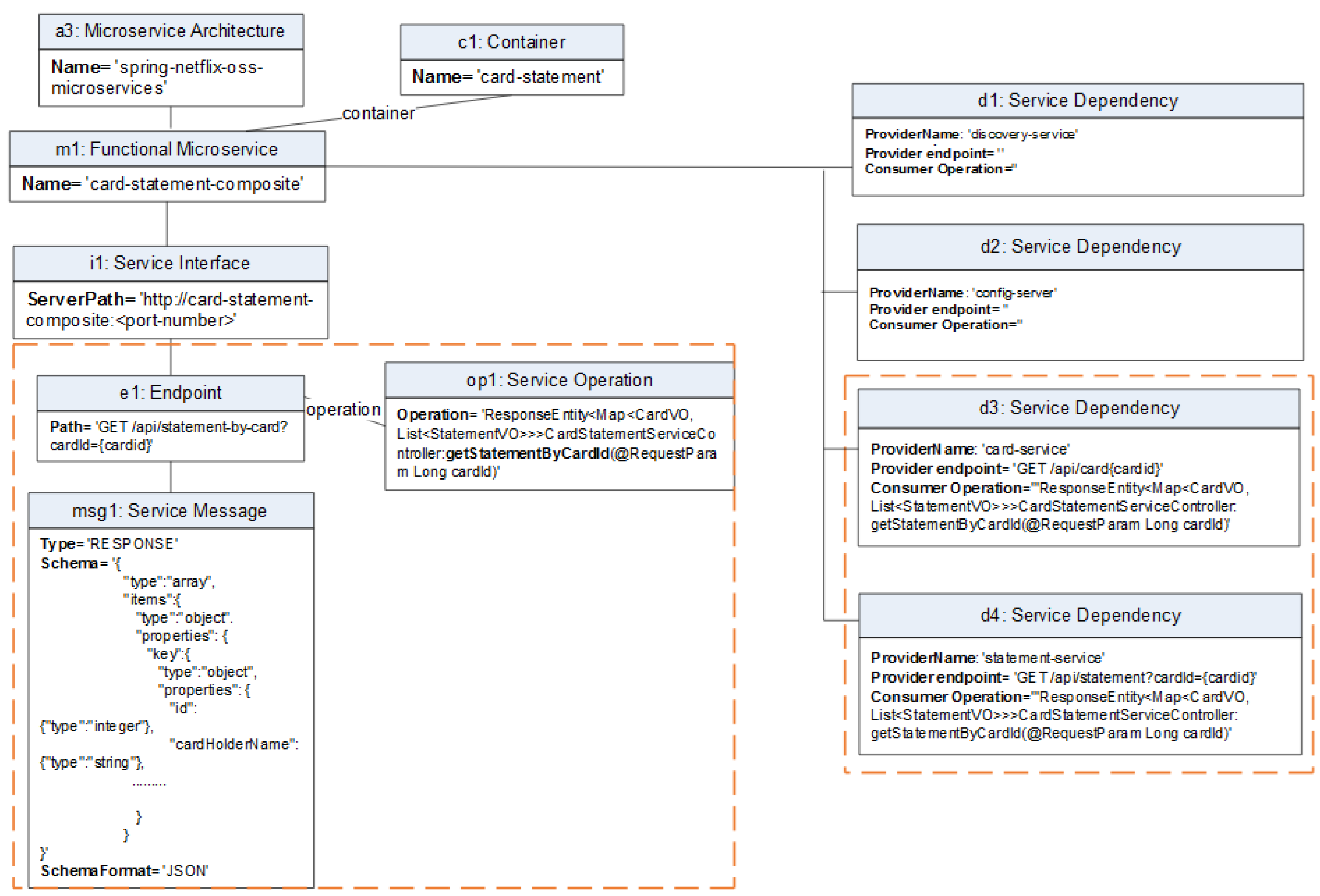


Figure 6-14: Enhanced PIM instance recovered for card-statement-composite microservice from case study 1 based on enhanced PIM metamodel (version 3).

Increment 3: Supporting asynchronous communication

Context-1: Unlike synchronous request-response, in asynchronous message-driven communication, the consumer does not directly invoke a remote Service Operation nor an Endpoint of the provider; instead, they send an event/message to an intermediary Infrastructure Microservice Asynchronous Messaging, which will eventually forward the event/message to the provider. However, the PIM metamodel (version 3) does not consider message-driven inter-service communication. To illustrate, consider the message-driven inter-service communication implemented in case study 2 using the sequence diagram depicted in Figure 6-15. This inter-service communication is initiated by an external synchronous request received on the endpoint of the weatherservice microservice: 'POST/weather/forecast'. Then, the forecast operation of the weatherservice microservice will publish an EventGetOutlook message accompanied with a routing key, weatherbackend:queue, to the RabbitMQ infrastructure microservice, i.e. a RabbitMQ message broker.

Inside RabbitMQ, as described in Figure 6-16, according to the AMQP protocol implemented by RabbitMQ, some kind of exchange will receive the message and then forward it to a particular queue, depending on the routing key provided. In this case, the EventGetOutlook message is received by the default exchange, forwarded to a particular queue named weatherbackend:queue and eventually received by the weatherbackend microservice, the provider, because it is subscribed to that particular queue. After that, the provider's operation handleMessage will process the message received and the response back to the weatherservice microservice by dropping an EventGeneralOutlook message onto RabbitMQ accompanied with a routing key: weatherservice:queue. The message will be forwarded by the exchange to weatherservice:queue and eventually received by the weatherservice microservice. The RabbitMQ infrastructure message bus/broker in case study 2 is configured to create two and three message queues (see Figure 6-16). The three queues are named weathersimple:queue, weatherbackend:queue and weatherservice:queue, and are bound to the message exchanges.

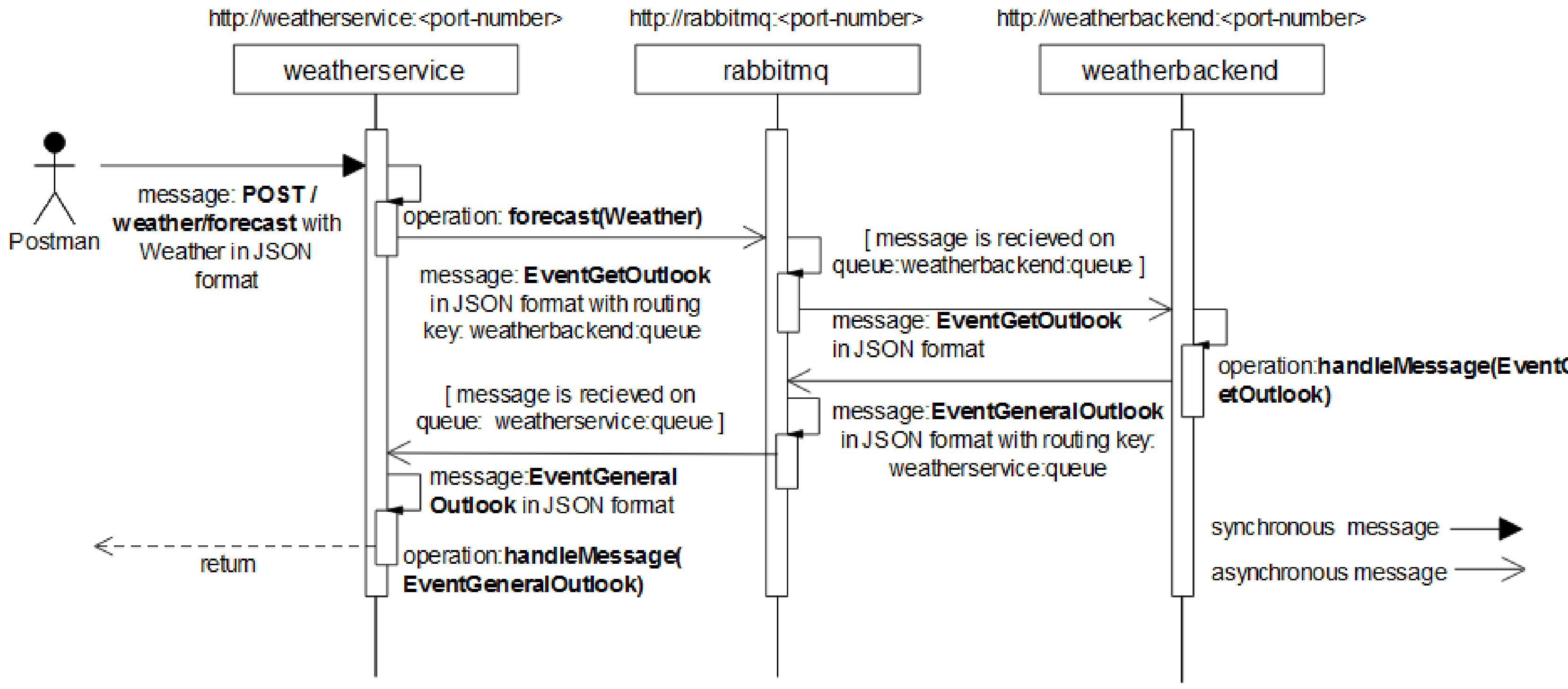


Figure 6-15:Asynchronous message-driven inter-service communication between weatherservice and weatherbackend microservices (case study 2).

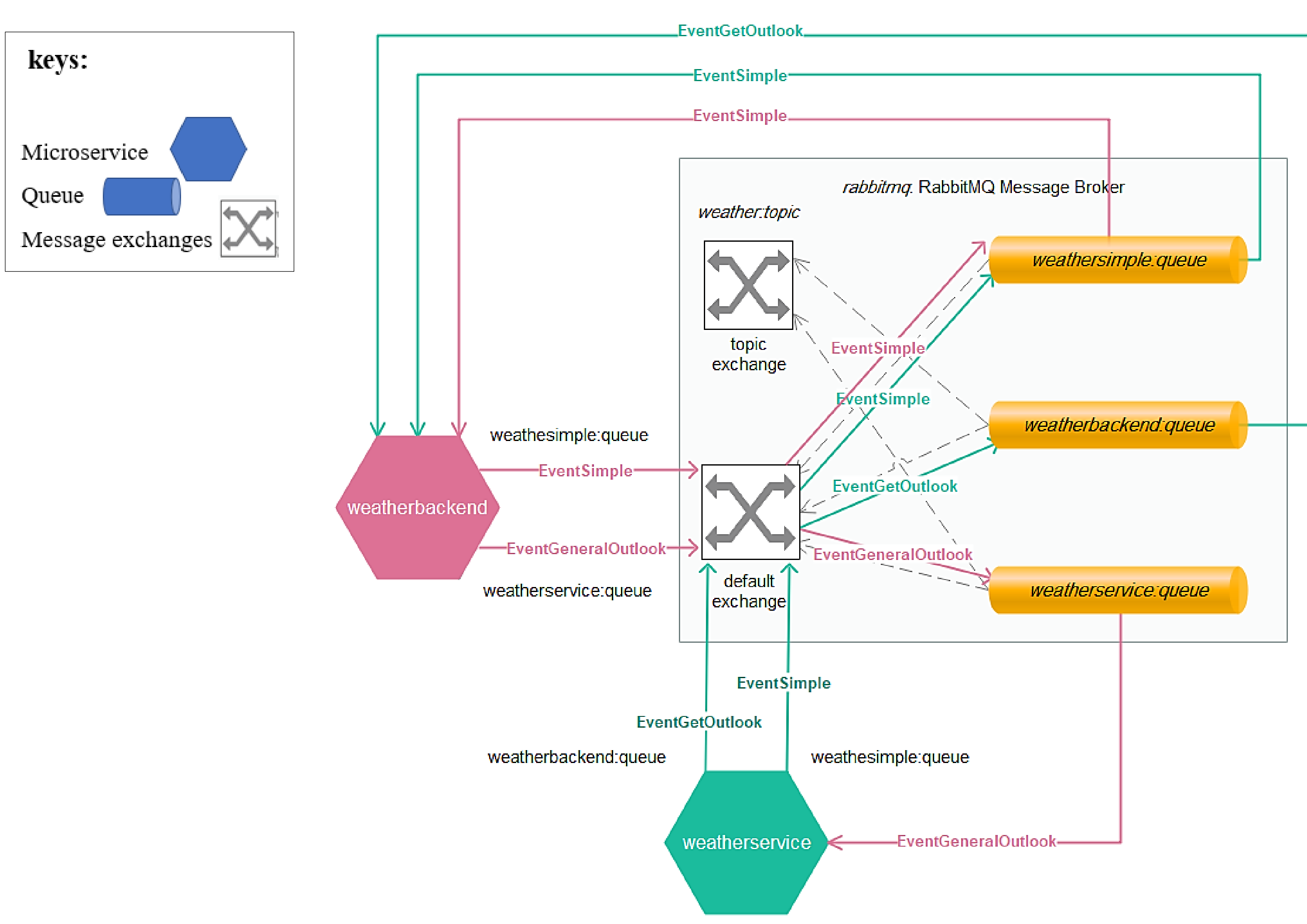


Figure 6-16: Internal setup of rabbitmq message broker from case study 2.

To illustrate, an outbound queue is where the weatherbackend microservice eventually forwards its events/messages to, however, an inbound queue is where the weatherbackend microservice is subscribed to and receives incoming event/messages. It can be concluded that if the outbound queue name of a consumer microservice matches the inbound queue name of the provider microservice, then the first microservice has a Service Dependency with the second, because it sends an event/message to the second's inbound queue.

This is comparable to request-response inter-service communication, where the microservice that sends a request to the Endpoint of another remote microservice is said to have a Service Dependency with that remote microservice. Therefore, an inbound queue of a microservice can be considered an asynchronous alternative to an Endpoint, and hence it needs to be exposed in a microservice's Service Interface in order to be reachable.

According to the previous statement, as seen in Figure 6-15, the weatherservice microservice is said to have a Service Dependency with the weatherbackend microservice, because it sends an asynchronous *EventGetOutlook* message to *weatherbackend:queue*, i.e. an inbound queue to the weatherbackend microservice. Similarly, the weatherbackend microservice is said to have a Service Dependency with the weatherservice microservice because it sends an asynchronous EventGeneralOutlook message to *weatherservice:queue*, i.e. an inbound queue to the weatherservice microservice. The Enhanced PIM (version 4) is provided in Figure 6-17. A PIM instance of weatherbackend microservices based on the enhanced PIM (version 4) is provided in Figure 6-18.

Requirement-3.1 ➔ A message-based asynchronous mechanism of inter-service communication using asynchronous inbound queues and messages should be represented.

Enhancement-3.1 ➔ The concept Queue Listener is introduced, defined by its Name and, as Endpoint, is associated with Service Interface.

Enhancement-3.2 ➔ As Endpoint, Queue Listener is associated with Service Message and maps to Service Operation.

Enhancement-3.3 ➔ Queue Listener and Endpoint are all generalized in a supertype concept called Message Destination, because they all represent the destination at which a remote message is received.

Enhancement-3.4 ➔ An attribute provider's Endpoint is replaced with the provider's destination, which represents both the provider's Endpoint and the provider's Queue.

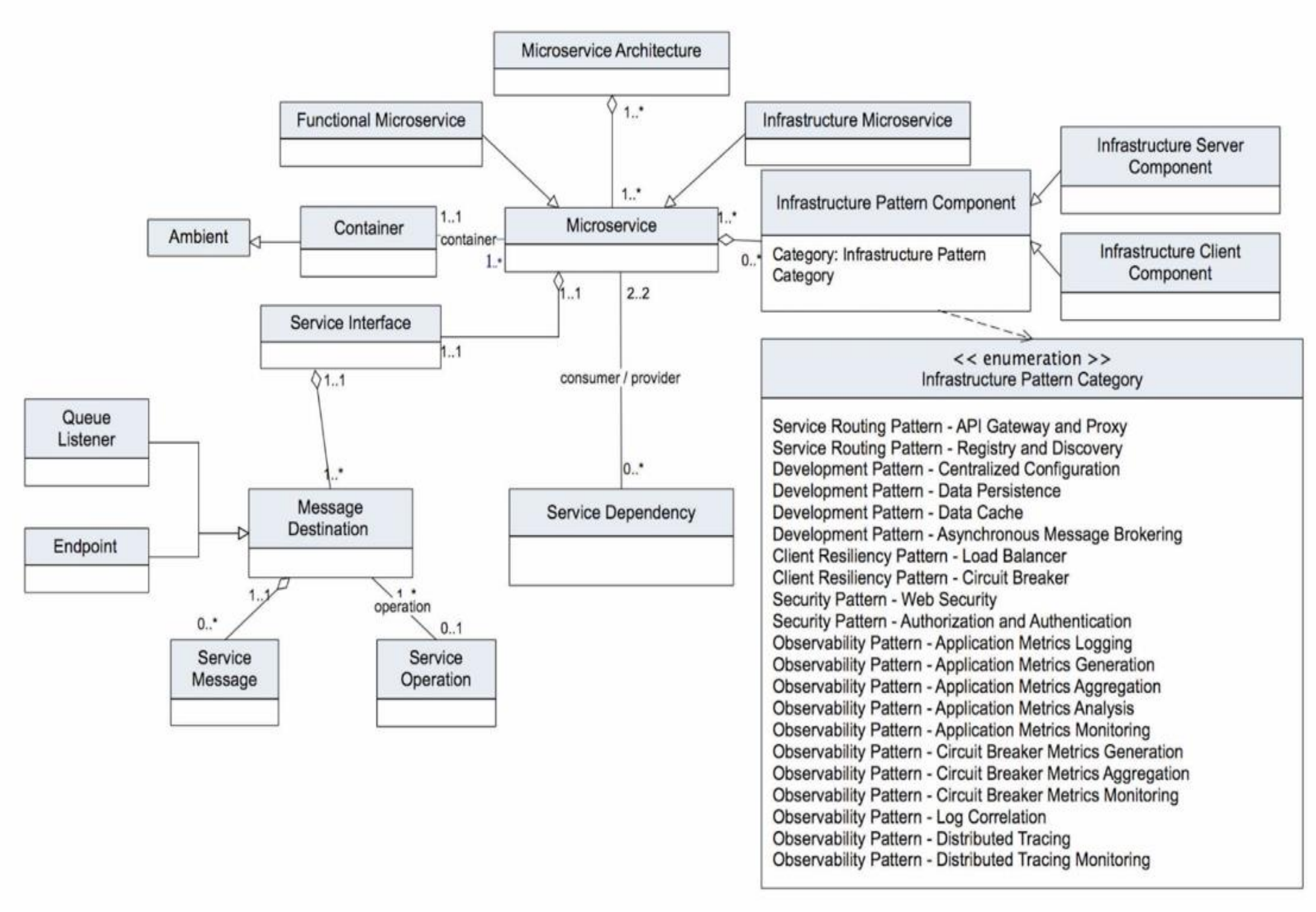


Figure 6-17: PIM metamodel (version 4).

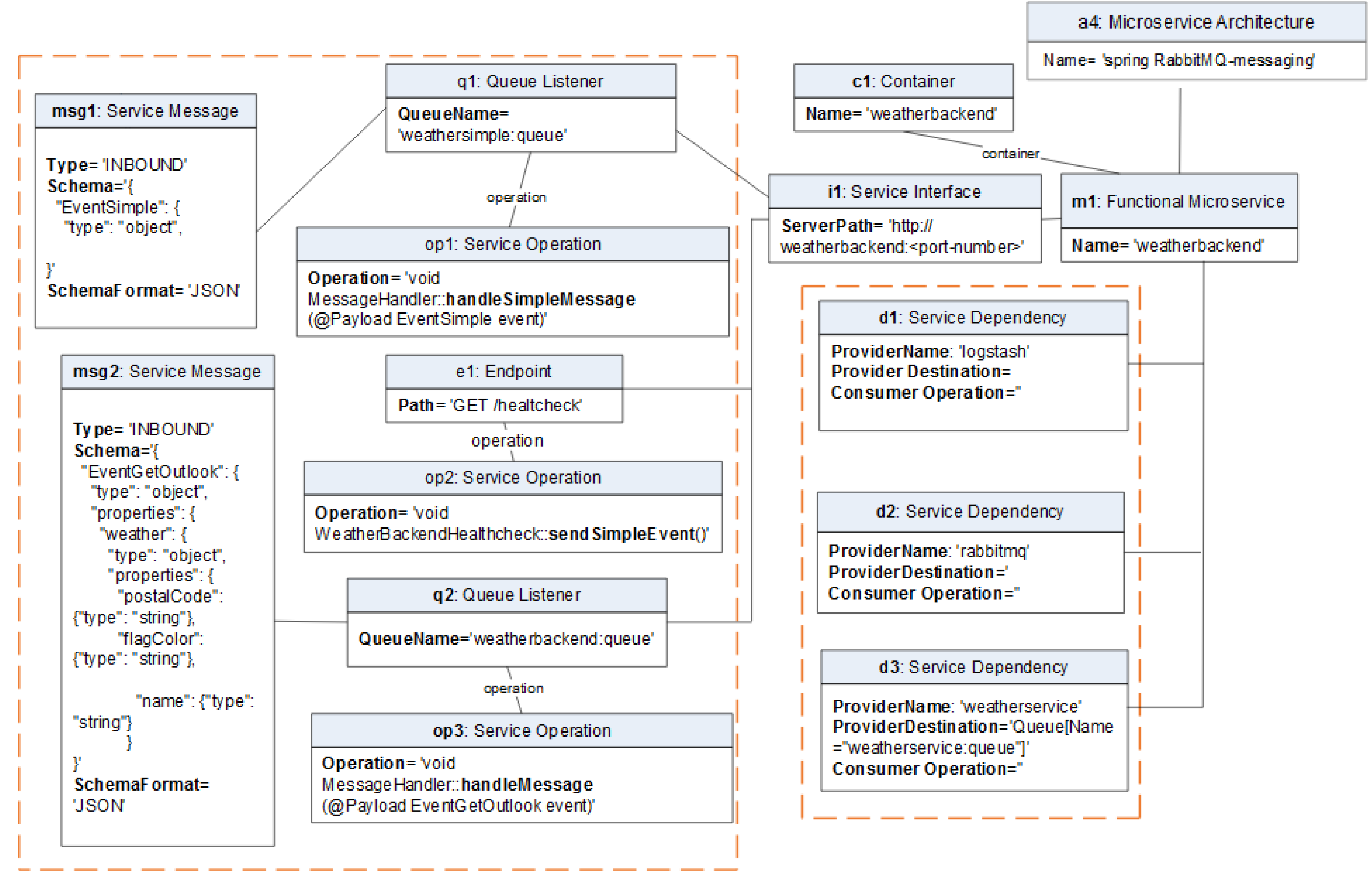


Figure 6-18: Enhanced PIM instance recovered for weatherbackend microservice from case study 2 based on PIM metamodel (version 4).

Increment 4: Support for Configuration

<u>Context-1:</u> PIM metamodel (version 4) doesn't support multiple configuration profiles implemented for a microservice application, which include default, development, Docker, Kubernetes and test. To illustrate, let us look at the analysis result of the microservice from case study 4, as illustrated in Figure 6-19. It is noticeable that the model has both embedded and client infrastructure components that are of the same pattern category. For example, components *comp2* and *comp5* both refer to the Kafka message broker; however, the first is a client component for a remote Kafka, while the second is an embedded component. Similarly, *comp3* and *comp4* both refer to the Neo4j graph database; however, the first is a client component for remote Neo4j, while the second is an embedded component. Both *comp6* and *comp7* are client components for service registry and discovery infrastructure patterns, however the first is implemented with Netflix Eureka and the second with Kubernetes. The reason for using this collection of similar components is that the application has multiple configuration profiles; each one is targeted at different running environments. The application uses embedded components in a testing environment only, while the use of service registry and discovery with Kubernetes is intended for an optional situation, where Kubernetes is running and available instead of Netflix Eureka. Apparently, PIM metamodel (version 4) needs enhancement to reflect such multiple configurations for multiple environments. The enhanced and final PIM (version 5) is provided in Figure 6-20. The PIM instance of *recommendation-service* microservice, based on the enhanced PIM (version 5), is provided in Figure 6-21.

Requirement-4.1 ➔ Multiple configuration profiles are needed to represent multiple environments.
Enhancement-4.1 ➔ An attribute named Environment is added for the Infrastructure Pattern Component, Message Destination and Dependency concepts.

As a result of this new empirical study, Misar PIM has become able to represent and abstract the technologies and patterns encountered in the systems analyzed as demonstrated in Table 6-5.

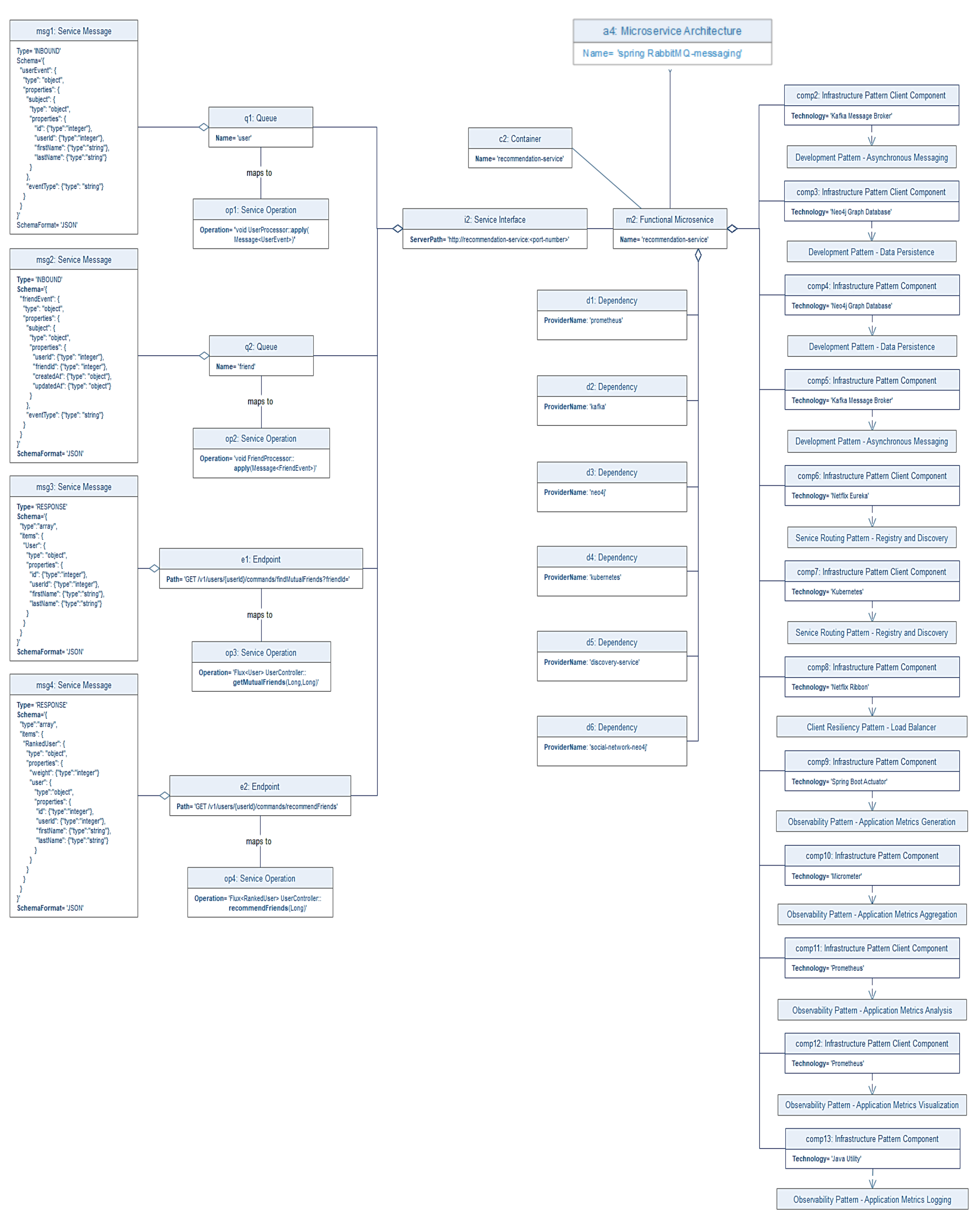


Figure 6-19: PIM instance recovered for *recommendation-service* microservice from case study 4 based on PIM metamodel (version 4).

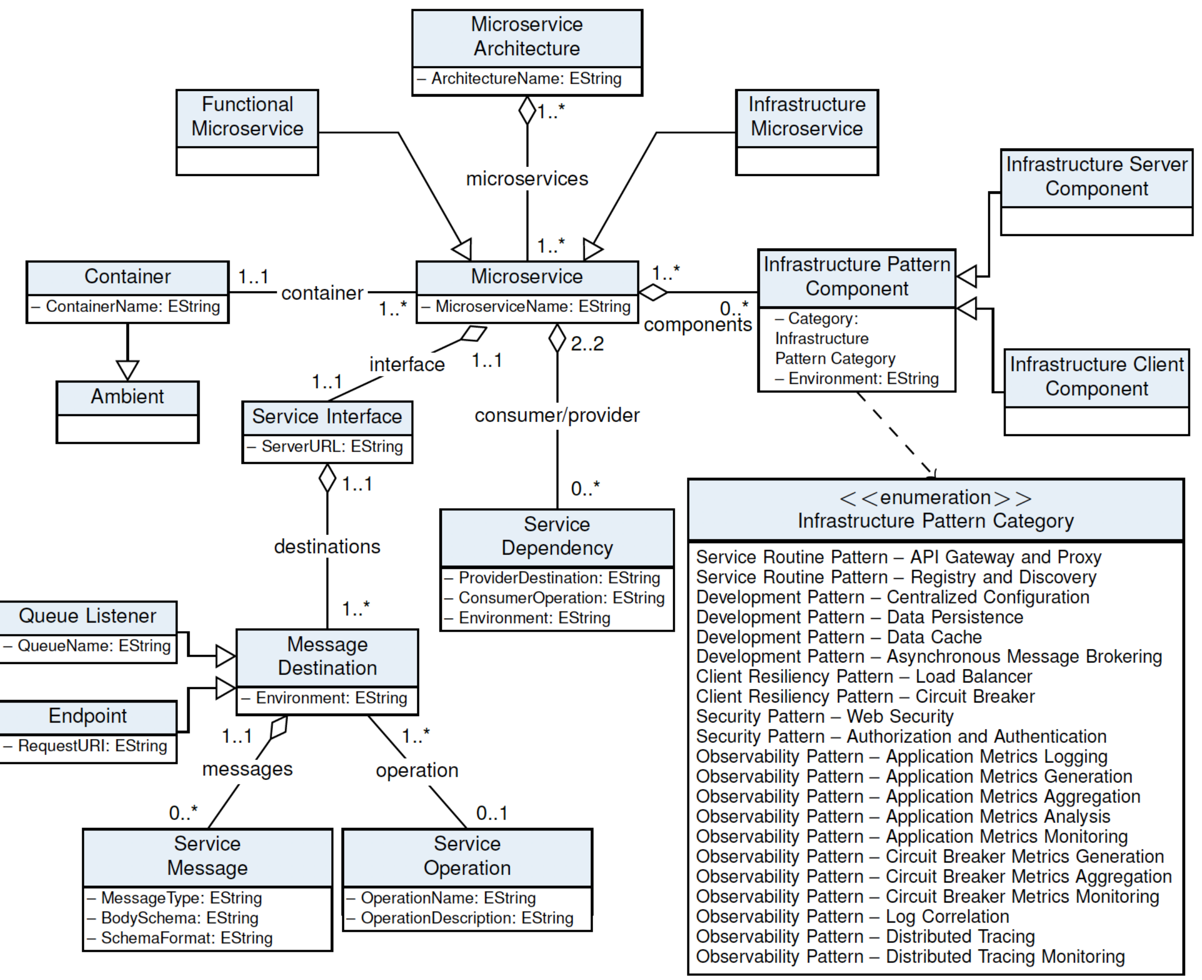


Figure 6-20: Final PIM metamodel (version 5).

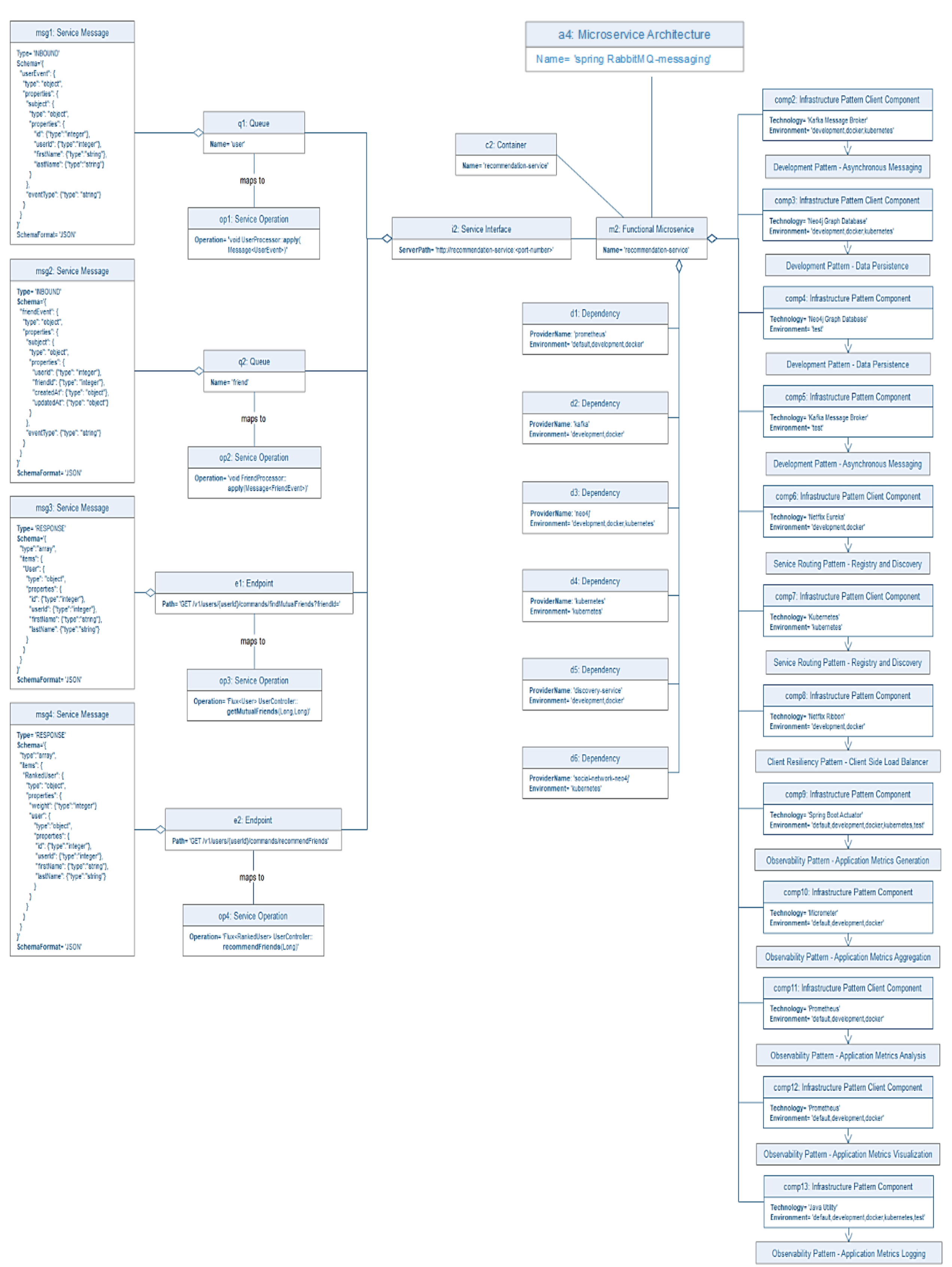


Figure 6-21: Enhanced PIM instance recovered for *recommendation-service* microservice from case study 4 based on enhanced PIM metamodel (version 5).

Table 6-5: Technologies encountered in the systems analyzed

ID	Main Pattern/Technology	MiSAR PIM Representation/Abstraction
1	Synchronous Communication implemented with *Spring Feign REST Client.*	The consumer **Microservice** will have a **Dependency** concept with a reference to the **Endpoint** of the provider which, in turn, has one or more **ServiceMessage** concepts to describe the schema of business data requested by the consumer and returned by the provider.
	Centralized Circuit Breaker Monitoring implemented with *Netflix Hystrix Dashboard* and *Netflix Turbine.*	The monitoring server is an **InfrastructureMicroservice** which has two **InfrastructureServerComponent** concepts; one with *category* value of *Observability_Pattern_Circuit_Breaker_Metrics_Aggregation* and another one with *Observability_Pattern_Circuit_Breaker_Metrics_Monitoring*.
2	Asynchronous Communication via *RabbitMQ* as a middle message broker.	The consumer **Microservice** will have a **Dependency** concept with a reference to the **QueueListener** of the provider which, in turn, has one or more **ServiceMessage** concepts to describe the schema of business data received at the provider's queue from the consumer. The communicating **Microservices** will have an **InfrastructureClientComponent** with *category* attribute of value: *Development_Pattern_Asynchronous_Message_Brokering*. The middle *RabbitMQ* message broker itself is an **InfrastructureMicroservice** which has an **InfrastructureServerComponent** with *category* attribute of value: *Development_Pattern_Asynchronous_Message_Brokering*
	Centralized Log Analysis and Monitoring implemented with *ELK* Stack *(Elasticsearch, Logstash and Kibana).*	The three servers in the stack are **InfrastructureMicroservices** each has one **InfrastructureServerComponent** with *category* attribute of value: *Observability_Pattern_Application_Metrics_Analysis*, *Observability_Pattern_Application_Metrics_Aggregation* and *Observability_Pattern_Application_Metrics_Monitoring* respectively.
3	OAuth2 Token-Based Security implemented with *Spring OAuth2* and *Spring Security.*	The OAuth2 server is an **InfrastructureMicroservice** which has **InfrastructureServerComponent** with *category* attribute of value: *Security_Pattern_Authorization_and_Authentication*
4	Event Sourcing and Command Query Responsibility Segregation (CQRS) where changes to the application's state are stored as a sequence of events in *Kafka* topics.	The **Microservice** that initiates the event will have a **Dependency** concept with references to the **QueueListener** concept of all **Microservices** that will receive the event. The **QueueListener** concept has one or more **ServiceMessage** concepts to describe the schema of event to be received. The communicating **Microservices** will have an **InfrastructureClientComponent** with *category* attribute of value: *Development_Pattern_Asynchronous_Message_Brokering*. The middle *Kafka* message broker itself is an **InfrastructureMicroservice** which has an **InfrastructureServerComponent** with *category* attribute of value: *Development_Pattern_Asynchronous_Message_Brokering*

	Reactive Streams implemented with *Spring WebFlux* and *Spring Data R2DBC*.	REST endpoints and database connections that execute internally using reactive streams are abstracted respectively by the **Endpoint** concept as well as the **InfrastructureClientComponent** with *category* attribute of value: *Development_Pattern_Data_Persistence*
5	Sidecar Proxy to non-JVM Applications implemented with *Spring Sidecar*.	Sidecar proxy is an **InfrastructureMicroservice** which has one **InfrastructureServerComponent** concept with *category* attribute of value: *Service_Routing_Pattern_API_Gateway_and_Proxy*
6	Distributed Tracing implemented with *Spring Sleuth* and *Zipkin*.	The **Microservice** that implements sleuth to tag requests and logs with tracing id will have two **InfrastructurePatternComponent** concepts with *category* attribute of value: *Observability_Pattern_Log_Correlation* and *Observability_Pattern_Distributed_Tracing* while *Zipkin* server is an **InfrastructureMicroservice** which has an **InfrastructureServerComponent** concept with *category* attribute of value: *Observability_Pattern_Distributed_Tracing_Monitoring*
7	Polyglot Data Management implemented with Relational Database (*MySQL*), NoSQL Database (*Mongo*), Graph Data Models (*Neo4j*) and Data Cache (*Redis*).	Each data store server is an **InfrastructureMicroservice** which has one **InfrastructureServerComponent** concept with *category* attribute of value: *Development_Pattern_Data_Persistence.* The Redis data cache is an **InfrastructureMicroservice** which has one **InfrastructureServerComponent** concept with *category* attribute of value: *Development_Pattern_Data_Cache.*
8	Big Data Storage and Analysis implemented with *Hadoop*.	Hadoop server is an **InfrastructureMicroservice** which has one **InfrastructureServerComponent** concept with *category* attribute of value: *Development_Pattern_Data_Persistence.*
9	*Netflix Consul* Infrastructure.	Consul server is an **InfrastructureMicroservice** which has three **InfrastructureServerComponent** concepts with *category* attribute of value: *Service_Routing_Pattern_Registry_and_Discovery, Development_Pattern_Centralized_Configuration* and *Development_Pattern_Asynchronous_Message_Brokering* respectively.

- **RQ2:** *Do the current MiSAR mapping rules map microservice Java and Spring Cloud systems into architectural models?*

To answer RQ2, in the first section I explained the mapping rules structure and demonstrated the enhancements made to the mapping rules structure that enabled them to map the microservice implementation into the architectural elements. Then in the second section, I present some of the mapping rules analysis applied to the selected systems in Table 6-7.

Mapping Rules Structure. The structure of MiSAR's mapping rules evolved from textual sentences written in natural language, as presented in Chapter 5, into a structured tree that maps PSM element(s) into target PIM element(s). Each mapping rule conforms to the metamodel depicted in Figure 6-22. Mapping rules are represented as a group of source PSM elements at the left-hand side (LHS), specified by their attributes' values and the references between them, which transforms to a group of target PIM elements at the right-hand side (RHS), with specific attribute values and references between them. LHS-elements are identified before the word 'indicates' and RHS-elements are identified after 'indicates'. Chapter 7 explain how an instance of the mapping rule metamodel is created.

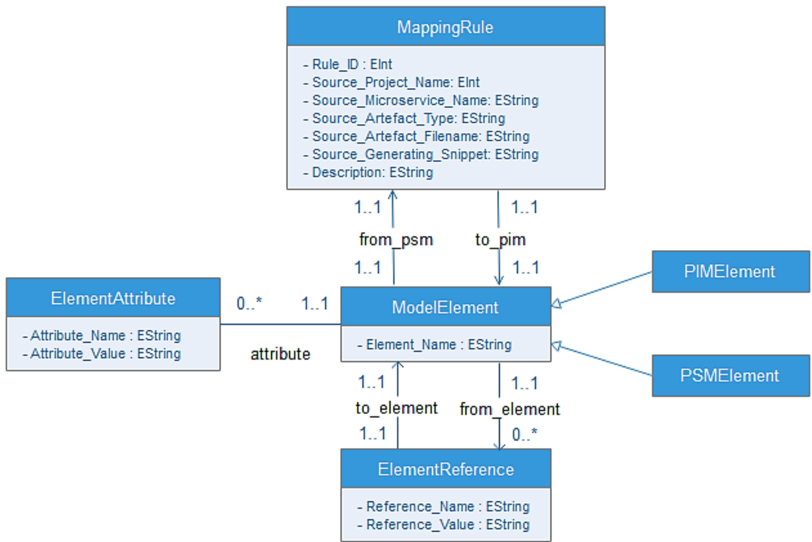


Figure 6-22: MiSAR mapping rule metamodel

The root of each rule includes textual description and information about the source project, artefact type and a key snippet from which this rule was first encountered (from the studies presented in chapters 5 and 6). This history of each rule was required to track the added value of each new case study. Rule Identifier (RID) is a unique value recorded for each mapping rule. Source Project Name and Source Microservice Name belong to a particular microservice in a particular case study where the mapping rule was first observed. Source Artefact Type indicates that this rule was extracted from a certain artefact type, e.g. a DockerComposeFile. Source Artefact Filename refers to the name and path of a particular file from which a given rule was first extracted. Source Generating Snippet is the first text/code snippet that contained keywords indicating the rule. The textual mapping rule, written in natural language, as presented in Chapter 5, is provided in the Description. Using a structured format, as in the dataset sample given in Table 6-6, we can obtain all target PIM elements transformed by a particular source PSM element, say DockerContainerDefinition (DCD). In addition, we can get alternative rules by selecting all rules which transform a specific target PIM element, say InfrastructurePatternComponent (IPC) with category value Development Pattern – Data Persistence, then group them by distinct source PSM elements, say DependencyLibrary (DL) and ConfigurationProperty (CP).

Table 6-6: Sample of MISAR mapping rules structured dataset.

Rule	PSM	[Attribute = Value]	PIM	[Attribute = Value]	PIM -> Reference
R1	DCD	[ImageField= '%**consul**%']	IPSC	[Technology = '**Consul**'] [Category = …Registry and Discovery]	
R2	DCD	[ImageField= '%**consul**%']	IPSC	[Technology = '**Consul**'] [Category = Centralized Configuration]	
R3	DCD	[ImageField= '%**consul**%']	IPSC	[Technology = '**Consul**'] [Category = …Asynchronous Messaging]	
R4	DCD	[ImageField= '%**consul**%']	E	[Path = '**PUT /catalog/register**']	
R5	DCD	[ImageField= '%**consul**%']	SM	[Type = '**REQUEST**'] [Schema = '**{…}**'] [SchemaFormat = '**JSON**]	E -> Service messages
R6	DCD	[ImageField= '%**consul**%']	IM		
R7	DL	[Library = '**neo4j-ogm-embedded-drive**'] [LibraryScope = *'{library-scope}'*]	IPC	[Technology = '**Neoj4 Graph Database**'] [Category = …Data Persistence] [Environment = *'{library-scope}'*]	
R8	CP	[…PropertyName = '**spring.data.neo4j.embedded.enabled**'] [PropertyValue = '**true**'] [ConfigurationProfile = *'{configuration-profile}'*]	IPC	[Technology = '**Neoj4 Graph Database**'] [Category = …Data Persistence] [Environment = *'{configuration-profile}'*]	

Key:

DCD=DockerContainerDefinition, **DL**=DependencyLibrary, **CP**=ConfigurationProperty, **M**=Microservice, **IM**=InfrastructureMicroservice, **C**=Container, **SI**=ServiceInterface, **SM**=ServiceMessage, **E**=Endpoint, **IPC**=InfrastructurePatternComponent, **IPSC**= InfrastructurePatternServerComponent

The aim of this change is to formalize the transformation terms, facilitate their implementation and eventually make the recovery process automatic. Using the structured format, one can facilitate storage, filtering and grouping of rules, query all the mapping rules that transform a particular target (PIM element), then group them by each source (PSM element). This grouping is important for implementing mapping rules using QVT mappings, since every QVT mapping transforms one PSM element into one PIM element. All mapping rules that belong to each group will be written as if they were statements inside one QVT mapping. Eventually, the mapping rules become easy to implement and the recovery process becomes automated.

Mapping rules analysis. I analysed the nine systems and a sample of the mapping rules obtained are shown in Table 6-7. The results are as follows:

(a) Validated Existing Mapping Rules: Most of the mapping rules that were defined in chapter 5 still apply in the new systems. Examples of these are rule **R1,** which recovers Microservice Architecture, rule **R2** and rule **R3**, which recovers Container and Microservice respectively. **R4**, which recovers Infrastructure Microservice, rule **R7**, which recovers Service Operation, and rule **R10**, which recovers Service Dependency, as illustrated in Table 6-7.

(b) Mapping Rules (Modification/New): This involves updates to existing mapping rules to reflect the enhanced PIM metamodel (version 5) presented in response to RQ1. Some resultant rules were newly created, while others are modified versions of existing rules. For instance, the introduction of the new PIM concept Infrastructure Pattern Component (and its sub-types) presented in Increment-1 resulted in rule **R5**, which is a replacement of a previous rule in study 1 (Chapter 5). Rules **R6**, **R9** and **R11** are newly added. Rule **R6** is newly added to recover a production Endpoint that is automatically created by the Spring Actuator library at runtime without any implementation code existing in the source files. Rules **R9** and **R11** were newly added as a result of Increment-3 to recover the Queue Listener concept which is a Message Destination based on message-driven inter-service communication. This concept was introduced after analysis of case study 2 and case study 8.

(c) Mapping Rules Variation: This involves the addition of new mapping rules that recover an existing PIM concept, which are implemented in technologies that were not encountered in previous studies (Chapter 5). As an example, rule **R8** recovers the Service Operation concept, which is the exact output of **R7** except that the input in **R8** represents a reactive, non-blocking microservice. This reactive architecture was introduced after the analysis of case study 4.

d) Hard-coded mapping rules: The addition of mapping rules with hard-coded values to recover production endpoints along with their message types and the formats of outsourced famous infrastructure technologies. An example is rule **R6**. The information of such implicit endpoints is hard-coded in the new version of mapping rules.

Table 6-7: Sample of mapping rules analysis

Mapping Rule	**Case study where the rule applies**									
	Ch5[17]	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
R1: One *Distributed Application Project* with *Application Name* value: *{architecture-name}* always maps to one *Microservice Architecture* with *Architecture Name* value: *{architecture-name}*. (Rule text at the time of writing (chapter 5) was: "The name of microservice architecture concept is indicated by the name of the root GitHub repository which contains all artefacts of the application's project.")	x	x	x	x	x	x	x	x	x	x
R2: One *Docker Container Definition* with *Container Name* value: *{container-name}* always maps to one *Container* with *Container Name* value: *{container-name}*. (Rule text at the time of writing (chapter 5) was: "The name of container concept is indicated by the key name of service container definition in the container orchestration file of the application's project.")	x	x	x	x	x	x	x	x	x	x
R3: One *Docker Container Definition* with *Container Name* value: *{microservice-name}* always maps to one *Microservice* with *Microservice Name* value: *{microservice-name}*. (Rule text at the time of writing (chapter 5) was: "The name of microservice concept is indicated by the key name of service container definition in the container orchestration file of the application's project.")	x	x	x	x	x	x	x	x	x	x
R4: One *Docker Hub Image Container* with *Container Name* value: *{microservice-name}* always maps to one *Infrastructure Microservice* with *Microservice Name* value: *{microservice-name}*. (Rule text at the time of writing (chapter 5) was: "An infrastructure microservice concept is indicated by a service container definition that does not have 'build:' key in the container orchestration file of the application's project.")	x	x	x	x	x	x	x	x	x	x
R5: One *Docker Hub Image Container* with *Image Field* value which contains: **"consul"** indicates one *Infrastructure Server Component* with *Category* value: *Service Routing Pattern - Registry and Discovery*, another *Infrastructure Server Component* with *Category* value: *Development Pattern - Centralized Configuration* and a third *Infrastructure Server Component* with *Category* value: *Development Pattern - Asynchronous Messaging*. (Rule text at the time of writing (chapter 5) was: "A registry and discovery concept with technology of 'Netflix Eureka' is indicated by an 'image:' key with value that starts with 'consul' in the container orchestration file of the application's project.")	x									X
R6: A *Dependency Library* with *Library Name* value: **"spring-boot-starter-actuator"** and *Library Scope* value: *{destination-environment}* indicates one	x	x		x	x	x		x	x	x

[17] Systems listed in Table 5-3, Chapter 5.

Endpoint with *Request URI* value: "**GET /actuator/health"** and *Environment* value: *{destination-environment}* which has a *Service Message* with *Message Type* value: "**RESPONSE"**, *Schema Format* value: "**JSON"** and *Body Schema* value: **"{"type":"object","properties":{"status":{"type":"string"},"details":{"type":"object"}}}".**										
R7: A *Java Annotation* with *Annotation Name* value that ends with: "**Mapping"** which belongs to a *Java Method* in a *Java Class* with *Java Annotation* value that ends with: **"Controller"** and returns a *Java Data Type* with *Element Identifier* value: *{datatype-name}* indicates a *Service Operation* with *Operation Name* value: *{operation-name}* and *Operation Description* value: **"Returns a response message:** *{datatype-name}***".** (Rule text at the time of writing (chapter 5) was: "A service operation concept is indicated by a java method with a java annotation that ends with 'Mapping' and belongs to a java class with annotation that ends with 'Controller' in the source code file of the microservice's project.")	x	x	x	x	x	x	x	x	x	x
R8: A *Java Annotation* with *Annotation Name* value that ends with: "**Mapping"** which belongs to a *Java Method* in a *Java Class* with *Java Annotation* value that ends with: **"Controller"** and returns a *Java Data Type* with *Element Identifier* value: **"[Mono|Flux<]***{datatype-name}***[>]"** indicates a *Service Operation* with *Operation Name* value: *{operation-name}* and *Operation Description* value: **"Returns a response message:** *{datatype-name}***".**					x					
R9: A *Java Annotation* with *Annotation Name* value: "**RabbitListener"** which has a *Java Annotation Parameter* with *Parameter Name* value: "[**value|queues]"** and *Parameter Value* value: *{queue-name}* and belongs to a *Java Method* with *Element Profile* value: *{destination-environment}* indicates a *Queue Listener* with *Queue Name* value: *{queue-name}* and *Environment* value: *{destination-environment}.*			x						x	
R10: A *Java Method* with *Element Identifier* value: **"[get|post]For[Entity|Object]|[put|delete]"** whose parent is a *Java User Defined Type* with *Element Identifier* value: **"[Rest|OAuth2Rest]Template**" and *Package Name* value: **"org.springframework[.web|.security.oauth2].client"** and *Element Profile* value: *{dependency-environment}* which has one *Java Method Parameter* with *Parameter Order* value: **"1"** and *Field Value* value: **"%://***{provider-name}***[:***{port-number}***]/***{endpoint-path}***"** indicates a *Service Dependency* with *Provider Name* value: *{provider-name}*, *Provider Destination* value: **"Endpoint[RequestURI:[GET|POST|PUT|DELETE]** *{endpoint-path}***]"** and *Environment* value: *{dependency-environment}.* (Rule text at the time of writing (chapter 5) was: "A microservice provider to a microservice is indicated by the first parameter of a java method 'getForEntity', 'postForEntity', 'getForObject', 'postForObject', 'put' or 'delete' which belongs to a java class	x			x	x		x	x	x	x

'RestTemplate' or 'OAuth2RestTemplate' in the source code file of the microservice's project.")										
R11: A *Java Method* with *Element Identifier* value: **"convertAndSend"** whose parent is a *Java User Defined Type* with *Element Identifier* value: "**[Amqp|Rabbit]Template",** *Package Name* value: **"[org.springframework.amqp[.core|.rabbit.core]"** and *Element Profile* value: *{dependency-environment}* which has one *Java Method Parameter* with *Parameter Order* value: **"2"** and *Field Value* value: *{routing-key}* whose type is a *Java Class Type* with *Element Identifier* value: **"String"** such that there is a *Queue Listener* with *Queue Name* value that contains: *{routing-key}* and belongs to a *Microservice* with *Microservice Name* value: *{provider-name}* indicates a *Service Dependency* with *Provider Name* value: *{provider-name}*, *Provider Destination* value: **"QueueListener[QueueName:***{queue-name}***]"** and *Environment* value: *{dependency-environment}.*			x						x	

RQ3: *Can an enhanced MiSAR approach recover architectural models?*

The final MiSAR artefacts (mapping rules and metamodel) were applied to check the validity of MiSAR in regards to its ability to create an architectural model (diagram). To validate this, a new system called Microservices Sample was selected (Vijayendra, 2019), which is an open-source microservice project. The following steps were taken to recover the architecture model of this system (steps 1 to 3) and the following shows the simplified architectural models for this case study (step 4). The methodology of the recovery process also supports internal feedback. This can take the form of built-in validation mechanisms to detect inconsistency and incompleteness (step 5) and then update the MiSAR repository (step 6).

Step 1- Artefact collection (manual): The artefacts collected from the project are shown in Table 6-8 that shows artefacts URL collected for each artefact type.

Table 6-8: Artefacts collected for Microservices Sample from its GitHub repository

Repository	Artifacts Collected	Artifact Type
project-initializer	project-initializer/pom.xml	Multi-Application Project Build File
	project-initializer/build/docker/docker-compose.yml	Multi-Application Project Docker Compose File
api-gateway	api-gateway/pom.xml	One-Application Project Build File
	api-gateway/src/main/docker/Dockerfile	One-Application Project Dockerfile File
	api-gateway/src/main/resources/application.yml	One-Application Project Configuration File
	api-gateway/src/main/java/com/mudigal/** (ALL*.java)	One-Application Project Java Source File
service-one	service-one/pom.xml	One-Application Project Build File
	service-one/src/main/docker/Dockerfile	One-Application Project Dockerfile File
	service-one/src/main/resources/application.yml	One-Application Project Configuration File
	service-one/src/main/java/com/mudigal/** (ALL *.java)	One-Application Project Java Source File
service-two	service-two/pom.xml	One-Application Project Build File
	service-two/src/main/docker/Dockerfile	One-Application Project Dockerfile File
	service-two/src/main/resources/application.yml	One-Application Project Configuration File
	service-two/src/main/java/com/mudigal/** (ALL *.java)	One-Application Project Java Source File
web-application	web-application/docker/Dockerfile	One-Application Project Dockerfile File

Step 2- Instantiate PSM instance (manually): As MiSAR adopts the PSM-to-PIM transformation, the next step is to create a PSM instance that reflects the system's artefacts prior to architecture recovery. The PSM metamodel is implemented as an Eclipse Ecore model[18], explained in Chapter 7. Each collected artefact is manually parsed to instantiate a set of corresponding PSM elements, which eventually build up the entire PSM instance model shown in Figure 6-23 that conforms to the PSM metamodel presented in Figure 6-4 and Figure 6-5. Instantiating PSM starts by parsing the Application Project Build File and the Docker Compose File. Then, for each Microservice Project, parsing takes place in the following order: Build file, Docker file and Configurations file. As for parsing the Java Source files, the class with @SpringBootApplication annotation is first processed, followed by the classes with methods annotated with @Scheduled and annotations that end with Mappings, such as @GetMapping, and Listener such as @RabbitListener. This order resembles the starting points of the application's execution. After that, the order of parsing the remaining files does not matter. A PSM instance of Microservices Sample that is created using the Eclipse Ecore model and represented in XMI format is presented in GitHub repository [19].

[18] https://github.com/nuha77/MiSAR/blob/master/PSM.ecore.
[19] https://github.com/nuha77/MiSAR/blob/master/MicroserviceSamplePSM.xmi.

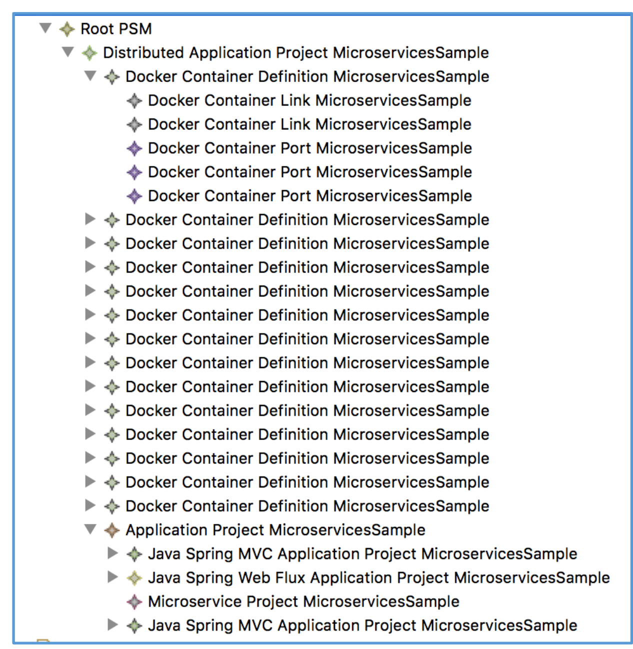


Figure 6-23: Recovered PSM instance of Microservices Sample.

Step 3- Recover PIM instance (automatic): To automate the MiSAR recovery tasks, the PIM metamodel is first implemented as an Ecore model in Eclipse[20]. Mapping rules are then implemented in the QVT language using Eclipse QVTo[21]. In Eclipse QVTo implementation, mapping rules are organised in top-bottom order. In other words, a rule that maps a top PSM element, such as ApplicationProject, to a top PIM element, such as Microservice Architecture, should invoke all mapping rules to recover subsequent PIM elements from PSM elements. The PIM instance recovered is shown in Figure 6-24, and available in the Github repository[22]. The fully recovered PIM instance of Microservices Sample is included in the Eclipse QVTo project, available in the project's Github repository.[23]

[20] https://github.com/nuha77/MiSAR/blob/master/PIM.ecore.
[21] https://github.com/nuha77/MiSAR/blob/master/MisarTransformation.qvto.
[22] https://github.com/nuha77/MiSAR/blob/master/MicroserviceSample.PIM.
[23] https://github.com/MiSAR-A/MiSAR-Eclipse-QVT-Operational/blob/master/MisarQVTv1%20(1).zip.

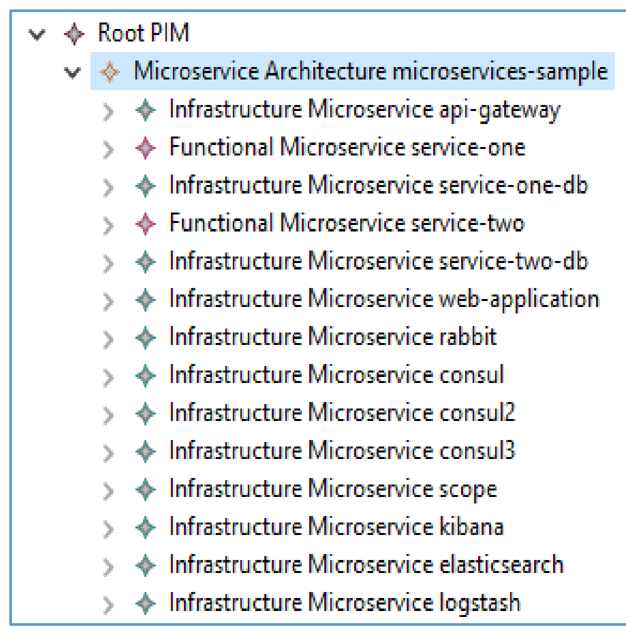


Figure 6-24: Recovered PIM instance of *Microservices Sample.*

Step 4- Visual architectural model (manual): In addition to the PIM instance represented in XML format, MiSAR's output includes an architecture diagram. The recovered architecture in MiSAR is graphically represented at two levels: an architectural level and a microservice level, where each microservice has a more detailed view.

In developing the architecture-level diagram, I was inspired by Sam Newman's (2019) graphical notations for representing high-level abstraction; for example, he uses a hexagon to represent the service and a link to represent the association between two services. For developing the microservice-level diagram I was inspired by the C4 model provided by Simon Brown (2018). The C4 diagrams provide different levels of abstraction, for example containers level, components level, etc. The rationale for using these notations is that they are the most appropriate and the clearest for the presented work, and useful for representing all elements of the PIM metamodel. I customized some diagram notations according to the needs of my project, as outlined in Appendix B-1 and B-4.

Figure 6-25 is an architectural-level diagram of Microservices Sample, which reflects the recovered PIM instance, they include the high-level view of all microservices, their types and dependencies. In the architecture diagram at the architectural level, both the *Infrastructure Microservice* and *Functional Microservice* concepts are represented

with a hexagon; the representation of the *Infrastructure Microservice* is distinguished by adding all of its *Infrastructure Server Components* as circles inside the hexagon in order to specify the composite category of that *Infrastructure Microservice* (see Figure 6-25 and Appendix B,1). A microservice-to-microservice *Service Dependency* is represented by a link connecting the two microservices, while a microservice-group-to-microservice *Service Dependency* is represented by a box surrounding all the microservices that share the same *Service Dependency* with one microservice.

Obviously, these solid lines represent the synchronous interactions between microservices. As for the representation of asynchronous interactions, the set of Queue Listener concepts of all microservices are first mapped to the Queue cylinder. Next, to represent that *service-one* (see Figure 6-25) is publishing messages to a particular queue named *com.mudigal.microservices-sample.service-two*, a dotted line labelled *writes_to_queue* will connect *service-one* to the Queue cylinder labelled with that name. If, alternatively, *service-one* is receiving messages from a queue named *com.mudigal.microservices-sample.service-one*, then a dotted line labelled *reads_from_queue* will connect *service-one* to the Queue cylinder labelled with that particular name. If *service-one* is both publishing to and receiving messages from the same queue, then one dotted line is drawn with two labels, << *writes_to* >> and<< *reads_from* >>. From this representation, one can directly tell that as *service-one* is writing to a queue from which *service-two* is receiving messages, then *service-one* has an asynchronous, indirect Service Dependency on *service-two*.

Figure 6-26 is a microservice-level diagram of microservice service-one in Microservices Sample, which reflects the recovered PIM instance in Figure 6-27, including its Service Interface, Messages Destinations (e.g. Endpoint and/or Queue Listener), Service Messages, Service Operations and the Infrastructure Pattern Components of this individual microservice, the graphical notations related to microservice level outlined in Appendix B-4.

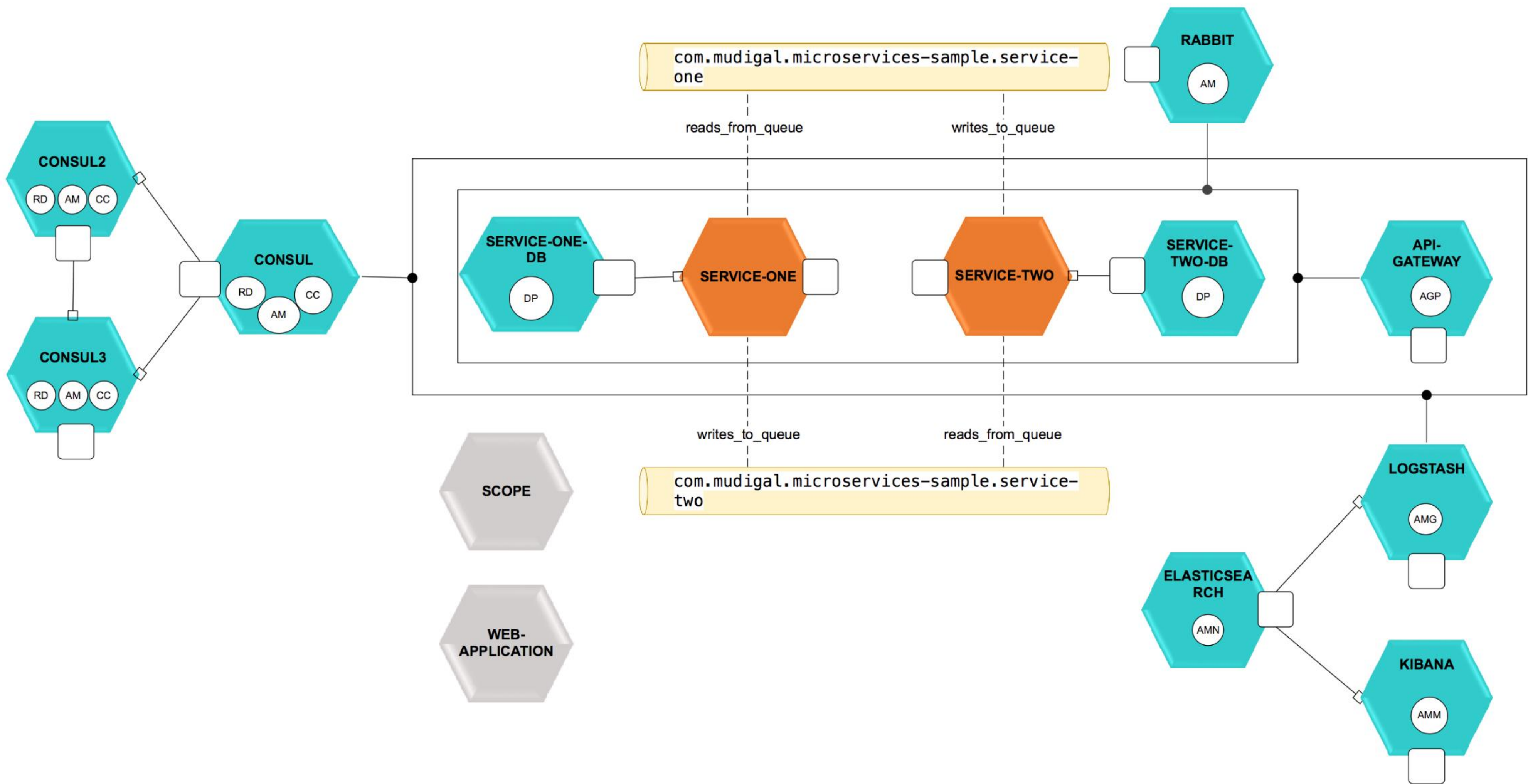


Figure 6-25: Architecture diagram at architectural level of the recovered PIM instance.

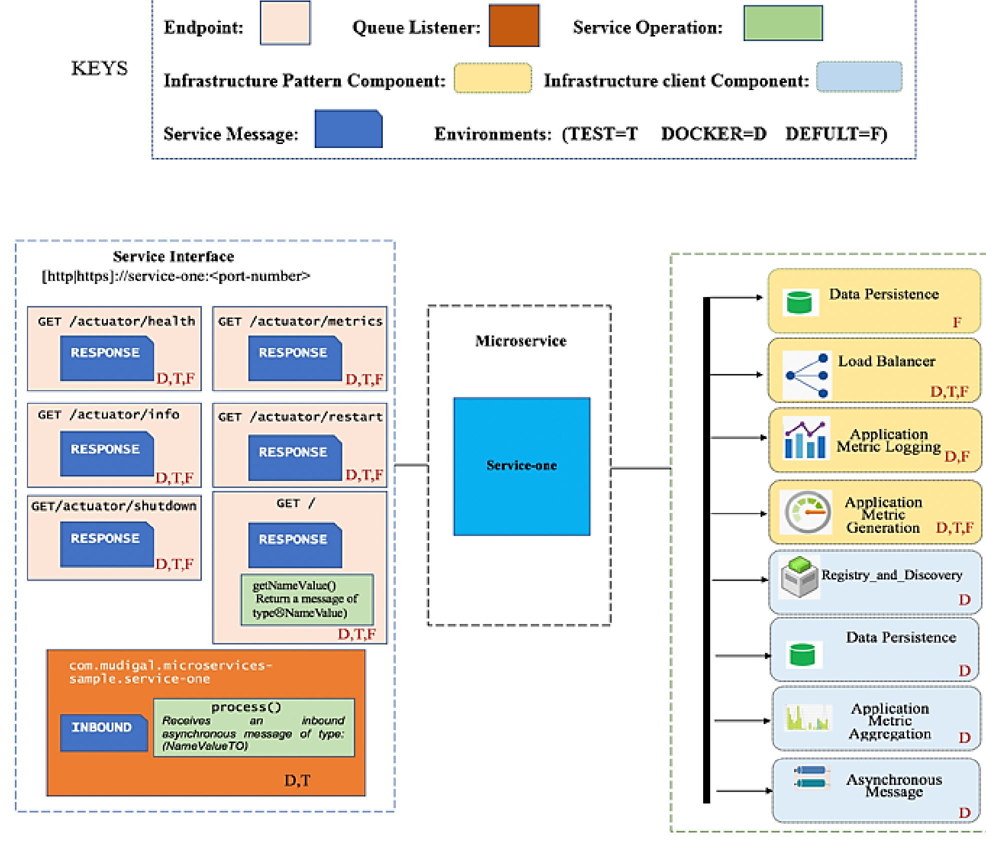


Figure 6-26: Microservice diagram of recovered PIM instance of service-one microservice.

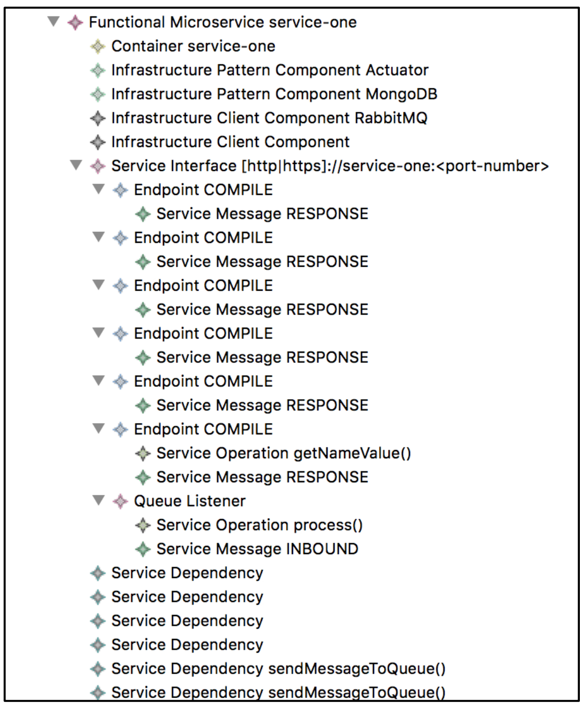


Figure 6-27: Recovered PIM Instance of service-one microservice.

Step 5- Consistency check between generated architectural models and documentation: The architecture diagrams of the Microservices Sample provided by the documentation of the developers in figure 6-28 and 6-29 were compared with the generated MiSAR's architecture diagrams, as in figures 6-25 and 6-26. The results are as follows:

Consistent with documentation: MiSAR successfully recovered two Functional Microservices, *service-one* and *service-two*, in addition to the remaining 12 Infrastructural Microservices (*service-one-db*, *service-two-db*, *rabbit*, *kibana*, *logstash*, *elasticsearch*, *consul*, *consul2*, *consul3, APIGateway, scope* and *web application*). By looking at the documented architecture diagram of the *service-one* microservice provided by the developer, as shown in Figure 6-29, it can be seen that the generated architecture diagram (Figure 6-26) successfully recovered the static components of *service-one* along with their configuration profiles for multiple environments (as explained in Increment-4). To illustrate, the *Embedded MongoDB Database* in the *default* environment in the documented model (Figure 6-26) is recovered as the Infrastructure Pattern Component of the *Data_Persistence* category with environment value *DEFAULT*, while the client component of the remote *MongoDB Database* in the *docker* environment is recovered as the Infrastructure Client Component of the *Data_Persistence* category with environment value *DOCKER*.

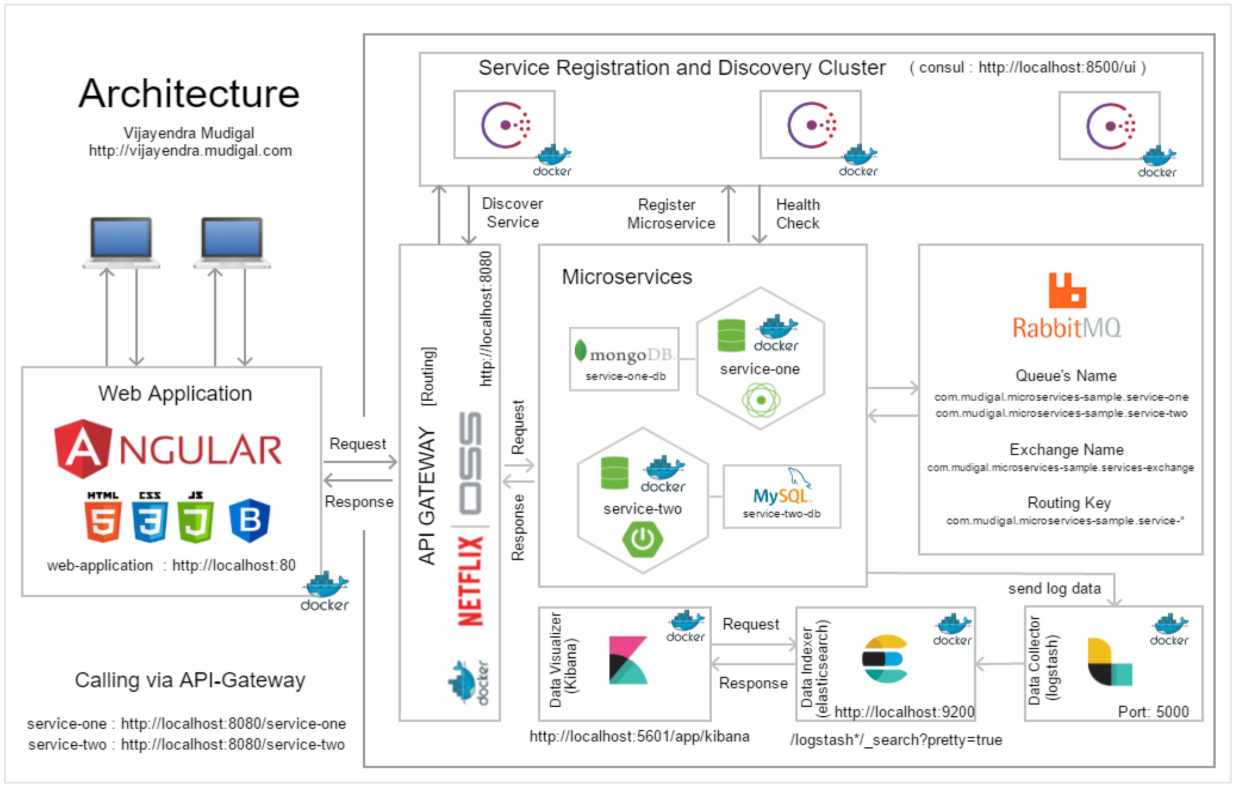


Figure 6-28: Documented architecture diagram of Microservices Sample provided by the developer.

Additional elements: MiSAR also recovered extra Infrastructural Microservice, named *scope*, which is included in the architecture's *Docker Compose* artefact but not represented in the documented diagram.

Missed elements: The dependencies among microservices were successfully recovered, except for the dependencies of the *scope* microservice, as well as the dependencies between the *web-application* and *api-gateway* microservices. The *scope* microservice is an outsourced service available as a remote Docker Hub image, hence it has no local source artefacts to parse and transform, so there were no corresponding hard-coded mapping rules to recover its specifications and dependencies. The *web-application* microservice has local artefacts but they are implemented in *Angular*, i.e. non-Java/Spring, so it doesn't conform to the requirements of the selected studies. Moreover, my PIM model concentrates on the recovery of backend architecture only, therefore the front-end/user interface microservice type, such as *web-application*, is

not covered. It is noticeable that the representation of microservice destinations is totally missed in the documented model compared to the generated model.

Additional expressivity: Compared to the generated model, the documented model demonstrates the direct dependencies of *service-one* and *service-two* on *Rabbit*, but it lacks the representation of the asynchronous functional dependencies between *service-one* and *service-two*. On the other hand, the internal interactions between microservice components are not covered in my model compared to the documented model.

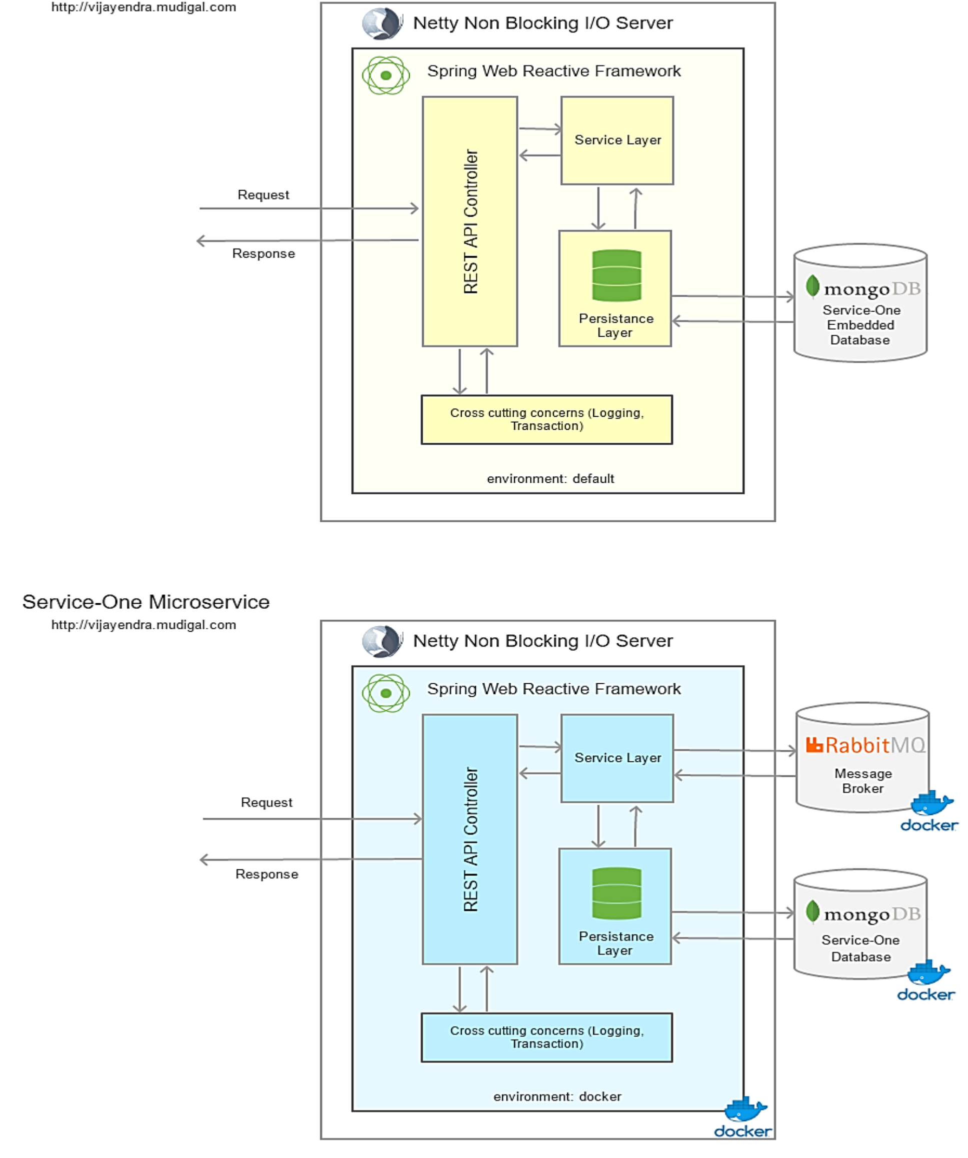


Figure 6-29: Documented architecture diagram of service-one microservice in Microservice sample.

Step 6- Update MiSAR's repository (manually): After the recovery process was completed and the generated model was analysed, some limitations can be seen. This gives rise to executing necessary updates to MiSAR's repository (metamodels and mapping rules). To illustrate, the following mapping rules were added in order to overcome limitations noticed in the recovery of the *scope* microservice identified in the previous step:

1) *One Docker Hub Image Container with Image Field value which contains '**scope**' indicates one Infrastructure Server Component with Category value: Observability Pattern – Application Metrics Monitoring.*
2) *One Docker Hub Image Container with Image Field value which contains '**scope**' indicates one Service Dependency with every microservice in a microservice architecture.*

This particular microservice is a Weave Scope[24] application which is a visualization and monitoring infrastructure for all Docker and Kubernetes containers in a distributed containerised application. It provides a top-down view of microservice-based applications as well as their entire infrastructure, and allows diagnosis of any problems, in real-time.

6.6. Summary

This chapter has presented an in-depth empirical investigation into microservice-based systems with the aim of defining requirements for a metamodel that defines microservice architecture. Through this study, the MiSAR metamodels were enhanced by including the requirements. MiSAR mapping rules were assessed, refined and formalized, with the aim of more efficiently recovering microservice architectures. The metamodels and mapping rules were validated by generating a software

[24] https://www.weave.works/docs/scope/latest/introducing/.

architecture model of an unanalysed software system and checking its conformance to the documentation to ensure the expressivity of the approach. In the next chapter, I present the implementation of MiSAR repository of metamodels and mapping rules and chapter 8 present the evaluation of MiSAR in the context of a large case study.

Chapter 7

MiSAR Metamodels and Mapping Rules: Features and Implementation

7.1. Introduction

This chapter provides an overview of the MiSAR artefacts developed with the help of the Eclipse framework[25] to offer assistance in the recovery of microservice architecture. Section 7.2 provides an introduction to the modelling language and model transformation that were used. Ecore, the metamodel implementation, is discussed in section 7.3. QVT, the transformation mapping rules implementation, is discussed in section 7-4. Finally, section 7-5 discusses the mapping rule features.

7.2. MiSAR Implementation Environment

The Eclipse Modelling Framework (EMF) forms the environment within which the metamodels and mapping rule transformations are developed. Metamodels were implemented as Ecore models using the EMF. The tools utilised for the development and execution of transformations rules is the Eclipse Model-to-Model Transformation (M2M) the sub-project of EMF, by incorporating the operational QVT transformation language (QVTo). Figure 7.1 presents the relationship between transformation rules, models, metamodels.

[25] http://www.eclipse.org.

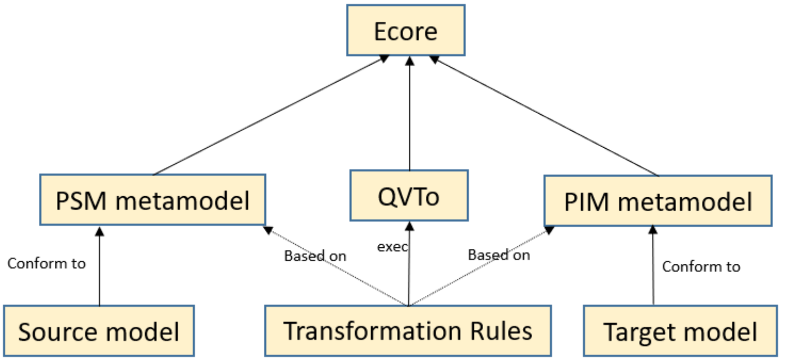


Figure 7-1: The relationship between transformation rules, models, metamodels.

7.3. Ecore Metamodel Implementation

Ecore is one of the important components of the EMF (Steinberg et al., 2008(Dave et al., 2008); Barendrecht, 2010), which is a graphical domain-specific language used for representing models, such as Unified Modelling Language (UML). MiSAR metamodel implementation utilises only a few parts of Ecore, the subset of which is illustrated in Figure 7-2. The figure illustrates four classes: EClass, EReference, EAttribute and EDatatype. EClass is a model class and has a specific name, 0 or more attributes. EReference establishes a one-way linkage between two distinct EClasses, which are termed source EClass and target EClass. The EReference subset in the figure embraces name, lower bound, upper bound and isContainment. Both bounds (upper and lower) in this figure are referred to the cardinality constraints. Two dots are used for classifying bounds; the right part indicates the upper bound, while the left indicates the lower bound. An asterisk (*) is used in the situation of an unlimited upper bound. A single number is used to visualise equal lower and upper bounds. The representation of EAttribute in the figure is a modelled attribute that also includes its name and the two bounds (upper and lower). EAttribute conforms to a specific type, an EData Type, which is an integer or string or object type.

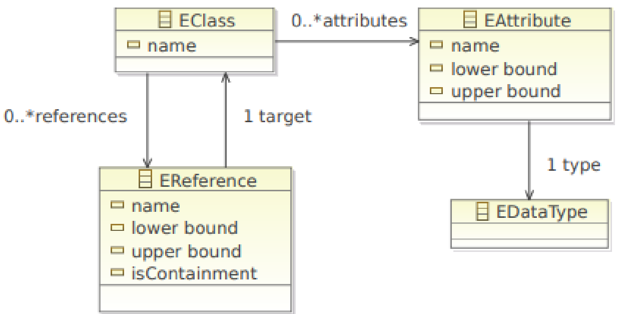


Figure 7-2: The four classes of Ecore used in the Ecore implementation (Barendrecht, 2010).

Metamodels have been implemented as Ecore models using the Eclipse Modeling Framework (EMF) (Steinberg et al., 2008) as shown in Figures 7-3 and 7-4, which present the MiSAR PSM[26]/PIM[27] metamodels and their properties respectively in XMI tree view. Figures 7-5 and 7-6 present the Ecore implementation of MiSAR PSM/PIM metamodels respectively in Ecore diagram.

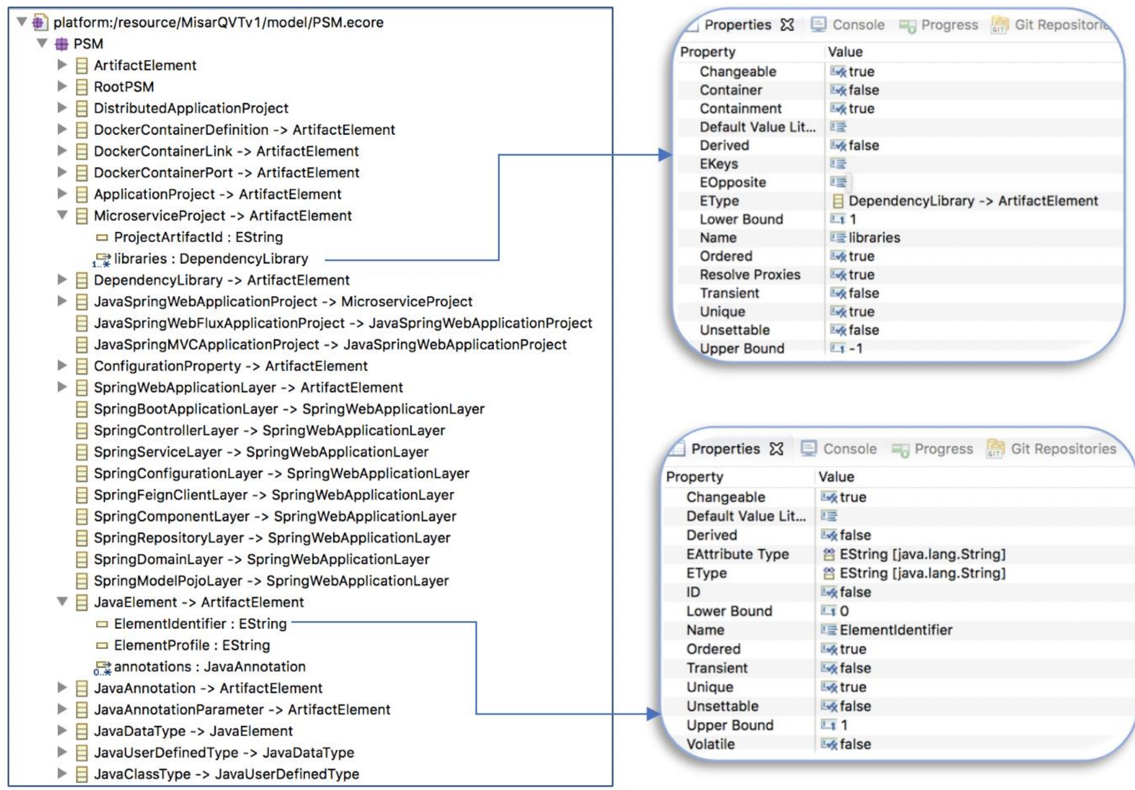


Figure 7-3: Ecore implementation (XMI tree view) of MiSAR PSM.

[26] https://github.com/nuha77/MiSAR/blob/master/PSM.ecore.
[27] https://github.com/nuha77/MiSAR/blob/master/PIM.ecore.

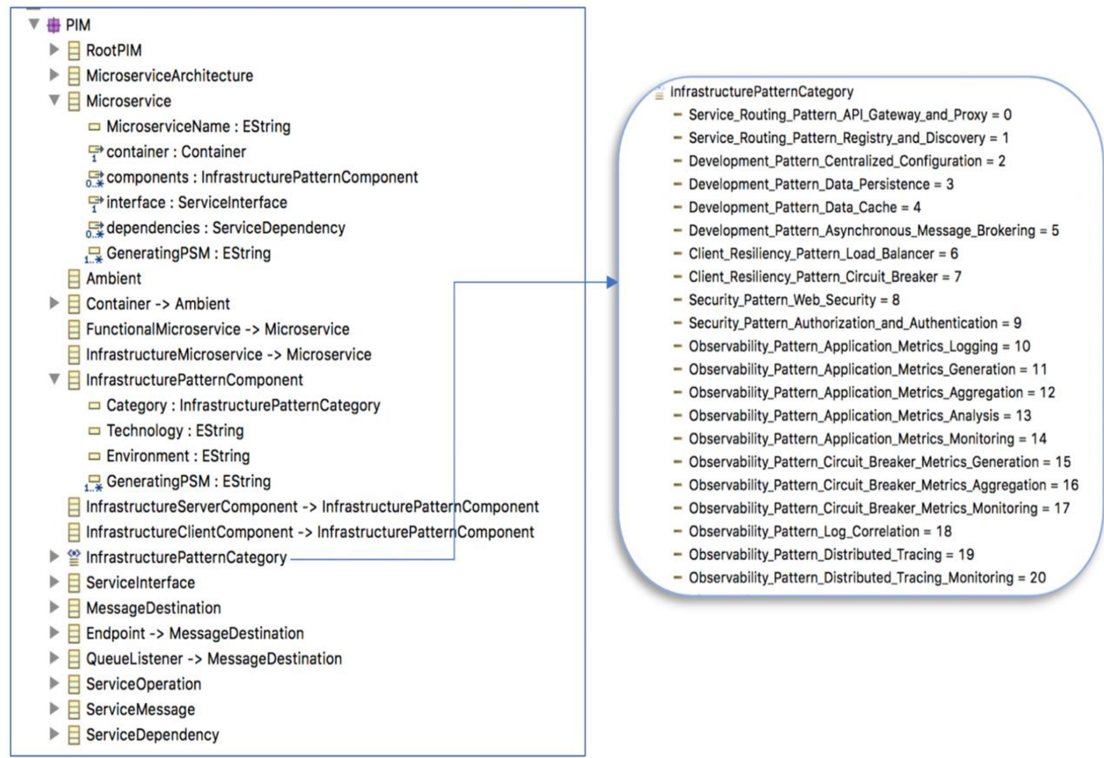


Figure 7-4: Ecore implementation (XMI tree view) of MiSAR PIM.

7.4. QVT, the Transformation Rules Implementation

To develop and automate the mapping rules I utilised the Eclipse Model-to-Model Transformation (M2M) project, by incorporating the operational QVT transformation language (QVTo) (Barendrecht, 2010). The following subsection outlines the main components of this language.

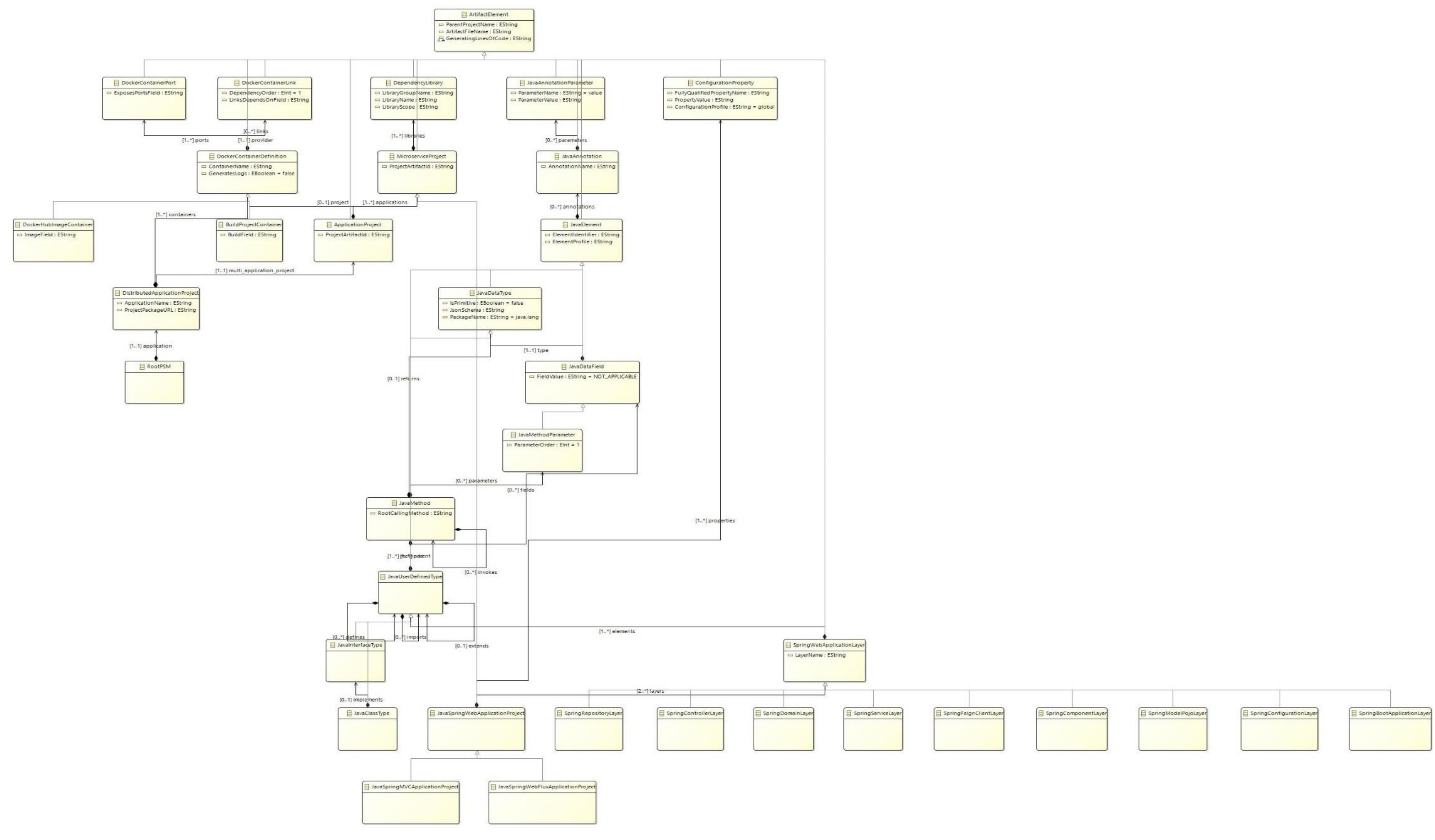


Figure 7-5: Ecore implementation (Ecore diagram) of MiSAR metamodel at PSM level.

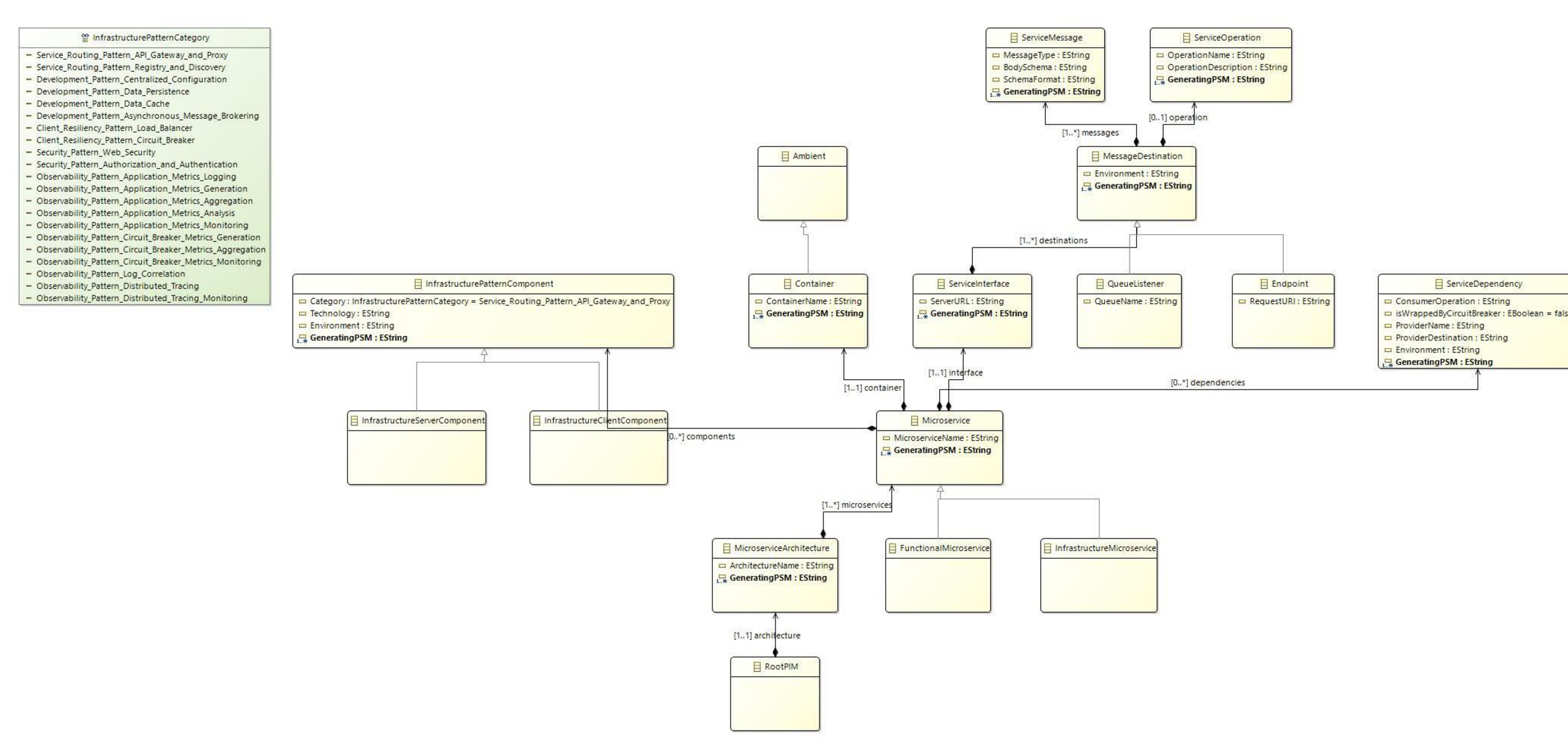


Figure 7-6: Ecore implementation (Ecore diagram) of MiSAR metamodel at PIM level.

7.4.1. Model Type Definitions

Model type is a reference to a metamodel; entire metamodels can be included in the transformation (in-line definition) or have the feature of referring the defined models externally. The following (listing 7-1) is an example of model type definitions for MiSAR transformation.

Listing 7-1: Model type definitions for MiSAR transformation.

```
modeltype PSM uses PSM('http://localhost/mdd/PSM.ecore');
modeltype PIM uses PIM('http://localhost/mdd/PIM.ecore');
```

7.4.2. Transformation Declaration

The declaration of transformation rules includes a name, input and output metamodel. The metamodels of input and output are referred to as defined model types. The following (listing 7-2) is transformation declaration for MiSAR transformation.

Listing 7-2: Transformation declaration for MiSAR transformation.

```
transformation MisarTransformation(in source: PSM, out target: PIM );
```

7.4.3. Main Function

The beginning of the transformation is considered with the starting of the main () function. The main function is subjected to establish environment variables and also by calling for the first mapping rule. The following (listing 7-3) present the main function of MiSAR transformation.

Listing 7-3: Main function of MiSAR transformation

```
main() {
	source.rootObjects()[RootPSM] -> map RootPSM2RootPIM();
}
```

7.4.4. Mapping

The core value of the QVT transformation is mapping. It is ensured by each mapping that an object from an instance of the source metamodel has to be transformed into an object that is basically a created instance of the target metamodel, it is bound to be invoked with the keyword *map.*

1. **Declaration:** The mapping declaration involves indulging the input class name of the object along with the name of the mapping, and it also includes the generated class name of the object, which is an outcome of the mapping. The complete mapping rules for transformation are available in the GitHub repository.[28]Example of the first mapping rule in MiSAR implementation is given in Listing 7-4. This rule maps a `RootPSM` into `RootPIM` as well as invoking two mapping rules.

Listing 7-4: RootPSM2RootPIM() mapping rule

```
// declaration section
mapping RootPSM:: RootPSM2RootPIM(): RootPIM
// pre-condition section
when {self.application <> null} {

// body section
   // population section
   architecture := self.application.map DistributedApplicationProject2MicroserviceArchitecture();
   architecture := self.application.application_project.map
ApplicationProject2MicroserviceArchitecture();
   // populate references
   // population section
   // end section
}
```

2. **Conditions:** Certain conditions can be used for the extension of the mapping declaration. It is also required for the source object to adapt these conditions with the use of defined Object Constraint Language (OCL) in order to successfully execute the mapping. There are two types of conditions, the pre- and post-condition, indicated with the keywords “where” and “when”. An example of pre-condition with “when” is illustrated in Listing 7-4. It asserts

[28] https://github.com/nuha77/MiSAR/blob/master/MisarTransformation.qvto.

that a DistributedApplicationProject must exist in the source model before executing the mapping.

3. **Body:** Population, end and init are identified to be the three sections of the mapping body. The init-section is primarily used for initialising parameters and variables as well as for printing messages to the terminal. The population section is found to be the section that specifies actual mapping. The idea of the end section is incorporating additional code executed prior to leaving the mapping. Listing 7-5 demonstrates the ApplicationProject2 MicroserviceArchitecture mapping invoked in Listing 7-4. This mapping maps an ApplicationProject into MicroserviceArchitecture. Every **init** section in MiSAR mapping rules will check whether the target element (e.g. MicroserviceArchitecture) has been mapped before or not. This check is necessary as there are many mappings that generate the same target element (e.g. as the invoked mappings in Listing 7-4). The population section will populate target attributes by assigning values to read from source attributes, then populate target references, if any, by invoking other mappings.

Listing 7-5: ApplicationProject2MicroserviceArchitecture() mapping rule

```
// declaration section
mapping ApplicationProject:: ApplicationProject2MicroserviceArchitecture(): MicroserviceArchitecture {

// body section
  // init section
  init {
    var architecture:= MicroserviceArchitecture.allInstances()->any(_architecture|_architecture<>
null);
    // if any Microservice Architecture was mapped before, update result, otherwise, create a new one.
      if architecture <> null then {
        result:= architecture;
      } endif;
  }
  // population section
  ArchitectureName := self.ProjectArtifactId;
  GeneratingPSM += 'ApplicationProject[ProjectArtifactId:'+self.ProjectArtifactId+']';
  // populate attributes
  microservices += self.modules.map MicroserviceProject2Microservice();
  // populate references
  // end section
}
```

4. **Parameters:** The mapping can be parameterised. Two examples of parameterised mapping rules and how they use the values of their parameters are illustrated in listings 7-7 and 7-8. Listings 7-6 and 7-7 present two examples of how mappings invoke the parameterised mappings.

Listing 7-6: MicroserviceProject2ServiceInterface() mapping rule

```
// declaration section
mapping MicroserviceProject:: MicroserviceProject2ServiceInterface(): ServiceInterface {

// body section
  // init section has been removed
  // population section
  ServerURL := '[http|https]://'+self.ProjectArtifactId+':<port-number>';
  GeneratingPSM += 'MicroserviceProject[ProjectArtifactId:'+self.ProjectArtifactId+']';
  // populate attributes
  self.libraries->forEach(_library){
       if _library.LibraryName = 'spring-boot-starter-actuator' then {
         destinations += _library.map DependencyLibrary2Endpoint('GET /actuator/health');
         // some code has been removed
       }endif;
  };
  // some code has been removed
  self.properties->forEach(_property){
       if _property.FullyQualifiedPropertyName = 'endpoints.health.sensitive' then {
         destinations += _property.map ConfigurationProperty2Endpoint('GET /actuator/health');
         // some code has been removed
       }endif;
  };
  // some code has been removed
  // populate references
  // end section
}
```

Listing 7-7: DependencyLibrary2Endpoint() mapping rule

```
// declaration section
mapping DependencyLibrary:: DependencyLibrary2Endpoint(uri: String): Endpoint {

// body section
  // init section has been removed
  // population section
  RequestURI := uri;
  Environment := self.LibraryScope;
  GeneratingPSM += 'DependencyLibrary[LibraryName:'+self.LibraryName+']';
  // populate attributes
  if uri = 'GET /actuator/health' then {
    messages += self.map DependencyLibrary2EndpointServiceMessage(uri, 'RESPONSE',
    '
       {
         "type": "object",
```

```
          "properties": {
            "status": {
              "type": "string"
            },
            "details": {
              "type": "object"
            }   }
      ', }
      'JSON');
  }endif;
  // some code has been removed
  // end section
}
```

Listing 7-8: DependencyLibrary2EndpointServiceMessage() mapping rule

```
// declaration section
mapping DependencyLibrary::
        DependencyLibrary2EndpointServiceMessage(endpoint: String, type: String, schema: String, format:
String):
        ServiceMessage {

// body section
  // init section has been removed where the value of endpoint parameter is used
  // population section
    MessageType := type;
    BodySchema := schema;
    SchemaFormat := format;
    GeneratingPSM += 'DependencyLibrary[LibraryName:'+self.LibraryName+']';
  // populate attributes
  // end section
}
```

5. **Inheritance:** Mappings can be reused in other mappings, one way to achieve this is inheritance. Inheritance is the transformation of subclasses in the source model, transforming into subclasses in the target model or both. A subclass is acquired by inheritance from one superclass, which is naturally represented graphically by an arrow with a white head in the Ecore. An inherited mapping is declared with keyword inherits and the name of the super mapping. An example of an inherited mapping is given in Listing 7-9, and the super mapping is given in Listing 7-10. The super mapping is executed first. Then, the target element will be updated and generated by the inherited mapping.

Listing 7-9: DockerContainerDefinition2InfrastructureMicroservice() mapping rule

```
// declaration section
mapping  DockerContainerDefinition::
         DockerContainerDefinition2InfrastructureMicroservice():
         InfrastructureMicroservice
// inherits declaration
inherits DockerContainerDefinition:: DockerContainerDefinition2Microservice {
}
```

Listing 7-10: DockerContainerDefinition2Microservice() mapping rule

```
// declaration section
mapping DockerContainerDefinition:: DockerContainerDefinition2Microservice(): Microservice {

// body section
  // init section has been removed
  // population section
  MicroserviceName := self.ContainerName;
  GeneratingPSM += 'DockerContainerDefinition[ContainerName:'+
                   self.ContainerName+',ImageField:'+
                   self.ImageField+', GeneratesLogs:'+
                   self.GeneratesLogs.toString()+']';
  // populate attributes
  container := self.map DockerContainerDefinition2Container();
  // some code has been removed
  if self.ImageField.indexOf('consul') <> 0 then{
    components += self.map DockerContainerDefinition2InfrastructureServerComponent(
                  InfrastructurePatternCategory::Development_Pattern_Asynchronous_Message_Brokering,
                  'Consul');
    components += self.map DockerContainerDefinition2InfrastructureServerComponent(
                  InfrastructurePatternCategory::Service_Routing_Pattern_Registry_and_Discovery,
                  'Consul');
    components += self.map DockerContainerDefinition2InfrastructureServerComponent(
                  InfrastructurePatternCategory::Development_Pattern_Centralized_Configuration
                  'Consul');
  }endif;
  // some code has been removed
  interface := self.map DockerContainerDefinition2ServiceInterface();
  self.links->forEach(_link){
    dependencies += _link.map DockerContainerLink2ServiceDependency()
  };
  // some code has been removed
  // populate references
  // end section
}
```

6. **Resolving:** The situation in which an object of the source model is transformed to an object belonging to the target model, it results into the appearing of source object just as a reference, which is not bound to be transformed again. In Eclipse QVTo, this is accomplished using the resolve function. There are four variants of the resolve function and these are concisely explained in Table 7-1. The resolve function is essentially used to resolve a source to a target and

returns the target object which appeared from mapping the source object. An example of a mapping that uses the resolve function is given in Listing 7-11.

Table 7-1: Resolve functions (Barendrecht, 2010).

resolve	Returns a set of target objects. These objects are the result of an earlier mapping from the source object on which the resolving is being applied.
resolveone	Returns one target object. If the mapping of the source object resulted in multiple target object, only the last is returned.
resolveIn	Like resolve, however only the target objects that were mapped in a specific mapping are returned. The name of the mapping is specified by a parameter.
resolveoneIn	Like resolveIn, one target object created in a specific mapping is returned.

Listing 7-11: ConfigurationProperty2ServiceInterface() mapping rule

```
// declaration section
mapping ConfigurationProperty:: ConfigurationProperty2ServiceInterface(): ServiceInterface {

// body section
  // init section has been removed
  // population section
  ServiceURL := := '[http|https]://'+self.PropertyValue+':<port-number>';
  GeneratingPSM += 'ConfigurationProperty[FullyQualifiedPropertyName:'+
                   self.FullyQualifiedPropertyName+
                   ']';
  self.container().oclAsType(JavaSpringWebApplicationProject).properties->forEach(_property){
    if _property.FullyQualifiedPropertyName = 'server.contextPath'
       and _property.PropertyValue <> '/' then {
         // resolving mapped target
         var mapped_interface := self.resolveone(ServiceInterface);
         // update attributes of mapped target rather than creating a new one
         mapped_interface.ServerURL := ServerURL + _property.PropertyValue;
         mapped_interface.GeneratingPSM += 'ConfigurationProperty[FullyQualifiedPropertyName:'+
                                          _property.FullyQualifiedPropertyName+']';
    } endif;
  };
  // populate attributes
  // end section
```

7.5. Mapping Rule Features

MiSAR mapping rules have been designed to follow the taxonomy of model transformations presented in (Mens and Van Gorp, 2006) and the explanation of mapping rules features are based on model transformation features presented in (Czarnecki and Helsen, 2003) as follows:

- **MiSAR Mapping Rule Structure:**

Mapping rules are represented as a group of source PSM elements at the left-hand side (LHS), specified by their attributes' values and references among them, which transforms to a group of target PIM elements at the right-hand side (RHS), with specific attributes values and references among them. Each mapping rule conforms to the metamodel depicted in Chapter 6, Figure 6-22.

To explain how an instance of the mapping rule metamodel is created, I will use the mapping rules implemented in listings 7-6, 7-7 and 7-8, which state that:

- A *Dependency Library* with *Library Name* value: "**spring-boot-starter-actuator**" and *Library Scope* value: *{destination-environment}* indicates one *Endpoint* with *Request URI* value: "**GET /actuator/health**" and *Environment* value: *{destination-environment}* which has a *Service Message* with *Message Type* value: "**RESPONSE**", *Schema Format* value: "**JSON**" and *Body Schema* value:

```
"{
"type": "object",
"properties": {
"status": {
"type": "string"
},
"details": {
"type": "object"
}
}
}"
```

The instance of this mapping rule is provided in Figure 7-8. Rule Identifier (RID) is a unique value recorded for each mapping rule. Source Microservice Name and Source Project Name belong to a particular microservice (e.g. `account-service`) in a particular case study (e.g. `piggymetrics`), respectively, where the mapping rule was first observed. Source Artefact Type indicates that this rule was extracted from a certain artefact type (e.g. `BuildFile`). Source Artefact Filename refers to the fully qualified filename of a particular artefact where the given mapping rule was first

observed. Source Generating Snippet is the first text/code snippet in the artefact indicating the given mapping rule. The textual mapping rule written in natural language is provided in the Description.

To return to the mapping rule represented in Figure 7-7; this rule is originally extracted from the source snippet taken from the POM build artefact of microservice "**account-service**", as presented in Figure 7-8. Lines 52 and 53 have been parsed into a PSM Element: **DependencyLibrary** with attributes: [**LibraryGroup** = 'org.springframework.boot'] (line 52), [**LibraryName** = 'spring-boot-starter-actuator'] (line 53) and [**LibraryScope** = 'COMPILE'] (the default value of '<scope>' element in '<dependency>' element if it is missing). This source PSM element will be transformed into two target PIM elements associated with each other: an **Endpoint** element with hard coded attributes [**RequestURI** = 'GET /actuator/health'] and [**Environment** = 'COMPILE'] associated by the reference '**messages**' to a **ServiceMessage** element with hard coded attributes [**Type** = 'REQUEST'], [**Schema** = '{...}'] and [**SchemaFormat** = 'JSON']. This transformation performed if and only if the **DependencyLibrary** has the attributes **LibraryName** with particular value 'spring-boot-starter-actuator'.

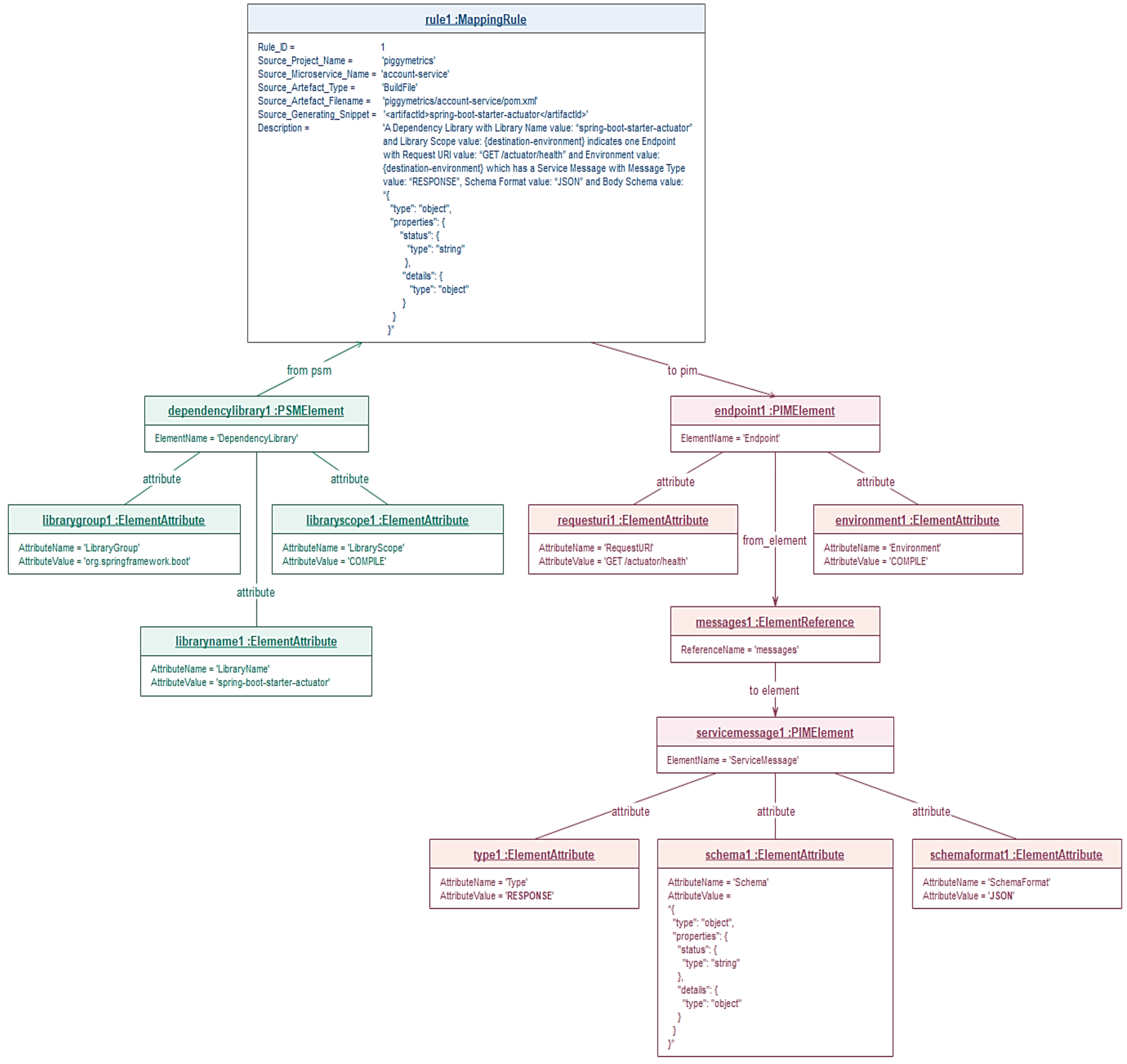
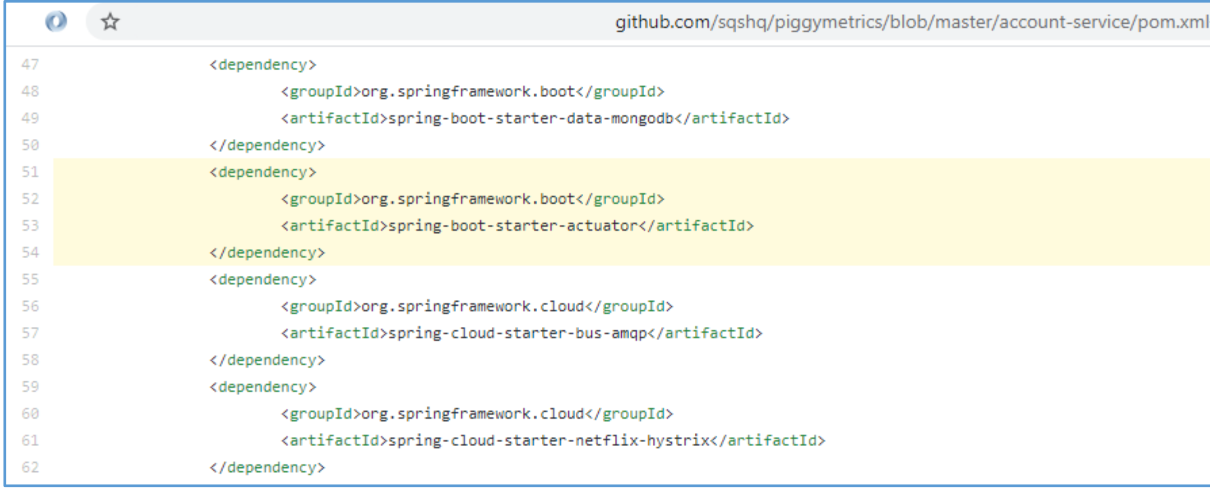


Figure 7-7: A mapping rule instance.


github.com/sqshq/piggymetrics/blob/master/account-service/pom.xml

```
47		<dependency>
48			<groupId>org.springframework.boot</groupId>
49			<artifactId>spring-boot-starter-data-mongodb</artifactId>
50		</dependency>
51		<dependency>
52			<groupId>org.springframework.boot</groupId>
53			<artifactId>spring-boot-starter-actuator</artifactId>
54		</dependency>
55		<dependency>
56			<groupId>org.springframework.cloud</groupId>
57			<artifactId>spring-cloud-starter-bus-amqp</artifactId>
58		</dependency>
59		<dependency>
60			<groupId>org.springframework.cloud</groupId>
61			<artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
62		</dependency>
```

Figure 7-8: Lines in POM file of the "account-service" project that originated the mapping rule.

The same target PIM element is transformed by alternative mapping rules also implemented in listings 7-6, 7-7 and 7-8, and states:

- A *Configuration Property* with *Fully Qualified Property Name* value: "**endpoints.health.sensitive**" and *Configuration Profile* value: *{destination-environment}* indicates one *Endpoint* with *Request URI* value: "**GET /actuator/health**" and *Environment* value: *{destination-environment}* which has a *Service Message* with *Message Type* value: "**RESPONSE**", *Schema Format* value: "**JSON**" and *Body Schema* value:

```
"{
  "type": "object",
  "properties": {
    "status": {
      "type": "string"
    },
    "details": {
      "type": "object"
    }
  }
}"
```

The instance for this alternative mapping rule is provided in Figure 7-9. This rule is originally extracted from the source snippet taken from the YAML configuration artefact of microservice "`service-one`", as presented in Figure 7-10. Lines 13, 18 and 19 have been parsed into a PSM element: `ConfigurationProperty` with attributes: [`FullyQualifiedPropertyName` = 'endpoints.health.sensitive'] (lines

13, 18 and 19), [**`PropertyValue`** = ‘false’] (line 19) and [**`ConfigurationProfile`** = ‘COMPILE’] (the default value of ‘spring.profiles’ configuration if it is missing). This source PSM element will be transformed into the same target element discussed previously.

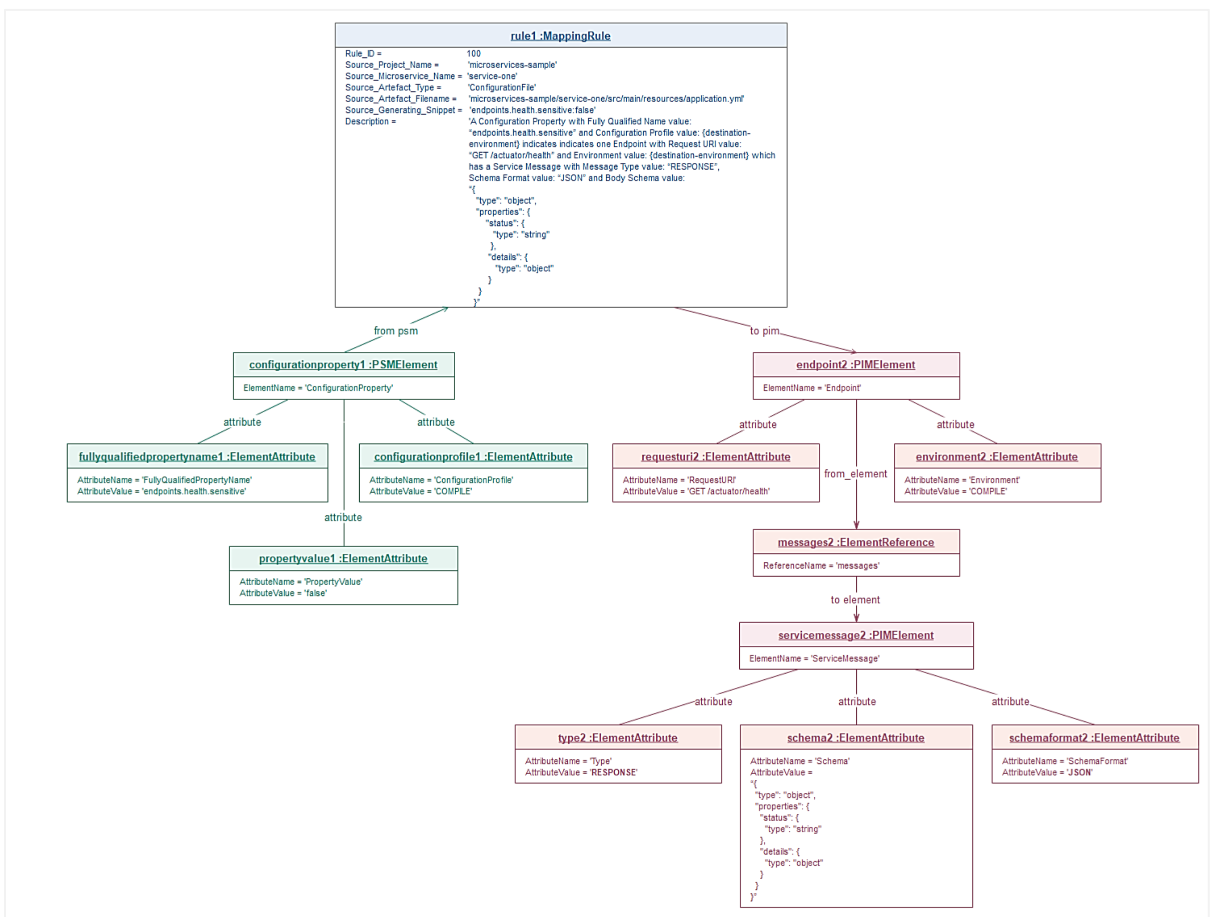


Figure 7-9: An alternative mapping rule instance.

```
github.com/microservices-sample/service-one/blob/master/src/main/resources/application.yml
6    jackson.serializationindent_output: true
7
8  info:
9    component: Service-One
10   artifact: service-one
11   version: 1.0.0
12
13 endpoints:
14   restart:
15     enabled: true
16   shutdown:
17     enabled: true
18   health:
19     sensitive: false
20
```

Figure 7-10: Lines in configuration file of the “service-one” project that originated the alternative mapping rule.

- **Rules Application Scoping**: MiSAR PSM selection approach is considered partial coverage. Not all elements of the PSM model were involved in the transformation. Instead, I selected a set of them based on particular attribute values and/or references to other model elements. This is done in order to provide the most relevant information for architecture recovery. For example, among all JavaMethod elements in a Java source artefact that are parsed into PSM elements, I only selected those methods that are decorated with particular annotations, such as @RequestMapping, @RabbitListener etc., to be later transformed into ServiceDestination PIM, i.e. Endpoints and QueueListener, respectively.

- **Rules Application Strategy:** The determination aspects have been bifurcated into location and scheduling where location determination ascertains the model locations in which the transformation rules are exploited potentially. While, scheduling determination deals with the orders which are subjected to execute the transformation rules. The mapping rules follows both location determination and scheduling determination as they are written in depth-first style with respective to PIM metamodel as well as to PSM metamodel. For example, the rule that maps the MicroserviceArchitecture element, which is the first element in the PIM metamodel, from the DistributedApplicationProject element, which is the first element in the PSM metamodel, is planned to execute first. Next will be the rule retrieving the Microservice element, which is the first ancestor of the MicroserviceArchitecture element in the PIM metamodel, from the DockerContainerDefinition element, which is the first ancestor of the DistributedApplicationProject element in the PSM metamodel, and so on.

- **Traceability:** With the use of Eclipse QVT Operational, traceability is achieved. During the process of QVT transformation, a trace is encountered and simultaneously recorded for every source element which is transferred by a mapping rule. These traces can assist the developers in analysing the orders in which mappings were invoked. Traces in QVT are invoked by “resolving”, as illustrated in Listing 7-11

- **Directionally:** Currently, MiSAR transformations are implemented using Eclipse Operational QVTo, which are uni-directional from source to target.

- **Rule application scoping**: MiSAR PSM selection approach is considered partial coverage. Not all elements from source artefacts were involved in the transformation. Instead, I selected a set of them based on particular attribute values and/or references to other model elements. This is done in order to provide the most relevant information for architecture recovery. For example, among all JavaMethod elements in a Java source artefact that are parsed into PSM elements, I only selected those methods that are decorated with particular annotations, such as @RequestMapping, @RabbitListener etc., to be later transformed into ServiceDestination PIM, i.e. Endpoints and QueueListener, respectively.

- **Mapping Rules Classification**: MiSAR mapping rules can be classified into the following types depicted in Figure 7-11:

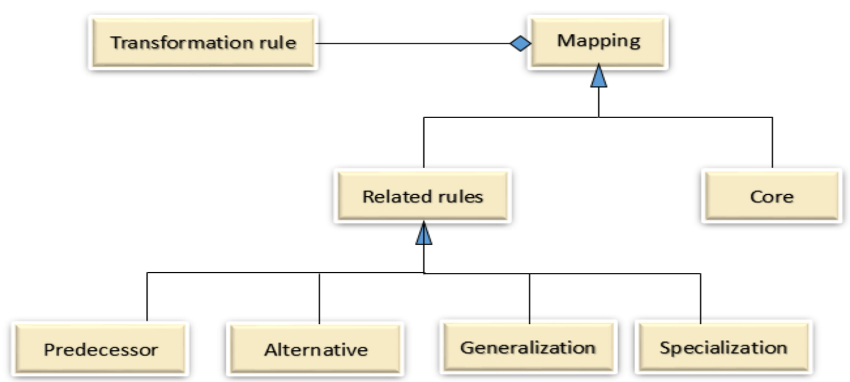


Figure 7-11: MiSAR mapping rules classification.

a) **Core rules:** are those rules that must exist to retrieve a concept at the PIM level, i.e. they always apply and do not have alternative rules. An example of such rule is the one implemented with QVTo in Listing 7-4, because there is no other rules that can retrieve the root **RootPIM** concept. All other mappings have alternatives.

b) **Alternative rules:** are those rules that give additional possibilities to represent the same concept at the PIM level. Examples include the two mapping rules: *DependencyLibrary2Endpoint()* and *ConfigurationProperty2Endpoint(),* they both retrieve the exact PIM concept **Endpoint** using different PSM concepts, i.e. **DependencyLibrary** and **ConfigurationProperty** as in Listings 7-6, 7-7.

c) **Predecessor rules:** are those rules that trigger the need to the current rule. Basically, rules that retrieve PIM concepts that are ancestors to other PIM concepts are considered predecessor rules. In other words, predecessor rules invoke other rules and/or construct different PIM concepts inside them. For example, the mapping rules that recover **ServiceInterface** in Listing 7-6, invokes mapping rules to recover **Endpoint** e.g. *DependencyLibrary2Endpoint(),* and this mapping rule invoke another mapping rules to recover **ServiceMessage**.

d) **Generalization rules**: a rule that is general used to represent the corresponding PIM concept. Rule *DockerContainerDefinition2Microservice ()* is one example of abstract general mapping rule implemented in Listing 7-10.

e) **Specialization rules**: those rules that are a specialized form of a particular rule. One example is the rule implemented in Listing 7-9, as it inherits from the rule implemented in Listing 7-10.

7.6. Summary

In this chapter, I have explained the language used to define MiSAR metamodels and the language used to implement the mapping rule transformations. PIM/PSM metamodels were implemented as Ecore models using the Eclipse Modelling Framework (EMF), while mapping rules were implemented as QVTo mappings using Eclipse M2M/QVT Operational. I ended this chapter with an elaboration on the features of MiSAR mapping rules.

Chapter 8

An Evaluation of MiSAR Artefacts through Microservice Architecture Recovery: A Case Study

8.1. Introduction

The objective of this chapter is to evaluate and demonstrate the usefulness of the MiSAR artefacts (metamodels and mapping rules), introduced in chapters 5 and 6, by applying MiSAR artefacts to the recovery of a case study system architecture. Section 8.2 presents the evaluation methodology, which includes architecture recovery that takes platform-specific models (PSM) concerning the system and obtains platform-independent models (PIM) to represent the target model. Section 8.3 presents the case study, which involves a realistic simulation of a real system that contains 69 microservices (out of which 41 are business-oriented microservices). The case study measures the efficiency and effectiveness of the MiSAR technique.

8.2. Evaluation Methodology

In order to evaluate MiSAR artefacts, first I followed the steps depicted in Figure 8-1, to recover the architecture model of the case study system's architecture. I then evaluated the results obtained through manually checking the consistency between the recovered architecture model by MiSAR artefacts and the system's architecture documentation. Finally, the MiSAR repository (metamodels and mapping rules) is updated with any new elements. Figure 8-1 illustrates the different parts of the architecture recovery process; the thick arrows signify the process in the recovery process; the boxes represent the different forms of information in the recovery process; the thin arrows indicate the inputs and output of the transformation engine. The recovery process consists of three steps that can be summarised as follows:

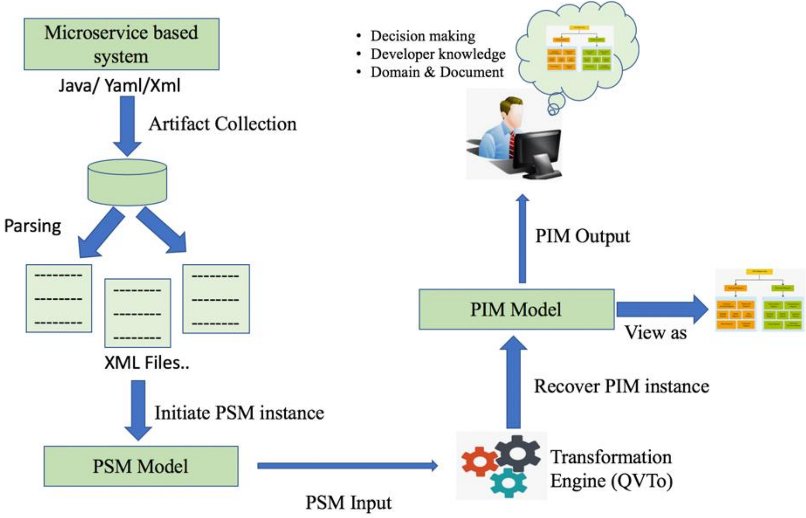


Figure 8-1: Steps of the MiSAR architecture recovery process.

Step 1 – Artefact collection (semi-automatic): This step involves collecting artefacts and reviewing them to search for any that may give information about the system. There are several artefacts in a microservice-based application, such as Java source files, Docker Compose files, Docker files, Build files and Configuration files. These are gathered together in an effort to build the knowledge base for the software system.

Step 2 – Instantiate PSM instance (automatic): This phase produces the information required to describe the software architecture. It extracts the static elements from the system's source code and other artefacts, and eventually generates a PSM that conforms to the PSM metamodel (presented in Chapter 6). This step is executed using the MiSAR parser (Fakeeh and Alshuqayran, 2019), which generates a PSM instance in XMI format that is readable by the Eclipse QVTo project. The path and/or directory of each required artefact is entered by the user as input to the parser.

Step 3 – Recover PIM instance (automatic): This phase populates the target model with a high-level abstraction of the system by applying the automated mapping rules implemented using the Eclipse QVTo project. The output of this stage is the architecture PIM that conforms to the PIM metamodel (presented in Chapter 6). The resulting PIM is in XMI format and is viewed in Eclipse QVTo as a tree of elements.

8.3. Case Study

This case study follows the protocols formulated by Brereton et al. (Brereton et al., 2008) to increase validity and reliability. The details of the case study according to these protocols, such as design, selection of the case, procedure, data collection and data analysis, are presented in the next section.

8.3.1. Design

The objective of the case study is to evaluate the MiSAR approach in terms of recovering an architectural model of a large system. In this regard, I applied the recovery process and made use of the implemented MiSAR artefacts: Ecore metamodels, QVTo Model Transformations and Parser. The case study was adopted to answer the following research questions:

RQ1: What degree of completeness does the recovered microservice architectural model have?
RQ2: What degree of correctness does the recovered microservice architectural model have?
RQ3: Is the execution time of the MiSAR transformations via QVT efficient or not?

Questions 1 and 2 were formulated to evaluate the effectiveness of the MiSAR transformations and question 3 was formulated to evaluate the efficiency of the MiSAR transformations. To answer RQ1 and RQ2, the total number of architectural elements in the architecture model recovered by MiSAR were compared to the total number of architectural elements in the documented model. The measurement of effectiveness was conducted with the help of recall, precision and F-measure. In precision, the correctness of the recovered architecture model was examined, while recall examined the completeness of the recovered architecture model. To answer RQ3, the efficiency was calculated on the basis of the time required for the recovery of relevant information by QVT/Eclipse.

8.3.2. Case Selection

The software system for the case study was selected by adopting the following criteria:

(i) **Large-size benchmark system:** The case study has 69 microservices (out of which 41 are business-oriented microservices), which is much more than any existing benchmark.

(ii) **The case study was designed using microservice design principles:** For example, the microservices are modularised and organised around business capabilities; a single microservice is small enough to be developed and deployed.

(iii) **Variety of techniques and implementations:** The programming languages and frameworks the case study implements are: Java (Spring Boot, Spring Cloud), Node.js (Express), Python (Django), Go (Webgo) and DB (Mongo, MySQL). Since MiSAR currently was designed to analyse applications developed with Java –Spring Boot, Spring Cloud, this criterion is important to assess how MiSAR behaves when it encounters implementations and technologies not part of MiSAR's design.

(i) **Sufficient testing:** The case study provides sufficient unit test cases and integration test cases, which are publicly available in the open-source project repository. These test cases ensure the quality of the system selected for the case study.

Description: TrainTicket system (Zhou et al., 2018) is a train ticket booking system based on a microservice architecture which contains 41 business-oriented microservices, 49 MB in size on disk. Figure 8-2 shows the TrainTicket system architecture, which also shows the dependencies among microservices. As shown in Figure 8-2, the business-oriented microservices (white hexagons) are distributed into five layers; the bottom layer contains those microservices that do not depend on any other microservice. The upper-layer microservices depend on the lower-layer

microservices. Microservices of the same layer may depend on each other. The system has two means for deployment, Docker and Kubernetes. The artefacts for Docker deployment are those which were selected for the analysis. Having selected the Docker deployment option, the tracing infrastructure microservice for the system was based on Jaeger open tracing. The Gateway, Load Balancer and Service Registry/Discovery infrastructure are handled by a web application built on NGINX called "ts-ui-dashboard". The monitoring infrastructure is implemented with a dedicated application named "ms-monitoring-core". A detailed list of the endpoints and dependency invocations of every microservice is provided in a wiki page at TrainTicket GitHub titled "Service Guide and API Reference"[29].

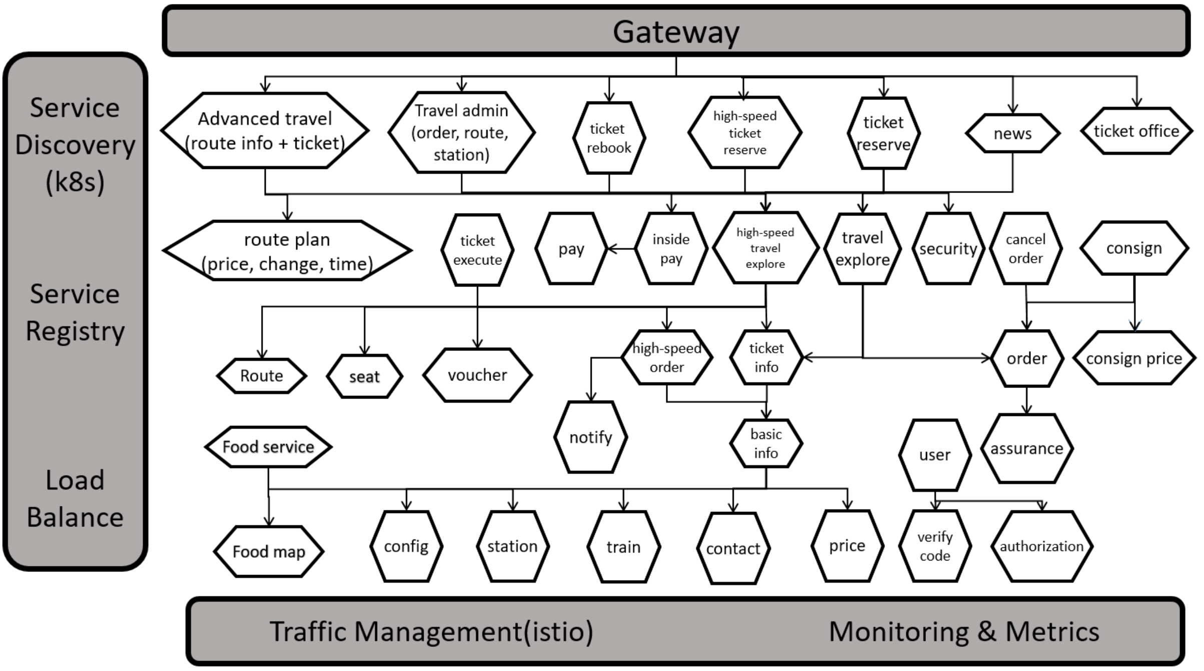


Figure 8-2: TrainTicket architectural diagram.

[29] https://github.com/FudanSELab/train-ticket/wiki/Service-Guide-and-API-Reference.

8.3.3. Application of MiSAR

In the following, I explain how I applied the MiSAR architecture recovery process.

Step 1 – Artefact collection (semi-automatic): The content of the TrainTicket GitHub was first downloaded locally. The required artefacts were then collected, in order to be able to upload them to the existing MiSAR parser, as illustrated in Figure 8-3.

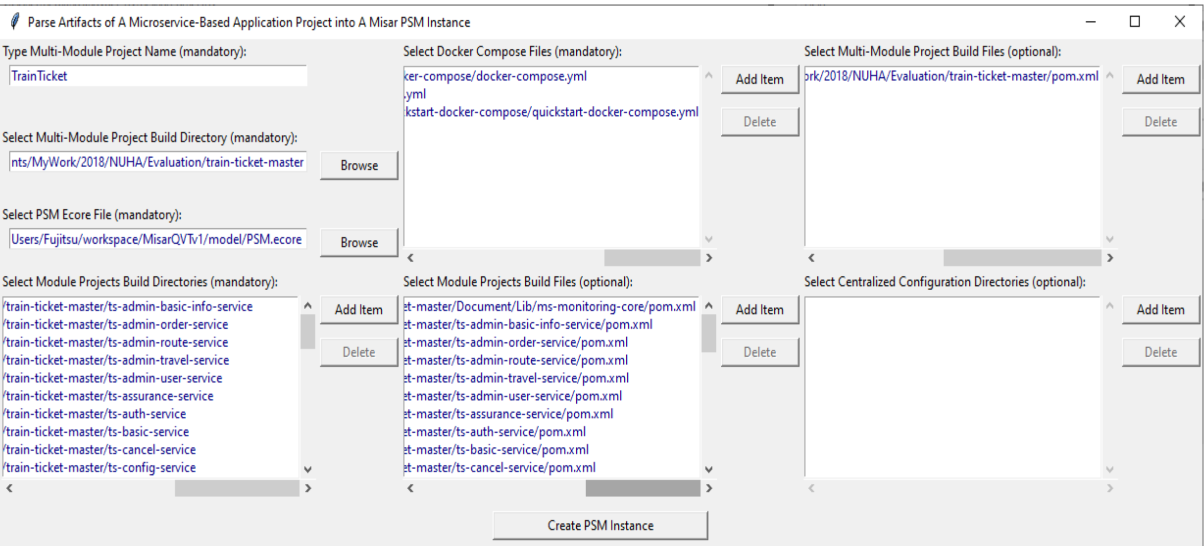


Figure 8-3: Artefact collection.

Mandatory artefact inputs for the recovery process are:

- Project name.
- Path of PSM Ecore metamodel file.
- Build directory of the system (multi-module) project.
- Path of every Docker Compose file (yml).
- Build directory of every microservice (single-module) project.

Additional (optional) artefact inputs include:

- Path of build file (POM) of the system (multi-module) project.
- Path of build file (POM) of every microservice (single-module) project.
- Directory of shared/centralised configurations (if they exist).

Configuration and Java Source artefacts are collected automatically by the parser with the help of the build directory of every microservice project. It is worth mentioning that the completeness of the recovery end results directly depends on the completeness of the artefacts provided in the input. It took around two minutes to manually type in all needed information about the artefacts for TrainTicket.

Step 2 – Instantiate PSM instance (automatic): The MiSAR parser will process the provided artefacts and eventually generate the PSM model in XMI format[30] at the same path as the PSM Ecore file. This XMI file is instantly readable and viewable by the Eclipse QVTo project, as illustrated in Figure 8-4. Parsing takes five minutes to execute.

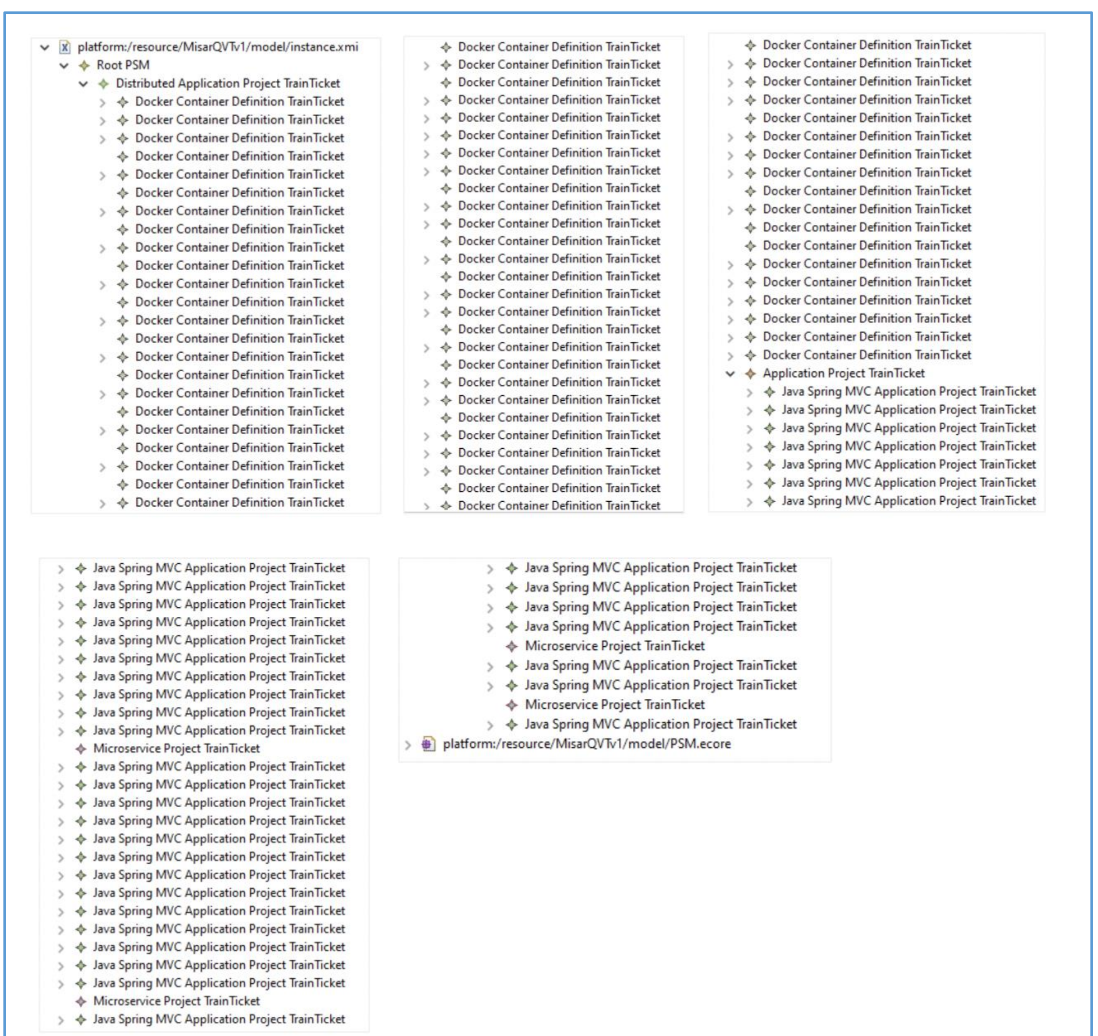


Figure 8-4: Resulting PSM model as viewed in Eclipse QVTo project (from left to right).

[30] https://github.com/nuha77/MiSAR/blob/master/TrainTicketPSM%20(1).xmi.

For simplicity, an elaboration of retrieved PSM instances for just one Java Spring Boot application “ts-auth-service” and four non-JVM applications “ts-news-service”, “ts-ticket-office-service”, “ts-ui-dashboard”, and “ts-voucher-service” are presented here. To illustrate, I will explain a PSM instance from the “ts-auth-service” microservice. Figure 8-5 shows the instance of Docker Container definition that is retrieved for the “ts-auth-service’” microservice, which is a Spring Java project, by parsing the Docker Compose artefacts. Information about this particular PSM instance is extracted by parsing lines 44 to 51 in the “docker-compose.yml” file, as illustrated in Figure 8-6. Source line numbers are attached to every PSM instance for backtrack and reference.

By parsing lines 20 to 23 in the “pom.xml” build file of the Java Spring project “ts-auth-service” (see Figure 8-8), a DependencyLibrary instance is generated, as depicted in Figure 8-7. This particular DependencyLibrary instance attaches the library “spring-boot-starter-data-mongodb” to the “ts-auth-service” project. On compilation, this library will create a MongoDB client component which enables the “ts-auth-service” service to connect to a MongoDB data store server at runtime. This connection (dependency) is enabled and realised by lines 3 to 8 in the “application.yml” configuration file, as presented in Figure 8-10. Parsing these lines results in the ConfigurationProperty that is presented in Figure 8-9.

In addition to direct dependency libraries that appear in the build file of a single project, POM specifications allow for transitive dependencies, i.e. dependency libraries that are defined in the build files of other projects, so that a tree of POM files has to be resolved so that their dependency libraries are parsed. An example of a transitive dependency library is shown in lines 24 to 28 of the “pom.xml” file of “ts-auth-service” (see Figure 8-12). The value “ts-common” of <artifactId> refers to a Java Spring project “ts-common” that is not a microservice project; instead, it contains dependency libraries and Java code files to be shared by other microservice projects, such as “ts-auth-service”. By parsing lines 24 to 28 in “pom.xml” of the “ts-common” project (see Figure 8-12), an additional DependencyLibrary instance is generated and then attached to the “ts-auth-service” PSM model, as shown in Figure 8-11.

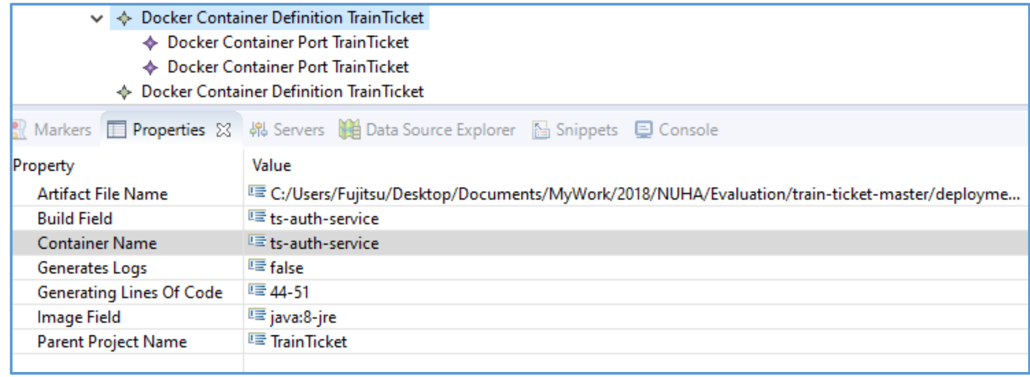


Figure 8-5: Docker Container definition instance retrieved for “ts-auth-service” container.

github.com/FudanSELab/train-ticket/blob/master/docker-compose.yml#L44-L51

```
43
44    ts-auth-service:
45      build: ts-auth-service
46      image: ts/ts-auth-service
47      restart: always
48      ports:
49        - 12340:12340
50      networks:
51        - my-network
52
53    ts-auth-mongo:
54      image: mongo
55      networks:
56        - my-network
```

Figure 8-6: Lines that generated Docker Container definition instance for “ts-auth-service”.

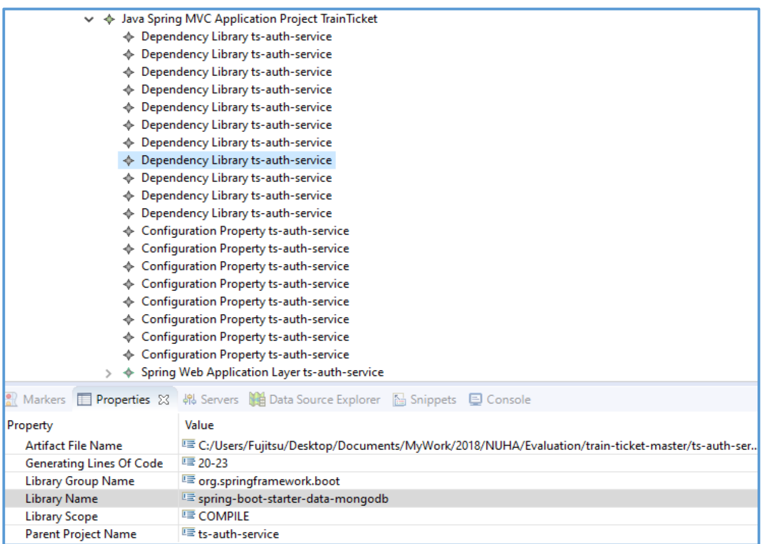


Figure 8-7: DependencyLibrary instance retrieved for “ts-auth-service” Spring Java project.

```
7       <groupId>fdse.microservice</groupId>
8       <artifactId>ts-auth-service</artifactId>
9       <version>1.0</version>
10      <packaging>jar</packaging>
11
12      <parent>
13          <artifactId>ts-service</artifactId>
14          <groupId>org.services</groupId>
15          <version>0.1.0</version>
16      </parent>
17      <modelVersion>4.0.0</modelVersion>
18
19      <dependencies>
20          <dependency>
21              <groupId>org.springframework.boot</groupId>
22              <artifactId>spring-boot-starter-data-mongodb</artifactId>
23          </dependency>
24          <dependency>
25              <groupId>org.services</groupId>
26              <artifactId>ts-common</artifactId>
27              <version>0.1.0</version>
28          </dependency>
```

Figure 8-8: Lines that generated DependencyLibrary instance for "ts-auth-service".

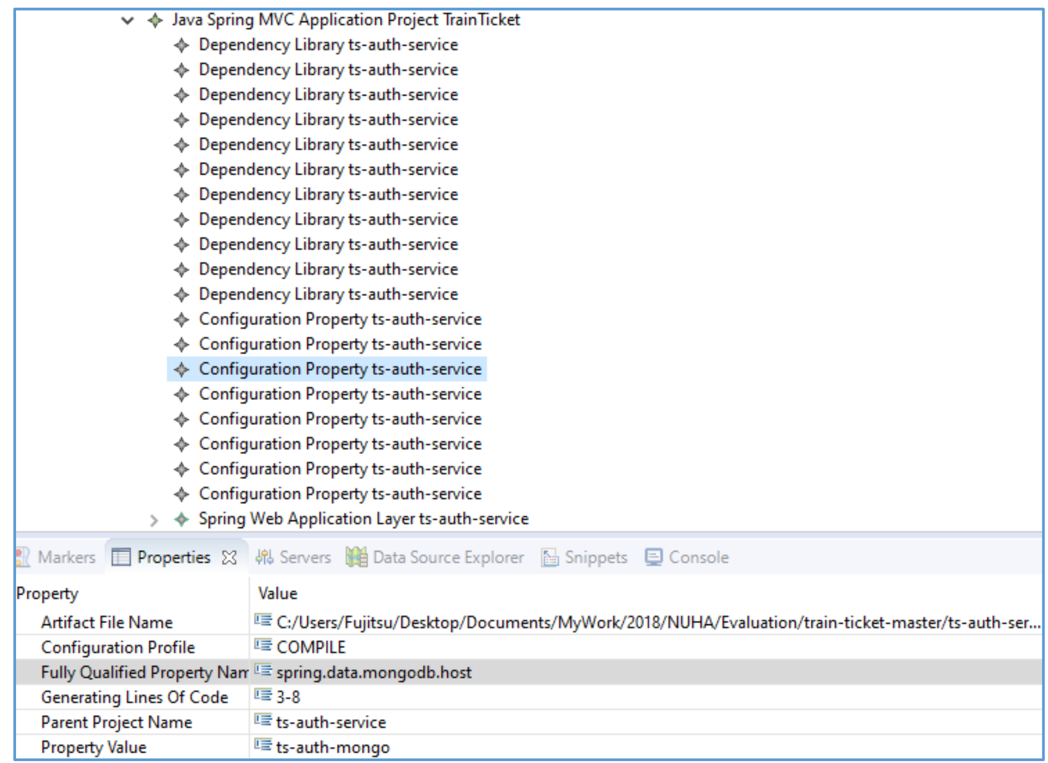


Figure 8-9: ConfigurationProperty instance retrieved for "ts-auth-service" Spring Java project.

```yaml
server:
  port: 12340
spring:
  application:
    name: ts-auth-service
  data:
    mongodb:
      host: ts-auth-mongo
      database: ts-auth-mongo
      port: 27017

swagger:
  controllerPackage: auth.controller

opentracing:
  jaeger:
    udp-sender:
      host: jaeger
      port: 6831
```

Figure 8-10: Lines that generated ConfigurationProperty instance for “ts-auth-service”.

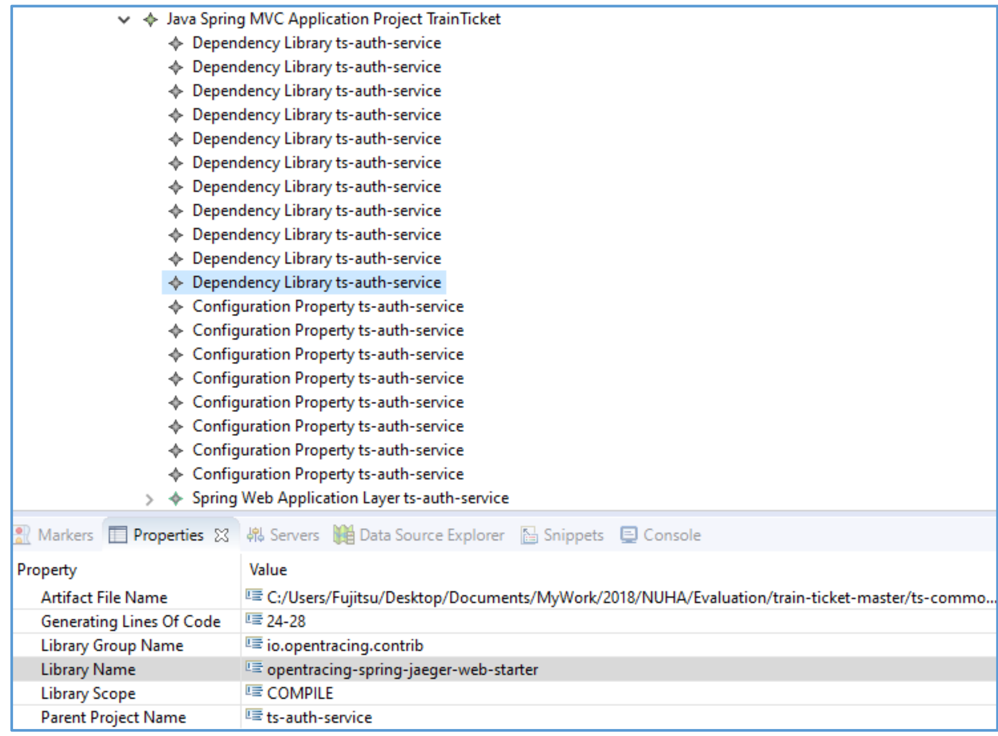


Figure 8-11: DependencyLibrary instance retrieved for “ts-auth-service” Spring Java project.

```
11
12      <artifactId>ts-common</artifactId>
13
14      <dependencies>
15          <dependency>
16              <groupId>io.jsonwebtoken</groupId>
17              <artifactId>jjwt</artifactId>
18              <version>0.8.0</version>
19          </dependency>
20          <dependency>
21              <groupId>org.springframework.boot</groupId>
22              <artifactId>spring-boot-starter-security</artifactId>
23          </dependency>
24          <dependency>
25              <groupId>io.opentracing.contrib</groupId>
26              <artifactId>opentracing-spring-jaeger-web-starter</artifactId>
27              <version>0.2.2</version>
28          </dependency>
29      </dependencies>
```

Figure 8-12: Lines that generated DependencyLibrary instance for “ts-auth-service”.

The retrieved Java Class Type instance depicted in Figure 8-13 reflects the Java source file “AuthController.java”, shown in Figure 8-14. According to the PSM metamodel, every Java Class Type may consist of a set of Java Annotations and Java Methods, along with their children, as illustrated in Figure 8-14. A Java Class Type that has “@RestController” defines the service RESTful endpoints of a microservice application as Java methods decorated with special annotations. For example, the Java class “AuthController” defined in Figure 8-14 declares two methods, “getHello()” and “createDefaultUser()”. By looking at their annotations, it becomes clear that these methods represent, respectively, the “GET” endpoint and the “POST” endpoint of “ts-auth-service”.

For non-JVM microservice applications, such as “ts-news-service”, developed with Go, “ts-ticket-office-service”, developed with “Node.js”, “ts-ui-dashboard”, built with “NGINX” and docker image and “ts-voucher-service”, developed with Python, the Docker Container definition instance (see Figure 8-15) will be parsed from Docker Compose, and the MicroserviceProject instance (see Figure 8-16), the supertype of Java Spring Web Application Project, will be parsed from the name of project’s build directory.

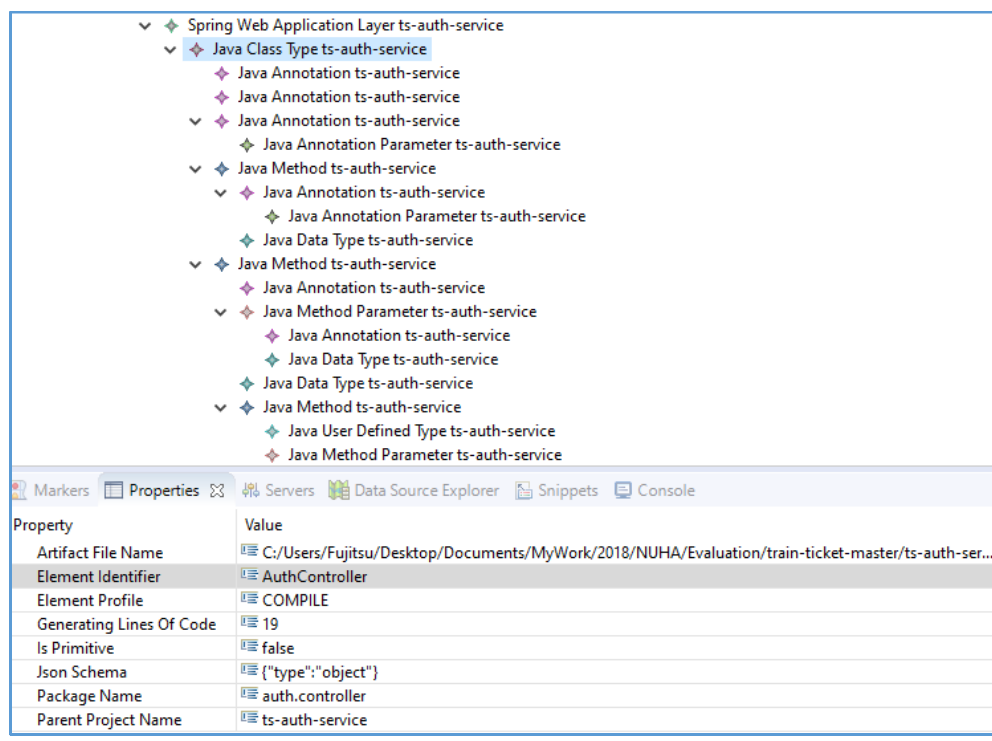


Figure 8-13: JavaClassType instance retrieved for “ts-auth-service” Spring Java project.

```
13  /**
14   * @author fdse
15   */
16  @RestController
17  @Slf4j
18  @RequestMapping("/api/v1/auth")
19  public class AuthController {
20
21      @Autowired
22      private UserService userService;
23
24      /**
25       * only while  user register, this method will be called by ts-user-service
26       * to create a default role use
27       *
28       * @return
29       */
30      @GetMapping("/hello")
31      public String getHello() {
32          return "hello";
33      }
34
35      @PostMapping
36      public HttpEntity<Response> createDefaultUser(@RequestBody AuthDto authDto) {
37          userService.createDefaultAuthUser(authDto);
38          Response response = new Response(1, "SUCCESS", authDto);
39          return new ResponseEntity<>(response, HttpStatus.CREATED);
40      }
41  }
```

Figure 8-14: Java source code that generated the Java Class Type instance for “ts-auth-service”.

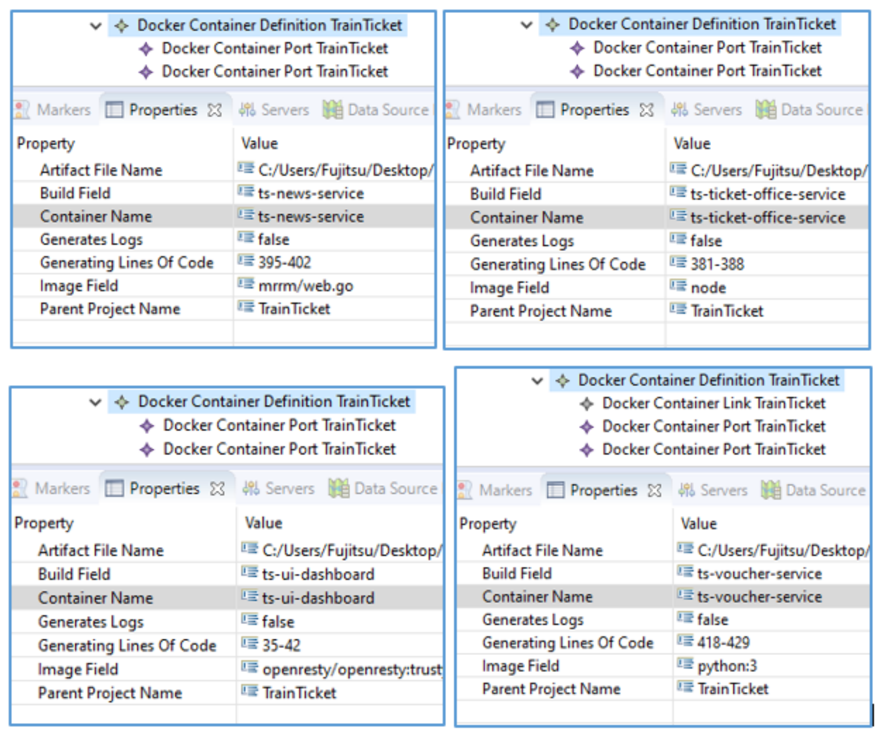


Figure 8-15: DockerContainerDefinition instance retrieved for non-JVM projects.

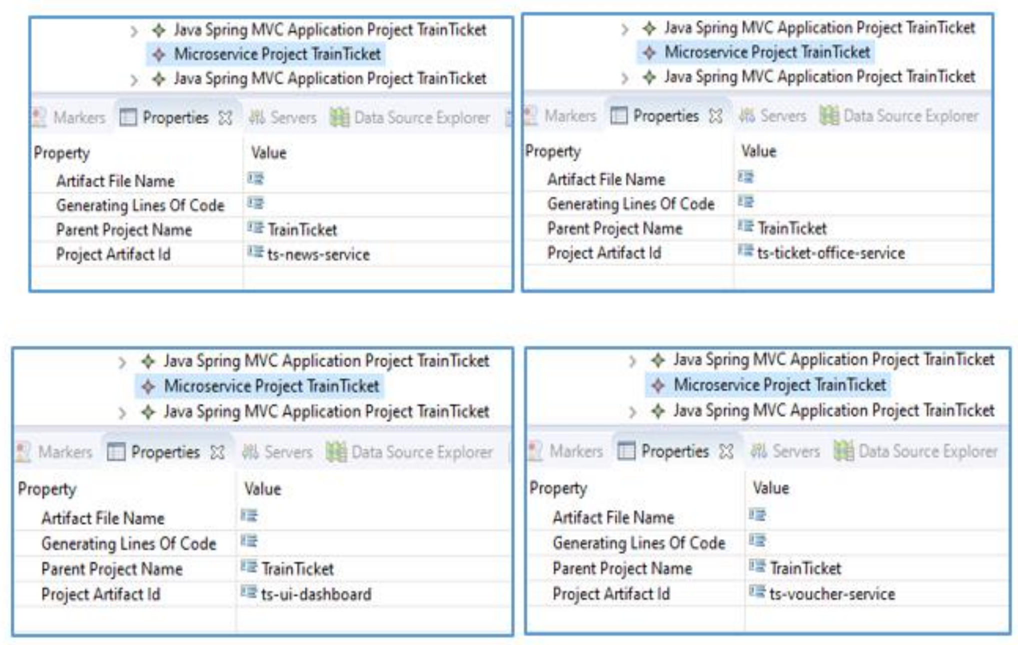


Figure 8-16: MicroserviceProject instance retrieved for non-JVM projects.

Step 3 – Recover PIM instance (automatic): this step is totally automated. The PIM architecture model for TrainTicket is recovered by running the Eclipse QVTo project, which contains the Ecore implementations of both PIM and PSM metamodels, the QVTo implementation of all transformation mapping rules, and the PSM model of TrainTicket generated in step 2. The TrainTicket PIM architecture was recovered within less than two minutes in XMI format. Figure 8-17 illustrates the output PIM model[31]. It can be seen that the TrainTicket architecture consists of 69 microservices: 36 instances of functional microservices (the purple diamonds), 27 instances of infrastructure microservices (the green diamonds) and 6 instances of the supertype microservice (the gold diamonds).

In fact, any recovered microservice is supposed to be an instance of either a functional microservice or an infrastructure microservice. If, instead, an instance of the supertype microservice is retrieved, it indicates that MiSAR has managed to capture the existence of a certain microservice, but for some reason was not able to precisely recognise (classify) its type. It happens that the set of partially recovered microservices includes the "ts-ui-dashboard", "ts-ticket-office-service", "ts-news-service" and "ts-voucher-service" microservices. It is known from the previous step (step 2) that the source artefacts of these microservices belong to non-JVM projects, which MiSAR is not able (yet) to completely recover. However, MiSAR managed to capture their existence in the architecture with the help of Docker Compose artefacts and the name of the build directory of every microservice project acquired by the parser at step 1.

In the following, I will demonstrate the results of the PIM instance for the "ts-auth-service" microservice, a complete and successful recovery, and the "ts-ui-dashboard", "ts-voucher-service", "ts-ticket-office-service", "ts-news-service", "jaeger" and "ms-monitoring-core" microservices, which were incompletely recovered. For every microservice previously mentioned, I will present the PIM instance output and a table with attribute values of recovered PIM concepts.

[31] https://github.com/nuha77/MiSAR/blob/master/TrainTicket.PIM.

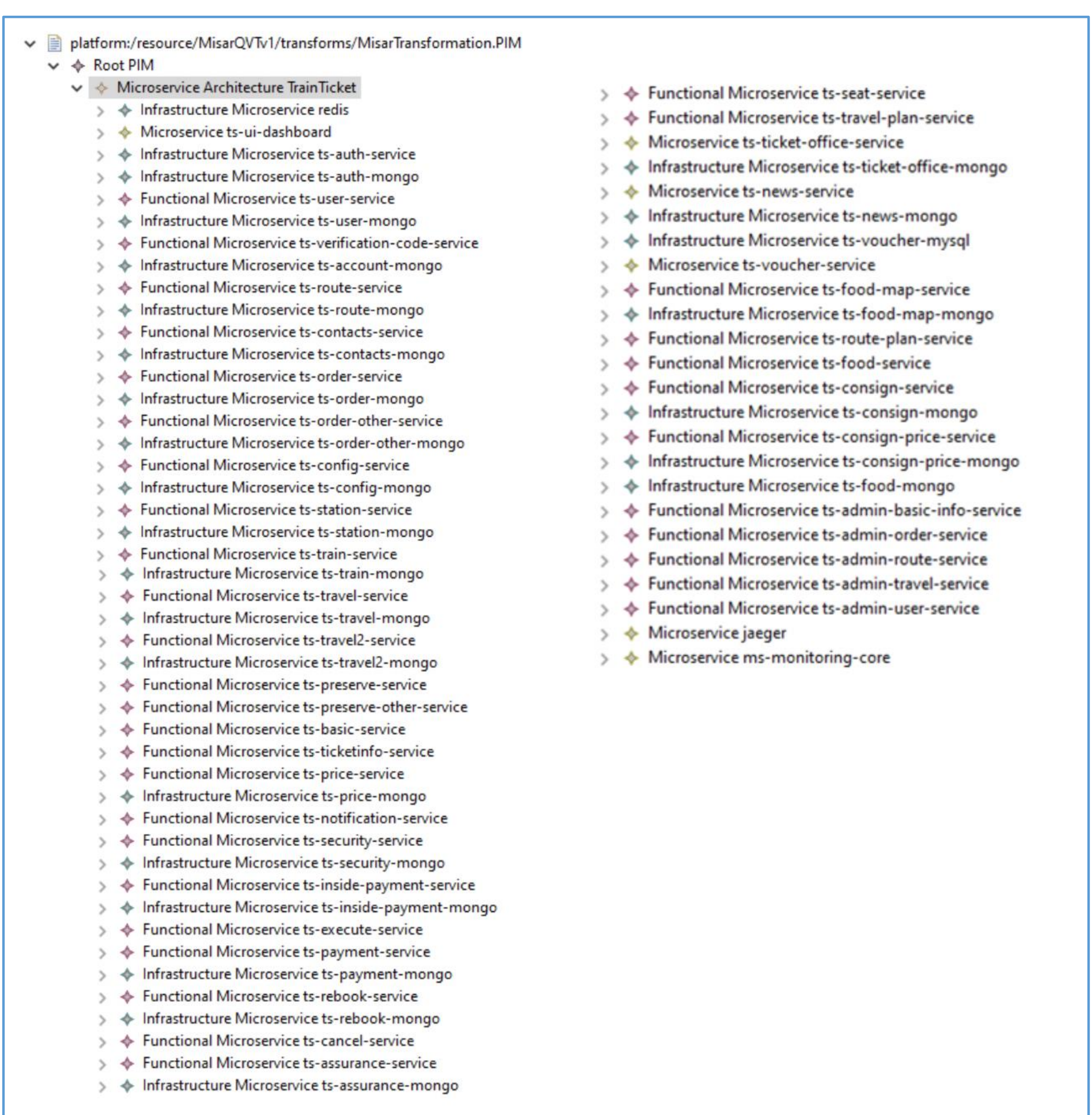


Figure 8-17: PIM model for TrainTicket recovered by MiSAR.

1) **Complete and successful recovery example:**

- **"ts-auth-service" Microservice:** "ts-auth-service" is an infrastructure microservice that provides APIs to manage user information and auth operations. Figure 8-18 represents the PIM instance for "ts-auth-service" and Table 8-1 depicts all the attribute values of the recovered PIM concepts for "ts-auth-service".

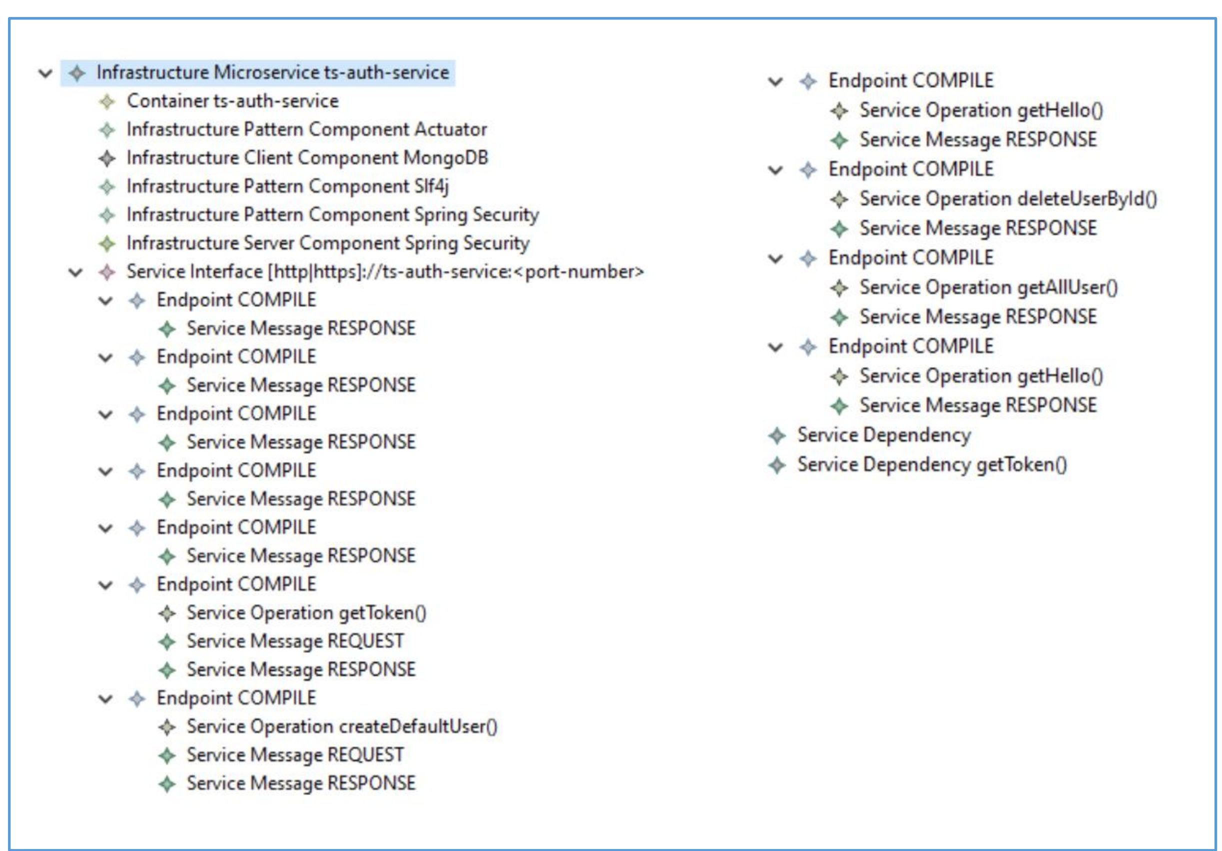


Figure 8-18: Infrastructure microservice instance recovered for "ts-auth-service".

Table 8-1: All recovered PIM elements for "ts-auth-service".

	ConceptName	ConceptAttribute	AttributeValue
1	Container	ContainerName	ts-auth-service
2	InfrastructureMicroservice	MicroserviceName	ts-auth-service
3	InfrastructurePatternComponent	Category:Technology	Observability_Pattern_Application_Metrics_Generation:Actuator
4	InfrastructureClientComponent	Category:Technology	Development_Pattern_Data_Persistence:MongoDB
5	InfrastructurePatternComponent	Category:Technology	Observability_Pattern_Application_Metrics_Logging:Slf4j
6	InfrastructurePatternComponent	Category:Technology	Security_Pattern_Web_Security:Spring Security
7	InfrastructureServerComponent	Category:Technology	Security_Pattern_Authorization_and_Authentication:Spring Security
8	Endpoint	RequestURI	GET /actuator/health
9	ServiceMessage	MessageType	RESPONSE
10	ServiceMessage	BodySchema	{"type":"object","properties":{"status":{"type":"string"},"details" …
11	Endpoint	RequestURI	GET /actuator/info
12	ServiceMessage	MessageType	RESPONSE
13	ServiceMessage	BodySchema	{"type":"object","properties":{"git":{"type":"object","properties" …
14	Endpoint	RequestURI	GET /actuator/metrics
15	ServiceMessage	MessageType	RESPONSE
16	ServiceMessage	BodySchema	{"type":"object","properties":{"Datacenter":{"type":"string"},"ID "…

17	Endpoint	RequestURI	POST /actuator/shutdown
18	ServiceMessage	MessageType	RESPONSE
19	ServiceMessage	BodySchema	{"type":"object","properties":{"message":{"type":"string"}}}
20	Endpoint	RequestURI	POST /actuator/restart
21	ServiceMessage	MessageType	RESPONSE
22	ServiceMessage	BodySchema	{"type":"object","properties":{"message":{"type":"string"}}}
23	Endpoint	RequestURI	POST /api/v1/users/login
24	ServiceOperation	OperationName	getToken()
25	ServiceOperation	OperationDescription	Receives a request of type: (BasicAuthDto) and returns a response…
26	ServiceMessage	MessageType	REQUEST
27	ServiceMessage	BodySchema	{"type":"object","properties":{"serialVersionUID":{"type":"intege…
28	ServiceMessage	MessageType	RESPONSE
29	ServiceMessage	BodySchema	{"type":"object","properties":{"status":{"type":"integer"},"msg":{…
30	Endpoint	RequestURI	POST /api/v1/auth
31	ServiceOperation	OperationName	createDefaultUser()
32	ServiceOperation	OperationDescription	Receives a request of type: (AuthDto) and returns a response mes…
33	ServiceMessage	MessageType	REQUEST
34	ServiceMessage	BodySchema	{"type":"object","properties":{"userId":{"type":"string"},"userna…
35	ServiceMessage	MessageType	RESPONSE
36	ServiceMessage	BodySchema	{"type":"object","properties":{"status":{"type":"integer"},"msg":{…
37	Endpoint	RequestURI	GET /api/v1/users/hello
38	ServiceOperation	OperationName	getHello()
39	ServiceOperation	OperationDescription	Returns a response message of type: (Object)
40	ServiceMessage	MessageType	RESPONSE
41	ServiceMessage	BodySchema	{"type":"object"}
42	Endpoint	RequestURI	DELETE /api/v1/users/{userId}
43	ServiceOperation	OperationName	deleteUserById()
44	ServiceOperation	OperationDescription	Returns a response message of type: (Response)
45	ServiceMessage	MessageType	RESPONSE
46	ServiceMessage	BodySchema	{"type":"object","properties":{"status":{"type":"integer"},"msg":{…
47	Endpoint	RequestURI	GET /api/v1/users
48	ServiceOperation	OperationName	getAllUser()
49	ServiceOperation	OperationDescription	Returns a response message of type: (List<User>)
50	ServiceMessage	MessageType	RESPONSE
51	ServiceMessage	BodySchema	{"type":"object","properties":{"status":{"type":"integer"},"msg":{…
52	Endpoint	RequestURI	GET /api/v1/auth/hello
53	ServiceOperation	OperationName	getHello()
54	ServiceOperation	OperationDescription	Returns a response message of type: (String)
55	ServiceMessage	MessageType	RESPONSE
56	ServiceMessage	BodySchema	{"type":"string"}
57	ServiceDependency	ProviderName	ts-auth-mongo
58	ServiceDependency	ProviderName	ts-verification-code-service

59	ServiceDependency	ProviderDestination	GET /api/v1/verifycode/verify/{verifyCode}
60	ServiceDependency	ConsumerOperation	getToken()

2) Incomplete and failed recovery example:

- The microservice "ts-ui-dashboard" is a service that provides all the UI interface to interact with the system. Figure 8-19 shows the PIM instance for "ts-ui-dashboard", as can be seen, the container and service interface of "ts-ui-dashboard" were both recovered. Table 8-2 depicts all attribute values of the recovered PIM concepts for "ts-ui-dashboard".

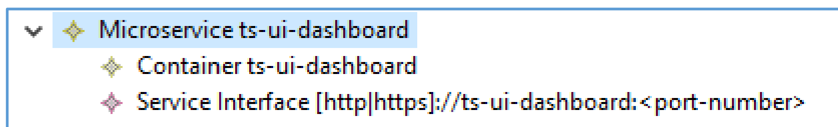


Figure 8-19: Microservice instance recovered for "ts-ui-dashboard".

Table 8-2: All recovered PIM elements for "ts-ui-dashboard".

ID	ConceptName	ConceptAttribute	AttributeValue
1	Container	ContainerName	ts-ui-dashboard
2	Microservice	MicroserviceName	ts-ui-dashboard

- The microservice "ts-voucher-service" provides APIs to generate the reimbursement voucher based on the order ID. Figure 8-20 shows the PIM instance for "ts-voucher-service", it can be seen that the container, service interface and service dependency were recovered. Table 8-3 depicts all attribute values of the recovered PIM concepts for "ts-voucher-service".

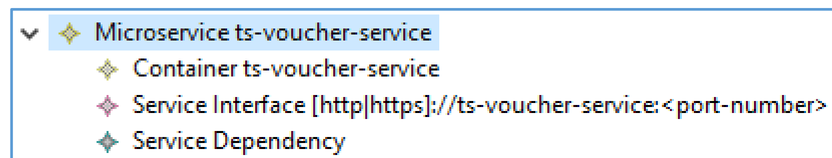


Figure 8-20: Microservice instance recovered for “ts-voucher-service”.

Table 8-3: All recovered PIM elements for “ts-voucher-service”.

ID	ConceptName	ConceptAttribute	AttributeValue
1	Container	ContainerName	ts-voucher-service
2	Microservice	MicroserviceName	ts-voucher-service
3	ServiceDependency	ProviderName	ts-voucher-mysql

- Microservice “ms-monitoring-core”: Figure 8-21 shows the PIM instance for “ms-monitoring-core”. Table 8-4 depict all attribute values of the recovered PIM concepts for “ms-monitoring-core”.

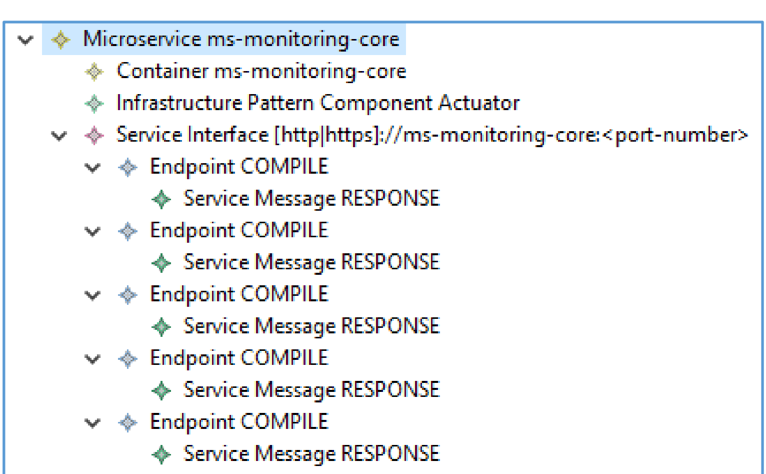


Figure 8-21: Microservice instance recovered for “ms-monitoring-core”.

Table 8-4: All recovered PIM elements for “ms-monitoring-core”.

ID	ConceptName	ConceptAttribute	AttributeValue
1	Container	ContainerName	ms-monitoring-core
2	Microservice	MicroserviceName	ms-monitoring-core
3	InfrastructurePatternComponent	Category:Technology	Observability_Pattern_Application_Metrics_Generation:Actuator
4	Endpoint	RequestURI	GET /actuator/health
5	ServiceMessage	MessageType	RESPONSE
6	ServiceMessage	BodySchema	{"type":"object","properties":{"status":{"type":"string"},"details"…
7	Endpoint	RequestURI	GET /actuator/info
8	ServiceMessage	MessageType	RESPONSE
9	ServiceMessage	BodySchema	{"type":"object","properties":{"git":{"type":"object","properties"…
10	Endpoint	RequestURI	GET /actuator/metrics
11	ServiceMessage	MessageType	RESPONSE
12	ServiceMessage	BodySchema	{"type":"object","properties":{"Datacenter":{"type":"string"},"ID" …
13	Endpoint	RequestURI	POST /actuator/shutdown
14	ServiceMessage	MessageType	RESPONSE
15	ServiceMessage	BodySchema	{"type":"object","properties":{"message":{"type":"string"}}}
16	Endpoint	RequestURI	POST /actuator/restart
17	ServiceMessage	MessageType	RESPONSE
18	ServiceMessage	BodySchema	{"type":"object","properties":{"message":{"type":"string"}}}

- Microservice “ts-ticket-office-service”: Figure 8-22 shows the PIM instance for “ts-ticket-office-service”. Table 8-5 depict all attribute values of the recovered PIM concepts for “ts-ticket-office-service”.

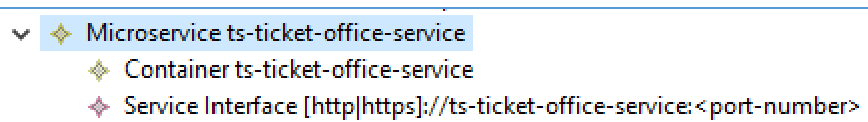


Figure 8-22: Microservice instance recovered for “ts-ticket-office-service”.

Table 8-5: All recovered PIM elements for “ts-ticket-office-service”.

ID	ConceptName	ConceptAttribute	AttributeValue
1	Container	ContainerName	ts-ticket-office-service
2	Microservice	MicroserviceName	ts-ticket-office-service

- Microservice “ts-news-service”: Figure 8-23 shows the PIM instance for “ts-news-service”. Table 8-6 depict all attribute values of the recovered PIM concepts for “ts-news-service”.

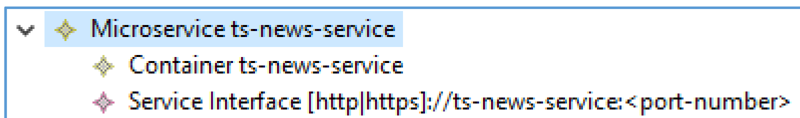


Figure 8-23: Microservice instance recovered for “ts-news-service”.

Table 8-6: All recovered PIM elements for “ts-news-service”.

ID	ConceptName	ConceptAttribute	AttributeValue
1	Container	ContainerName	ts-news-service
2	Microservice	MicroserviceName	ts-news-service

- Microservice “jaeger”: Figure 8-24 shows the PIM instance for “jaeger”. Table 8-7 depict all attribute values of the recovered PIM concepts for “jaeger”.

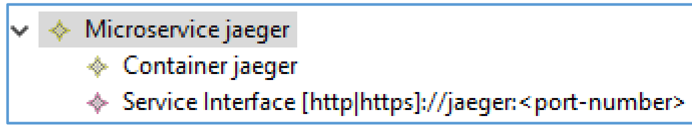


Figure 8-24: Microservice instance recovered for “jaeger”.

Table 8-7: All recovered PIM elements for “jaeger”.

ID	ConceptName	ConceptAttribute	AttributeValue
1	Container	ContainerName	jaeger
2	Microservice	MicroserviceName	jaeger

8.3.4. Consistency Checks

This evaluation is based on manually checking the consistency between the generated architectural model and the documentation. A consistency check is executed by constructing a table with the expected target PIM elements according to the available documentation against the PIM elements recovered by MiSAR, the results of the previous section.

The selected case study, TrainTicket, has comprehensive documentation in two forms, an architecture diagram and a wiki page[32]; the latter describes infrastructure technologies used in the development of TrainTicket, as well as providing a complete list of endpoints and invocations for every business microservice. The available documentation corresponds to a subset of the recovered architecture elements in MiSAR's PIM metamodel. While the documented architecture diagram represents functional/infrastructure microservices as well as infrastructure server/client/pattern components, it doesn't demonstrate, for instance, any data store infrastructure microservices although they are defined in the Docker Compose artefact.

Moreover, while the list of endpoints and invocations defined in the wiki page is consistent with the endpoint and service dependency elements recovered, it totally lacks representation of the service operation and service message elements. This encouraged me to add some architecture elements missed in the documentation to the set of test cases, such as data store containers being infrastructure microservices and the service messages to every endpoint, to ensure they contribute to the metrics, hence a more accurate and comprehensive evaluation is performed of the MiSAR repository (i.e. mapping rules and metamodels).

The comparison was conducted based on attributes for every element where the expected architecture elements and attributes are deduced from the documentation and then compared to the recovered elements. After that, the recovery result for every element attribute (test case) was recorded in a table. For simplicity, I will present in the following the test tables for the microservices considered previously in section 8.3.3 (step 3).

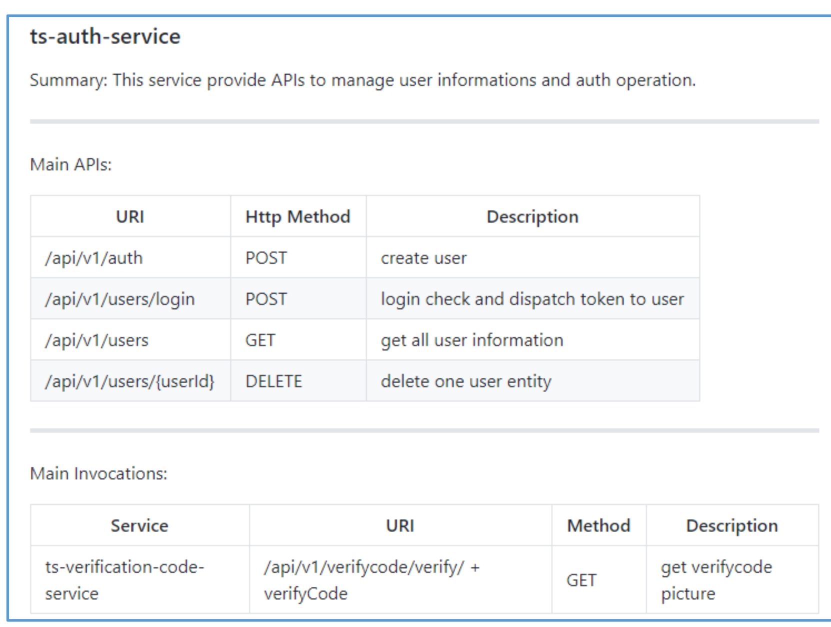
[32] https://github.com/FudanSELab/train-ticket/wiki/Service-Guide-and-API-Reference.

Microservice "ts-auth-service": By checking the documentation of "ts-auth-service" provided in Figure 8-25, one can directly deduce that this microservice is providing authorisation and authentication infrastructure. Moreover, this microservice exposes four REST-full endpoints and interacts with the "ts-verification-code-service" microservice by sending requests to one endpoint at the provider. In addition, by checking the documented architecture diagram in Figure 8-2, it can be seen that this microservice makes use of three types of infrastructure components: service registry/discovery, logging and tracing server. To facilitate the comparison, the documented architecture is described in terms of PIM elements, as illustrated in Table 8-8.

- **Consistent with documentation:** By comparing Table 8-1 showing the recovered PIM instance for "ts-auth-service" to Table 8-8 showing the expected PIM elements, it is clear that MiSAR correctly recovered 23 architecture elements with their attributes out of 26, with the help of its current repository of the PSM metamodel, PSM instance and QVTo mapping rules.

- **Additional elements:** With reference to Table 8-1, MiSAR also recovered more architecture elements compared to the documentation. In particular, MiSAR was able to recover the service operation of the documented endpoints, in addition to several infrastructure pattern component, infrastructure client component and service dependency elements that are not documented. To illustrate, the infrastructure client component element of the category Development_Pattern_Data_Persistence (ID=4 in Table 8-1) and the service dependency element with provider name "ts-auth-mongo" (ID=57 in Table 8-1) both state that the "ts-auth-service" microservice uses and interacts with a data store named "ts-auth-mongo". To ensure the validity of this statement, I backtracked the Generating PSM attribute of the two retrieved elements, as indicated in figures 8-26 and 8-27. The first refers to a dependency library PSM element extracted from the build file, as shown in figures 8-7 and 8-8, while the second refers to a configuration property PSM element extracted from the config file, as shown in figures 8-9 and 8-10.

- **Missed elements:** The first missed component is a client to service registry infrastructure (ID=4 in Table 8-8), which, according to the documentation, implements Kubernetes (k8s). The second and third are related to tracing infrastructure (ID=5 and ID=26 in Table 8-8), which, according to the documentation, implements Jaeger. MiSAR does not yet support the two technologies in its repository of mapping rules.


ts-auth-service

Summary: This service provide APIs to manage user informations and auth operation.

Main APIs:

URI	Http Method	Description
/api/v1/auth	POST	create user
/api/v1/users/login	POST	login check and dispatch token to user
/api/v1/users	GET	get all user information
/api/v1/users/{userId}	DELETE	delete one user entity

Main Invocations:

Service	URI	Method	Description
ts-verification-code-service	/api/v1/verifycode/verify/ + verifyCode	GET	get verifycode picture

Figure 8-25: TrainTicket's documentation for "ts-auth-service".

Table 8-8: Expected elements for “ts-auth-service” as per the documentation vs MiSAR result.

ID	Element Name	Element Attribute	Attribute Value	MiSAR Output
1	Container	ContainerName	ts-auth-service	RECOVERED
2	InfrastructureMicroservice	MicroserviceName	ts-auth-service	RECOVERED
3	InfrastructureServerComponent	Category	Security_Pattern_Authorization_and_Authentication	RECOVERED
4	InfrastructureClientComponent	Category	Service_Routing_Pattern_Registry_and_Discovery	NOT_RECOVERED
5	InfrastructureClientComponent	Category	Observability_Pattern_Distributed_Tracing	NOT_RECOVERED
6	InfrastructurePatternComponent	Category	Observability_Pattern_Application_Metrics_Logging	RECOVERED
7	InfrastructurePatternComponent	Category	Observability_Pattern_Application_Metrics_Generation	RECOVERED
8	Endpoint	RequestURI	POST /api/v1/auth	RECOVERED
9	ServiceMessage	MessageType	REQUEST	RECOVERED
10	ServiceMessage	BodySchema	{"type":"object","properties":{"userId":{"type":"string"},"…	RECOVERED
11	ServiceMessage	MessageType	RESPONSE	RECOVERED
12	ServiceMessage	BodySchema	{"type":"object","properties":{"status":{"type":"integer"},"…	RECOVERED
13	Endpoint	RequestURI	POST /api/v1/users/login	RECOVERED
14	ServiceMessage	MessageType	REQUEST	RECOVERED
15	ServiceMessage	BodySchema	{"type":"object","properties":{"verificationCode":{"type…	RECOVERED
16	ServiceMessage	MessageType	RESPONSE	RECOVERED
17	ServiceMessage	BodySchema	{"type":"object","properties":{"status":{"type":"integer"},"…	RECOVERED
18	Endpoint	RequestURI	GET /api/v1/users	RECOVERED
19	ServiceMessage	MessageType	RESPONSE	RECOVERED
20	ServiceMessage	BodySchema	{"type":"object","properties":{"status":{"type":"integer"},"…	RECOVERED
21	Endpoint	RequestURI	DELETE /api/v1/users/{userId}	RECOVERED
22	ServiceMessage	MessageType	RESPONSE	RECOVERED
23	ServiceMessage	BodySchema	{"type":"object","properties":{"status":{"type":"integer"},"…	RECOVERED
24	ServiceDependency	ProviderName	ts-verification-code-service	RECOVERED
25	ServiceDependency	ProviderDestination	GET /api/v1/verifycode/verify/{verifyCode}	RECOVERED
26	ServiceDependency	ProviderName	jaeger	NOT_RECOVERED

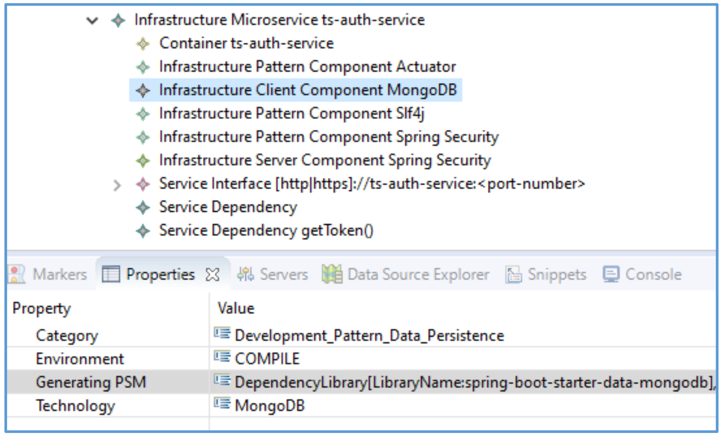


Figure 8-26: Generating PSM attribute for recovered infrastructure pattern component element.

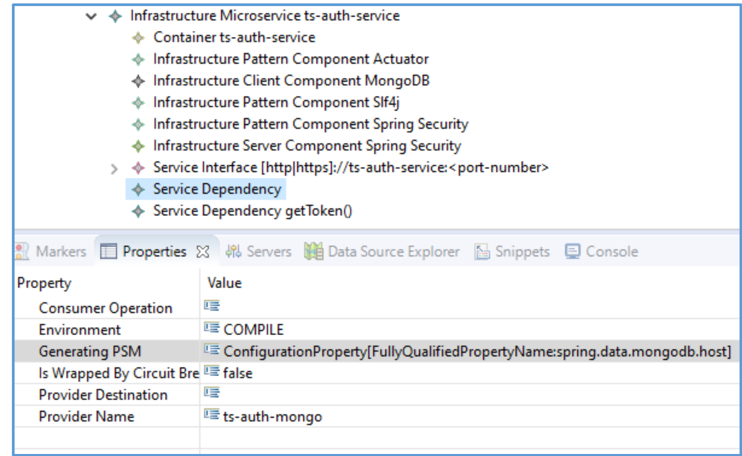


Figure 8-27: Generating PSM attribute for recovered service dependency element.

Microservice "ts-ui-dashboard": By reading the brief documentation of "ts-ui-dashboard" provided in Figure 8-28, one might assume that this microservice is providing gateway and routing infrastructure. To confirm this assumption, I checked artefacts in the build directory of the "ts-ui-dashboard" project. I found that this microservice is built from a Docker hub image of the NGINX proxy and web server. Therefore, "ts-ui-dashboard" offers proxy, service registry/discovery and load balancer infrastructure. Moreover, I checked the "nginx.conf" file and found that "ts-ui-dashboard", as it interfaces all backend microservices, has 83 endpoints in addition to 40 service dependencies via a total of 83 provider endpoints. Similar to "ts-auth-

service", by checking the documented diagram in Figure 8-2, "ts-ui-dashboard" makes use of three types of infrastructure, monitoring, logging and tracing server. To facilitate the comparison, the documented architecture is described in terms of PIM elements, as illustrated in Table 8-9.

- **Consistent with documentation:** By comparing Table 8-2 for the recovered PIM instance for "ts-ui-dashboard" to Table 8-9 for the expected PIM elements, it is clear that MiSAR, out of 212 expected elements, correctly recovered the container element and inaccurately recovered the infrastructural microservice element as a microservice. It only captured the existence of this microservice with the help of the Docker Compose artefact and the name of build directory acquired from the parser.

- **Missed elements:** Apparently, MiSAR failed to recover infrastructure components, endpoints and service dependencies for "ts-ui-dashboard". In fact, MiSAR does not yet transform NGINX configuration artefacts such as "nginx.conf" to recover endpoints and service dependencies. Although NGINX as a proxy, service registry/discovery and load balancer server is familiar to MiSAR, the Docker hub image with the name "openresty" is not.

> **ts-ui-dashboard**
>
> Summary: This repository provide all the UI interface to interact with the system.

Figure 8-28: TrainTicket's documentation for "ts-ui-dashboard".

Table 8-9: Expected elements for "ts-ui-dashboard" as per the documentation vs MiSAR result

ID	Element Name	Element Attribute	Attribute Value	MiSAR Output
1	Container	ContainerName	ts-ui-dashboard	RECOVERED
2	InfrastructureMicroservice	MicroserviceName	ts-ui-dashboard	MICROSERVICE
3	InfrastructureServerComponent	Category	Service_Routing_Pattern_API_Gateway_and_Proxy	NOT_RECOVERED
4	InfrastructureServerComponent	Category	Client_Resiliency_Pattern_Load_Balancer	NOT_RECOVERED
5	InfrastructureServerComponent	Category	Service_Routing_Pattern_Registry_and_Discovery	NOT_RECOVERED

6	InfrastructureClientComponent	Category	Observability_Pattern_Distributed_Tracing	NOT_RECOVERED
7	InfrastructurePatternComponent	Category	Observability_Pattern_Application_Metrics_Logging	NOT_RECOVERED
8	InfrastructurePatternComponent	Category	Observability_Pattern_Application_Metrics_Generation	NOT_RECOVERED
9	Endpoint	RequestURI	/api/v1/travelservice/trips/left	NOT_RECOVERED
10	ServiceDependency	ProviderName	ts-travel-service	NOT_RECOVERED
11	ServiceDependency	ProviderDestination	/api/v1/travelservice/trips/left	NOT_RECOVERED
12	Endpoint	RequestURI	/api/v1/travel2service/trips/left	NOT_RECOVERED
13	ServiceDependency	ProviderName	ts-travel2-service	NOT_RECOVERED
14	ServiceDependency	ProviderDestination	/api/v1/travel2service/trips/left	NOT_RECOVERED
15	Endpoint	RequestURI	/api/v1/userservice/users	NOT_RECOVERED
16	ServiceDependency	ProviderName	ts-user-service	NOT_RECOVERED
17	ServiceDependency	ProviderDestination	/api/v1/userservice/users	NOT_RECOVERED
18	Endpoint	RequestURI	/api/v1/users/auth	NOT_RECOVERED
19	Endpoint	RequestURI	/api/v1/users/login	NOT_RECOVERED
20	ServiceDependency	ProviderName	ts-user-service	NOT_RECOVERED
21	ServiceDependency	ProviderDestination	/api/v1/users/auth	NOT_RECOVERED
22	ServiceDependency	ProviderDestination	/api/v1/users/login	NOT_RECOVERED
23	Endpoint	RequestURI	/api/v1/verifycode/generate	NOT_RECOVERED
24	ServiceDependency	ProviderName	ts-verification-code-service	NOT_RECOVERED
25	ServiceDependency	ProviderDestination	/api/v1/verifycode/generate	NOT_RECOVERED
26	Endpoint	RequestURI	/api/v1/stationservice	NOT_RECOVERED
27	ServiceDependency	ProviderName	ts-station-service	NOT_RECOVERED
28	ServiceDependency	ProviderDestination	/api/v1/stationservice	NOT_RECOVERED
29	Endpoint	RequestURI	/api/v1/trainservice	NOT_RECOVERED
30	ServiceDependency	ProviderName	ts-train-service	NOT_RECOVERED
31	ServiceDependency	ProviderDestination	/api/v1/trainservice	NOT_RECOVERED
32	Endpoint	RequestURI	/api/v1/configservice	NOT_RECOVERED
33	ServiceDependency	ProviderName	ts-config-service	NOT_RECOVERED
34	ServiceDependency	ProviderDestination	/api/v1/configservice	NOT_RECOVERED
35	Endpoint	RequestURI	/api/v1/securityservice	NOT_RECOVERED
36	ServiceDependency	ProviderName	ts-security-service	NOT_RECOVERED
37	ServiceDependency	ProviderDestination	/api/v1/securityservice	NOT_RECOVERED
38	Endpoint	RequestURI	/api/v1/executeservice/execute/execute	NOT_RECOVERED
39	Endpoint	RequestURI	/api/v1/executeservice/execute/collected	NOT_RECOVERED
40	ServiceDependency	ProviderName	ts-execute-service	NOT_RECOVERED
41	ServiceDependency	ProviderDestination	/api/v1/executeservice/execute/execute	NOT_RECOVERED
42	ServiceDependency	ProviderDestination	/api/v1/executeservice/execute/collected	NOT_RECOVERED
43	Endpoint	RequestURI	/api/v1/contactservice/contacts	NOT_RECOVERED
44	Endpoint	RequestURI	/api/v1/contactservice/contacts/account	NOT_RECOVERED
45	ServiceDependency	ProviderName	ts-contacts-service	NOT_RECOVERED
46	ServiceDependency	ProviderDestination	/api/v1/contactservice/contacts	NOT_RECOVERED
47	ServiceDependency	ProviderDestination	/api/v1/contactservice/contacts/account	NOT_RECOVERED
48	Endpoint	RequestURI	/api/v1/orderservice/order/refresh	NOT_RECOVERED
49	ServiceDependency	ProviderName	ts-order-service	NOT_RECOVERED

50	ServiceDependency	ProviderDestination	/api/v1/orderservice/order/refresh	NOT_RECOVERED
51	Endpoint	RequestURI	/api/v1/orderOtherService/orderOther/refresh	NOT_RECOVERED
52	ServiceDependency	ProviderName	ts-order-other-service	NOT_RECOVERED
53	ServiceDependency	ProviderDestination	/api/v1/orderOtherService/orderOther/refresh	NOT_RECOVERED
54	Endpoint	RequestURI	/api/v1/preserveservice/preserve	NOT_RECOVERED
55	ServiceDependency	ProviderName	ts-preserve-service	NOT_RECOVERED
56	ServiceDependency	ProviderDestination	/api/v1/preserveservice/preserve	NOT_RECOVERED
57	Endpoint	RequestURI	/api/v1/preserveotherservice/preserveOther	NOT_RECOVERED
58	ServiceDependency	ProviderName	ts-preserve-other-service	NOT_RECOVERED
59	ServiceDependency	ProviderDestination	/api/v1/preserveotherservice/preserveOther	NOT_RECOVERED
60	Endpoint	RequestURI	/price/query	NOT_RECOVERED
61	Endpoint	RequestURI	/price/queryAll	NOT_RECOVERED
62	Endpoint	RequestURI	/price/create	NOT_RECOVERED
63	Endpoint	RequestURI	/price/delete	NOT_RECOVERED
64	Endpoint	RequestURI	/price/update	NOT_RECOVERED
65	ServiceDependency	ProviderName	ts-price-service	NOT_RECOVERED
66	ServiceDependency	ProviderDestination	/price/query	NOT_RECOVERED
67	ServiceDependency	ProviderDestination	/price/queryAll	NOT_RECOVERED
68	ServiceDependency	ProviderDestination	/price/create	NOT_RECOVERED
69	ServiceDependency	ProviderDestination	/price/delete	NOT_RECOVERED
70	ServiceDependency	ProviderDestination	/price/update	NOT_RECOVERED
71	Endpoint	RequestURI	/basic/queryForTravel	NOT_RECOVERED
72	ServiceDependency	ProviderName	ts-basic-service	NOT_RECOVERED
73	ServiceDependency	ProviderDestination	/basic/queryForTravel	NOT_RECOVERED
74	Endpoint	RequestURI	/ticketinfo/queryForTravel	NOT_RECOVERED
75	ServiceDependency	ProviderName	ts-ticketinfo-service	NOT_RECOVERED
76	ServiceDependency	ProviderDestination	/ticketinfo/queryForTravel	NOT_RECOVERED
77	Endpoint	RequestURI	/notification/preserve_success	NOT_RECOVERED
78	Endpoint	RequestURI	/notification/order_create_success	NOT_RECOVERED
79	Endpoint	RequestURI	/notification/order_changed_success	NOT_RECOVERED
80	ServiceDependency	ProviderName	ts-notification-service	NOT_RECOVERED
81	ServiceDependency	ProviderDestination	/notification/preserve_success	NOT_RECOVERED
82	ServiceDependency	ProviderDestination	/notification/order_create_success	NOT_RECOVERED
83	ServiceDependency	ProviderDestination	/notification/order_changed_success	NOT_RECOVERED
84	Endpoint	RequestURI	/api/v1/inside_pay_service/inside_payment	NOT_RECOVERED
85	ServiceDependency	ProviderName	ts-inside-payment-service	NOT_RECOVERED
86	ServiceDependency	ProviderDestination	/api/v1/inside_pay_service/inside_payment	NOT_RECOVERED
87	Endpoint	RequestURI	/payment/pay	NOT_RECOVERED
88	Endpoint	RequestURI	/payment/addMoney	NOT_RECOVERED
89	Endpoint	RequestURI	/payment/query	NOT_RECOVERED
90	ServiceDependency	ProviderName	ts-payment-service	NOT_RECOVERED
91	ServiceDependency	ProviderDestination	/payment/pay	NOT_RECOVERED
92	ServiceDependency	ProviderDestination	/payment/addMoney	NOT_RECOVERED
93	ServiceDependency	ProviderDestination	/payment/query	NOT_RECOVERED

94	Endpoint	RequestURI	/rebook	NOT_RECOVERED
95	Endpoint	RequestURI	/api/v1/rebookservice/rebook	NOT_RECOVERED
96	Endpoint	RequestURI	/api/v1/rebookservice/rebook/difference	NOT_RECOVERED
97	ServiceDependency	ProviderName	ts-rebook-service	NOT_RECOVERED
98	ServiceDependency	ProviderDestination	/rebook	NOT_RECOVERED
99	ServiceDependency	ProviderDestination	/api/v1/rebookservice/rebook	NOT_RECOVERED
100	ServiceDependency	ProviderDestination	/api/v1/rebookservice/rebook/difference	NOT_RECOVERED
101	Endpoint	RequestURI	/api/v1/cancelservice/cancel	NOT_RECOVERED
102	Endpoint	RequestURI	/api/v1/cancelservice/cancel/refound	NOT_RECOVERED
103	ServiceDependency	ProviderName	ts-cancel-service	NOT_RECOVERED
104	ServiceDependency	ProviderDestination	/api/v1/cancelservice/cancel	NOT_RECOVERED
105	ServiceDependency	ProviderDestination	/api/v1/cancelservice/cancel/refound	NOT_RECOVERED
106	Endpoint	RequestURI	/api/v1/stationservice/stations/name	NOT_RECOVERED
107	ServiceDependency	ProviderName	ts-station-service	NOT_RECOVERED
108	ServiceDependency	ProviderDestination	/api/v1/stationservice/stations/name	NOT_RECOVERED
109	Endpoint	RequestURI	/route/createAndModify	NOT_RECOVERED
110	Endpoint	RequestURI	/route/delete	NOT_RECOVERED
111	Endpoint	RequestURI	/route/queryAll	NOT_RECOVERED
112	Endpoint	RequestURI	/route/queryById	NOT_RECOVERED
113	Endpoint	RequestURI	/route/queryByStartAndTerminal	NOT_RECOVERED
114	ServiceDependency	ProviderName	ts-route-service	NOT_RECOVERED
115	ServiceDependency	ProviderDestination	/route/createAndModify	NOT_RECOVERED
116	ServiceDependency	ProviderDestination	/route/delete	NOT_RECOVERED
117	ServiceDependency	ProviderDestination	/route/queryAll	NOT_RECOVERED
118	ServiceDependency	ProviderDestination	/route/queryById	NOT_RECOVERED
119	ServiceDependency	ProviderDestination	/route/queryByStartAndTerminal	NOT_RECOVERED
120	Endpoint	RequestURI	/api/v1/assuranceservice/assurances/types	NOT_RECOVERED
121	Endpoint	RequestURI	/assurance/getAssuranceById	NOT_RECOVERED
122	Endpoint	RequestURI	/assurance/findAssuranceByOrderId	NOT_RECOVERED
123	Endpoint	RequestURI	/assurance/findAll	NOT_RECOVERED
124	Endpoint	RequestURI	/assurance/create	NOT_RECOVERED
125	Endpoint	RequestURI	/assurance/deleteAssurance	NOT_RECOVERED
126	Endpoint	RequestURI	/assurance/deleteAssuranceByOrderId	NOT_RECOVERED
127	Endpoint	RequestURI	/assurance/modifyAssurance	NOT_RECOVERED
128	ServiceDependency	ProviderName	ts-assurance-service	NOT_RECOVERED
129	ServiceDependency	ProviderDestination	/api/v1/assuranceservice/assurances/types	NOT_RECOVERED
130	ServiceDependency	ProviderDestination	/assurance/getAssuranceById	NOT_RECOVERED
131	ServiceDependency	ProviderDestination	/assurance/findAssuranceByOrderId	NOT_RECOVERED
132	ServiceDependency	ProviderDestination	/assurance/findAll	NOT_RECOVERED
133	ServiceDependency	ProviderDestination	/assurance/create	NOT_RECOVERED
134	ServiceDependency	ProviderDestination	/assurance/deleteAssurance	NOT_RECOVERED
135	ServiceDependency	ProviderDestination	/assurance/deleteAssuranceByOrderId	NOT_RECOVERED
136	ServiceDependency	ProviderDestination	/assurance/modifyAssurance	NOT_RECOVERED
137	Endpoint	RequestURI	/office/getRegionList	NOT_RECOVERED

138	Endpoint	RequestURI	/office/getAll	NOT_RECOVERED
139	Endpoint	RequestURI	/office/getSpecificOffices	NOT_RECOVERED
140	Endpoint	RequestURI	/office/addOffice	NOT_RECOVERED
141	Endpoint	RequestURI	/office/deleteOffice	NOT_RECOVERED
142	Endpoint	RequestURI	/office/updateOffice	NOT_RECOVERED
143	ServiceDependency	ProviderName	ts-ticket-office-service	NOT_RECOVERED
144	ServiceDependency	ProviderDestination	/office/getRegionList	NOT_RECOVERED
145	ServiceDependency	ProviderDestination	/office/getAll	NOT_RECOVERED
146	ServiceDependency	ProviderDestination	/office/getSpecificOffices	NOT_RECOVERED
147	ServiceDependency	ProviderDestination	/office/addOffice	NOT_RECOVERED
148	ServiceDependency	ProviderDestination	/office/deleteOffice	NOT_RECOVERED
149	ServiceDependency	ProviderDestination	/office/updateOffice	NOT_RECOVERED
150	Endpoint	RequestURI	/travelPlan/getTransferResult	NOT_RECOVERED
151	Endpoint	RequestURI	/api/v1/travelplanservice/travelPlan/cheapest	NOT_RECOVERED
152	Endpoint	RequestURI	/api/v1/travelplanservice/travelPlan/quickest	NOT_RECOVERED
153	Endpoint	RequestURI	/api/v1/travelplanservice/travelPlan/minStation	NOT_RECOVERED
154	ServiceDependency	ProviderName	ts-travel-plan-service	NOT_RECOVERED
155	ServiceDependency	ProviderDestination	/travelPlan/getTransferResult	NOT_RECOVERED
156	ServiceDependency	ProviderDestination	/api/v1/travelplanservice/travelPlan/cheapest	NOT_RECOVERED
157	ServiceDependency	ProviderDestination	/api/v1/travelplanservice/travelPlan/quickest	NOT_RECOVERED
158	ServiceDependency	ProviderDestination	/api/v1/travelplanservice/travelPlan/minStation	NOT_RECOVERED
159	Endpoint	RequestURI	/api/v1/consignservice/consigns	NOT_RECOVERED
160	Endpoint	RequestURI	/api/v1/consignservice/consigns/account	NOT_RECOVERED
161	ServiceDependency	ProviderName	ts-consign-service	NOT_RECOVERED
162	ServiceDependency	ProviderDestination	/api/v1/consignservice/consigns	NOT_RECOVERED
163	ServiceDependency	ProviderDestination	/api/v1/consignservice/consigns/account	NOT_RECOVERED
164	Endpoint	RequestURI	/getVoucher	NOT_RECOVERED
165	ServiceDependency	ProviderName	ts-voucher-service	NOT_RECOVERED
166	ServiceDependency	ProviderDestination	/getVoucher	NOT_RECOVERED
167	Endpoint	RequestURI	/routePlan/minStopStations	NOT_RECOVERED
168	Endpoint	RequestURI	/routePlan/cheapestRoute	NOT_RECOVERED
169	Endpoint	RequestURI	/routePlan/quickestRoute	NOT_RECOVERED
170	ServiceDependency	ProviderName	ts-route-plan-service	NOT_RECOVERED
171	ServiceDependency	ProviderDestination	/routePlan/minStopStations	NOT_RECOVERED
172	ServiceDependency	ProviderDestination	/routePlan/cheapestRoute	NOT_RECOVERED
173	ServiceDependency	ProviderDestination	/routePlan/quickestRoute	NOT_RECOVERED
174	Endpoint	RequestURI	/api/v1/foodservice/foods	NOT_RECOVERED
175	Endpoint	RequestURI	/food/createFoodOrder	NOT_RECOVERED
176	Endpoint	RequestURI	/food/cancelFoodOrder	NOT_RECOVERED
177	Endpoint	RequestURI	/food/updateFoodOrder	NOT_RECOVERED
178	Endpoint	RequestURI	/food/findAllFoodOrder	NOT_RECOVERED
179	Endpoint	RequestURI	/food/findFoodOrderByOrderId	NOT_RECOVERED
180	ServiceDependency	ProviderName	ts-food-service	NOT_RECOVERED
181	ServiceDependency	ProviderDestination	/api/v1/foodservice/foods	NOT_RECOVERED

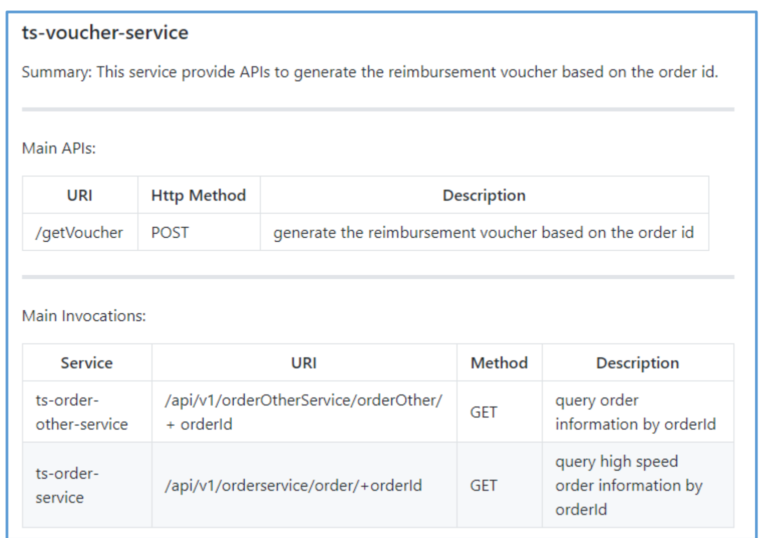
182	ServiceDependency	ProviderDestination	/food/createFoodOrder	NOT_RECOVERED
183	ServiceDependency	ProviderDestination	/food/cancelFoodOrder	NOT_RECOVERED
184	ServiceDependency	ProviderDestination	/food/updateFoodOrder	NOT_RECOVERED
185	ServiceDependency	ProviderDestination	/food/findAllFoodOrder	NOT_RECOVERED
186	ServiceDependency	ProviderDestination	/food/findFoodOrderByOrderId	NOT_RECOVERED
187	Endpoint	RequestURI	/news-service/news	NOT_RECOVERED
188	ServiceDependency	ProviderName	ts-news-service	NOT_RECOVERED
189	ServiceDependency	ProviderDestination	/news-service/news	NOT_RECOVERED
190	Endpoint	RequestURI	/api/v1/adminbasicservice/adminbasic/contacts	NOT_RECOVERED
191	Endpoint	RequestURI	/api/v1/adminbasicservice/adminbasic/stations	NOT_RECOVERED
192	Endpoint	RequestURI	/api/v1/adminbasicservice/adminbasic/trains	NOT_RECOVERED
193	Endpoint	RequestURI	/api/v1/adminbasicservice/adminbasic/prices	NOT_RECOVERED
194	Endpoint	RequestURI	/api/v1/adminbasicservice/adminbasic/configs	NOT_RECOVERED
195	ServiceDependency	ProviderName	ts-admin-basic-info-service	NOT_RECOVERED
196	ServiceDependency	ProviderDestination	/api/v1/adminbasicservice/adminbasic/contacts	NOT_RECOVERED
197	ServiceDependency	ProviderDestination	/api/v1/adminbasicservice/adminbasic/stations	NOT_RECOVERED
198	ServiceDependency	ProviderDestination	/api/v1/adminbasicservice/adminbasic/trains	NOT_RECOVERED
199	ServiceDependency	ProviderDestination	/api/v1/adminbasicservice/adminbasic/prices	NOT_RECOVERED
200	ServiceDependency	ProviderDestination	/api/v1/adminbasicservice/adminbasic/configs	NOT_RECOVERED
201	Endpoint	RequestURI	/api/v1/adminorderservice/adminorder	NOT_RECOVERED
202	ServiceDependency	ProviderName	ts-admin-order-service	NOT_RECOVERED
203	ServiceDependency	ProviderDestination	/api/v1/adminorderservice/adminorder	NOT_RECOVERED
204	Endpoint	RequestURI	/api/v1/adminrouteservice/adminroute	NOT_RECOVERED
205	ServiceDependency	ProviderName	ts-admin-route-service	NOT_RECOVERED
206	ServiceDependency	ProviderDestination	/api/v1/adminrouteservice/adminroute	NOT_RECOVERED
207	Endpoint	RequestURI	/api/v1/admintravelservice/admintravel	NOT_RECOVERED
208	ServiceDependency	ProviderName	ts-admin-travel-service	NOT_RECOVERED
209	ServiceDependency	ProviderDestination	/api/v1/admintravelservice/admintravel	NOT_RECOVERED
210	Endpoint	RequestURI	/api/v1/adminuserservice/users	NOT_RECOVERED
211	ServiceDependency	ProviderName	ts-admin-user-service	NOT_RECOVERED
212	ServiceDependency	ProviderDestination	/api/v1/adminuserservice/users	NOT_RECOVERED

Microservice “ts-voucher-service”: By checking the documentation of “ts-voucher-service” provided in Figure 8-29, one can deduce that this microservice is functional (business), has one endpoint as well as interaction (dependency) with two microservices, “ts-order-service” and “ts-order-other-service”. Moreover, by checking the documented diagram in Figure 8-2, it can be seen that “ts-voucher-service” makes use of three types of infrastructure, service registry and discovery, logging, and tracing server. To facilitate the comparison, the documented architecture is described in terms of PIM elements, as illustrated in Table 8-10.

- **Consistent with documentation:** By comparing Table 8-3 for the recovered PIM instance for "ts-voucher-service" to Table 8-10 for the expected PIM elements, it is clear that MiSAR, out of 13 documented elements, correctly recovered the container element and inaccurately recovered the functional microservice element as a microservice. It only captured the existence of this microservice with the help of the Docker Compose artefact and the name of the build directory acquired from the parser.

- **Additional elements:** With reference to Table 8-3, it can be seen that MiSAR recovered one undocumented service dependency element with "ts-voucher-mysql" data store (ID=3 in Table 8-3). To ensure the validity of this extra element, I checked the Generating PSM attribute that is attached to it, as indicated in Figure 8-30. It refers to a Docker Container Link element extracted from the Docker Compose file, as shown in figures 8-31 and 8-32.

- **Missed elements:** Apparently, MiSAR failed to recover infrastructure components, endpoint and some service dependencies for "ts-voucher-service". In fact, the project "ts-voucher-service" is a "Python" application and MiSAR does not yet transform Python source artefacts.



ts-voucher-service

Summary: This service provide APIs to generate the reimbursement voucher based on the order id.

Main APIs:

URI	Http Method	Description
/getVoucher	POST	generate the reimbursement voucher based on the order id

Main Invocations:

Service	URI	Method	Description
ts-order-other-service	/api/v1/orderOtherService/orderOther/ + orderId	GET	query order information by orderId
ts-order-service	/api/v1/orderservice/order/+orderId	GET	query high speed order information by orderId

Figure 8-29: TrainTicket's documentation for "ts-voucher-service".

Table 8-10: Expected elements for “ts-voucher-service” as per the documentation vs MiSAR result.

ID	Element Name	Element Attribute	Attribute Value	MiSAR Output
1	Container	ContainerName	ts-voucher-service	RECOVERED
2	FunctionalMicroservice	MicroserviceName	ts-voucher-service	MICROSERVICE
3	InfrastructureClientComponent	Category	Service_Routing_Pattern_Registry_and_Discovery	NOT_RECOVERED
4	InfrastructureClientComponent	Category	Observability_Pattern_Distributed_Tracing	NOT_RECOVERED
5	InfrastructurePatternComponent	Category	Observability_Pattern_Application_Metrics_Logging	NOT_RECOVERED
6	InfrastructurePatternComponent	Category	Observability_Pattern_Application_Metrics_Generation	NOT_RECOVERED
7	Endpoint	RequestURI	POST /getVoucher	NOT_RECOVERED
8	ServiceMessage	MessageType	RESPONSE	NOT_RECOVERED
9	ServiceMessage	BodySchema	{"type":"object","properties":{"orderId":{"type":"string"},"…	NOT_RECOVERED
10	ServiceDependency	ProviderName	ts-order-service	NOT_RECOVERED
11	ServiceDependency	ProviderDestination	GET /api/v1/orderservice/order/{orderId}	NOT_RECOVERED
12	ServiceDependency	ProviderName	ts-order-other-service	NOT_RECOVERED
13	ServiceDependency	ProviderDestination	GET /api/v1/orderOtherService/orderOther/{orderId}	NOT_RECOVERED

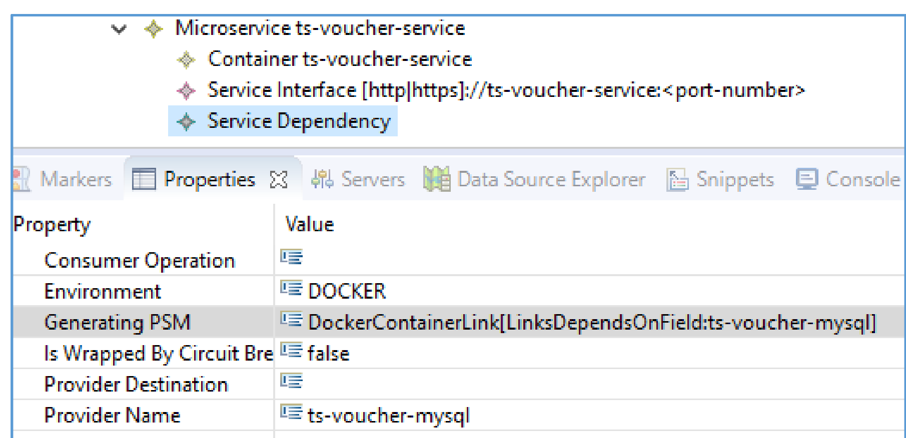


Figure 8-30: Generating PSM attribute for recovered service dependency element.

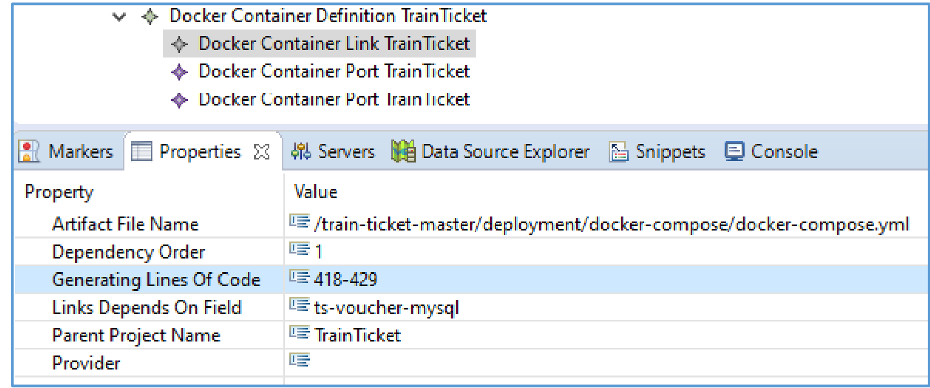


Figure 8-31: Docker Container Link instance retrieved for “ts-voucher-service”.

```
github.com/FudanSELab/train-ticket/blob/master/docker-compose.yml#L418-L429
409   ts-voucher-mysql:
410         image: mysql
411         expose:
412           - "3306"
413         environment:
414           MYSQL_ROOT_PASSWORD: root
415         networks:
416           - my-network
417
418   ts-voucher-service:
419         build: ts-voucher-service
420         image: ts/ts-voucher-service
421         restart: always
422         ports:
423           - 16101:16101
424         depends_on:
425           - ts-voucher-mysql
426         volumes:
427           - /var/lib/mysql
428         networks:
429           - my-network
```

Figure 8-32: Lines that generated DockerContainerLink instance for “ts-voucher-service”.

Microservice “ms-monitoring-core”: There is no documentation for the “ms-monitoring-core” microservice in the wiki page. However, in the documented architecture diagram in Figure 8-2, the “ms-monitoring-core” microservice is implied by the “Monitoring” infrastructure at the bottom layer. In addition, the documented architecture diagram implies that “ms-monitoring-core” has dependencies with all 42 backend microservices. No other information can be directly deduced from the available documentation. To facilitate the comparison, the documented architecture is described in terms of PIM elements, as illustrated in Table 8-11.

- **Consistent with documentation**: By comparing Table 8-4 showing the recovered PIM instance for “ms-monitoring-core” to Table 8-11 showing the expected PIM elements, it is clear that MiSAR, out of 45 expected elements,

correctly recovered the container element and inaccurately recovered the infrastructure microservice element as a microservice. It only captured the existence of this microservice with the help of the POM build file artefact and the name of the build directory acquired from the parser.

- **Additional elements:** With reference to Table 8-4, it can be seen that MiSAR recovered many undocumented elements, including one infrastructure pattern component element with category "Observability_Pattern_Application_ Metrics_Generation", as well as five endpoint elements along with their response service message elements. These elements were also recovered in "ts-auth-service". To ensure their validity, I checked the Generating PSM attribute attached to each of them and found that they all were generated by the transformation of one dependency library with library name "spring-boot-starter-actuator" in the POM build file. This Spring Actuator library enables the microservice to generate and expose its health metrics. Hence, it is transformed into a metric generation infrastructure component in addition to a set of production endpoints.

- **Missed elements:** Apparently, MiSAR failed to recover infrastructure components and service dependencies for "ms-monitoring-core". MiSAR mapping rules are currently based on applications that make use of distributed application frameworks such as Spring Boot, Spring Cloud and Netflix OSS, because they have libraries, annotations and methods with distinguished identifiers that act as keywords to indicate infrastructure and other architecture elements. Alternatively, "ms-monitoring-core" is a Java Spring Boot application which implements monitoring infrastructure with developer-specific logic that doesn't utilise any out-of-the-box monitoring framework such as ELK or Prometheus (which are supported by MiSAR).

Table 8-11: Expected elements for “ms-monitoring-core” as per the documentation vs MiSAR result.

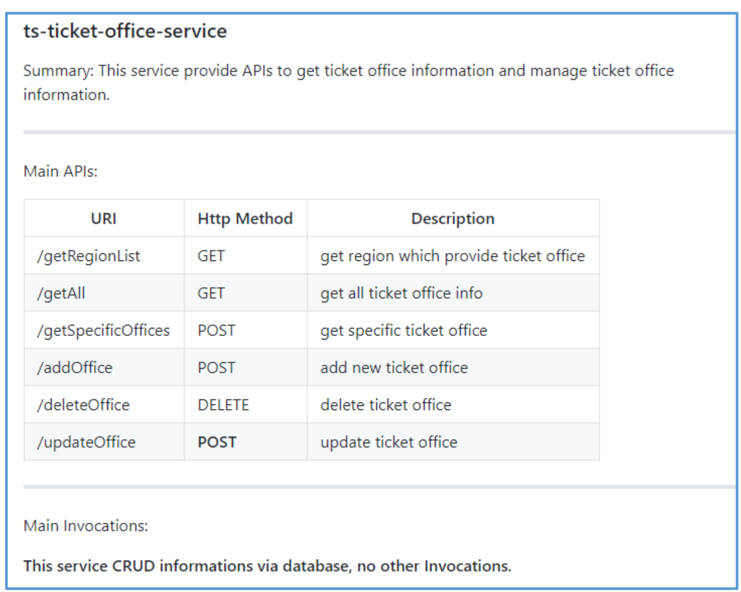
ID	Element Name	Element Attribute	Attribute Value	MiSAR Output
1	Container	ContainerName	ms-monitoring-core	RECOVERED
2	InfrastructureMicroservice	MicroserviceName	ms-monitoring-core	MICROSERVICE
3	InfrastructureServerComponent	Category	Observability_Pattern_Application_Metrics_Monitoring	NOT_RECOVERED
4	ServiceDependency	ProviderName	ts-admin-basic-info-service	NOT_RECOVERED
5	ServiceDependency	ProviderName	ts-admin-order-service	NOT_RECOVERED
6	ServiceDependency	ProviderName	ts-admin-route-service	NOT_RECOVERED
7	ServiceDependency	ProviderName	ts-admin-travel-service	NOT_RECOVERED
8	ServiceDependency	ProviderName	ts-admin-user-service	NOT_RECOVERED
9	ServiceDependency	ProviderName	ts-assurance-service	NOT_RECOVERED
10	ServiceDependency	ProviderName	ts-auth-service	NOT_RECOVERED
11	ServiceDependency	ProviderName	ts-basic-service	NOT_RECOVERED
12	ServiceDependency	ProviderName	ts-cancel-service	NOT_RECOVERED
13	ServiceDependency	ProviderName	ts-config-service	NOT_RECOVERED
14	ServiceDependency	ProviderName	ts-consign-price-service	NOT_RECOVERED
15	ServiceDependency	ProviderName	ts-consign-service	NOT_RECOVERED
16	ServiceDependency	ProviderName	ts-contacts-service	NOT_RECOVERED
17	ServiceDependency	ProviderName	ts-execute-service	NOT_RECOVERED
18	ServiceDependency	ProviderName	ts-food-map-service	NOT_RECOVERED
19	ServiceDependency	ProviderName	ts-food-service	NOT_RECOVERED
20	ServiceDependency	ProviderName	ts-inside-payment-service	NOT_RECOVERED
21	ServiceDependency	ProviderName	ts-news-service	NOT_RECOVERED
22	ServiceDependency	ProviderName	ts-notification-service	NOT_RECOVERED
23	ServiceDependency	ProviderName	ts-order-other-service	NOT_RECOVERED
24	ServiceDependency	ProviderName	ts-order-service	NOT_RECOVERED
25	ServiceDependency	ProviderName	ts-payment-service	NOT_RECOVERED
26	ServiceDependency	ProviderName	ts-preserve-other-service	NOT_RECOVERED
27	ServiceDependency	ProviderName	ts-preserve-service	NOT_RECOVERED
28	ServiceDependency	ProviderName	ts-price-service	NOT_RECOVERED
29	ServiceDependency	ProviderName	ts-rebook-service	NOT_RECOVERED
30	ServiceDependency	ProviderName	ts-route-plan-service	NOT_RECOVERED
31	ServiceDependency	ProviderName	ts-route-service	NOT_RECOVERED
32	ServiceDependency	ProviderName	ts-seat-service	NOT_RECOVERED
33	ServiceDependency	ProviderName	ts-security-service	NOT_RECOVERED
34	ServiceDependency	ProviderName	ts-station-service	NOT_RECOVERED
35	ServiceDependency	ProviderName	ts-ticket-office-service	NOT_RECOVERED
36	ServiceDependency	ProviderName	ts-ticketinfo-service	NOT_RECOVERED
37	ServiceDependency	ProviderName	ts-train-service	NOT_RECOVERED
38	ServiceDependency	ProviderName	ts-travel-plan-service	NOT_RECOVERED
39	ServiceDependency	ProviderName	ts-travel-service	NOT_RECOVERED
40	ServiceDependency	ProviderName	ts-travel2-service	NOT_RECOVERED

41	ServiceDependency	ProviderName	ts-ui-dashboard	NOT_RECOVERED
42	ServiceDependency	ProviderName	ts-user-service	NOT_RECOVERED
43	ServiceDependency	ProviderName	ts-verification-code-service	NOT_RECOVERED
44	ServiceDependency	ProviderName	ts-voucher-service	NOT_RECOVERED
45	ServiceDependency	ProviderName	jaeger	NOT_RECOVERED

Microservice “ts-ticket-office-service”: By checking the documentation of “ts-ticket-office-service” provided in Figure 8-33, one can deduce that this microservice is functional (business), has six endpoints and uses a data store infrastructure. Moreover, by checking the documented diagram in Figure 8-2, “ts-ticket-office-service” also makes use of three types of infrastructure, service registry and discovery, logging, and tracing server. To facilitate the comparison, the documented architecture is described in terms of PIM elements, as illustrated in Table 8-12.

- **Consistent with documentation:** By comparing Table 8-5 for the recovered PIM instance for “ts-ticket-office-service” to Table 8-12 for the expected PIM elements, it is clear that MiSAR, out of 26 expected elements, correctly recovered the container element and inaccurately recovered the functional microservice element as a microservice. It only captured the existence of this microservice with the help of the Docker Compose artefact and the name of the build directory acquired from the parser.

- **Missed elements:** Apparently, MiSAR failed to recover infrastructure components, endpoints and service dependencies for “ts-ticket-office-service”. In fact, the project “ts-ticket-office-service” is a “Node.js” application and MiSAR does not yet transform JavaScript source artefacts.


ts-ticket-office-service

Summary: This service provide APIs to get ticket office information and manage ticket office information.

Main APIs:

URI	Http Method	Description
/getRegionList	GET	get region which provide ticket office
/getAll	GET	get all ticket office info
/getSpecificOffices	POST	get specific ticket office
/addOffice	POST	add new ticket office
/deleteOffice	DELETE	delete ticket office
/updateOffice	POST	update ticket office

Main Invocations:

This service CRUD informations via database, no other Invocations.

Figure 8-33: TrainTicket's documentation for "ts-ticket-office-service".

Table 8-12: Expected elements for "ts-ticket-office-service" as per the documentation vs MiSAR result.

ID	Element Name	Element Attribute	Attribute Value	MiSAR Output
1	Container	ContainerName	ts-ticket-office-service	RECOVERED
2	FunctionalMicroservice	MicroserviceName	ts-ticket-office-service	MICROSERVICE
3	InfrastructureClientComponent	Category	Service_Routing_Pattern_Registry_and_Discovery	NOT_RECOVERED
4	InfrastructureClientComponent	Category	Observability_Pattern_Distributed_Tracing	NOT_RECOVERED
5	InfrastructureClientComponent	Category	Development_Pattern_Data_Persistence	NOT_RECOVERED
6	InfrastructurePatternComponent	Category	Observability_Pattern_Application_Metrics_Logging	NOT_RECOVERED
7	InfrastructurePatternComponent	Category	Observability_Pattern_Application_Metrics_Generation	NOT_RECOVERED
8	Endpoint	RequestURI	GET /getRegionList	NOT_RECOVERED
9	ServiceMessage	MessageType	RESPONSE	NOT_RECOVERED
10	ServiceMessage	BodySchema	{"type":"array","items":{"type":"object"," properties":{"prov...	NOT_RECOVERED
11	Endpoint	RequestURI	GET /getAll	NOT_RECOVERED
12	ServiceMessage	MessageType	RESPONSE	NOT_RECOVERED
13	ServiceMessage	BodySchema	{"type":"array","items":{"type":"object"," properties":{"prov...	NOT_RECOVERED
14	Endpoint	RequestURI	POST /getSpecificOffices	NOT_RECOVERED
15	ServiceMessage	MessageType	REQUEST	NOT_RECOVERED
16	ServiceMessage	BodySchema	{"type":"object","properties":{"province":{ "type":"string"},"…	NOT_RECOVERED
17	Endpoint	RequestURI	POST /addOffice	NOT_RECOVERED
18	ServiceMessage	MessageType	REQUEST	NOT_RECOVERED

19	ServiceMessage	BodySchema	{"type":"object","properties":{"province":{ "type":"string"},"…	NOT_RECOVERED
20	Endpoint	RequestURI	DELETE /deleteOffice	NOT_RECOVERED
21	ServiceMessage	MessageType	REQUEST	NOT_RECOVERED
22	ServiceMessage	BodySchema	{"type":"object","properties":{"province":{ "type":"string"},"…	NOT_RECOVERED
23	Endpoint	RequestURI	POST /updateOffice	NOT_RECOVERED
24	ServiceMessage	MessageType	REQUEST	NOT_RECOVERED
25	ServiceMessage	BodySchema	{"type":"object","properties":{"province":{ "type":"string"},"…	NOT_RECOVERED
26	ServiceDependency	ProviderName	ts-ticket-office-mongo	NOT_RECOVERED

Microservice "ts-news-service": There is no documentation for the "ts-news-service" microservice in the wiki page. However, by checking the documented diagram in Figure 8-2, it can be seen that the "ts-news-service" microservice has an interaction (service dependency) with the "ts-travel-service" microservice. Also, it makes use of three types of infrastructure: service registry and discovery, logging, and tracing server. By inspecting the "main.go" script file, I found one endpoint named "hello" for the microservice. To facilitate the comparison, the documented architecture is described in terms of PIM elements, as illustrated in Table 8-13.

- **Consistent with documentation:** By comparing Table 8-6 for the recovered PIM instance for "ts-news-service" to Table 8-13 for the expected PIM elements, it is clear that MiSAR, out of 10 documented elements, correctly recovered the container element and inaccurately recovered the functional microservice element as a microservice. It only captured the existence of this microservice with the help of the Docker Compose artefact and the name of the build directory acquired from the parser.

- **Missed elements:** Apparently, MiSAR failed to recover infrastructure components, endpoint and service dependency for "ts-news-service". In fact, the project "ts-news-service" is a "Go" application and MiSAR does not yet transform Go source artefacts.

Table 8-13: Expected elements for “ts-news-service” as per the documentation vs MiSAR result.

ID	Element Name	Element Attribute	Attribute Value	MiSAR Output
1	Container	ContainerName	ts-news-service	RECOVERED
2	FunctionalMicroservice	MicroserviceName	ts-news-service	MICROSERVICE
3	InfrastructureClientComponent	Category	Service_Routing_Pattern_Registry_and_Discovery	NOT_RECOVERED
4	InfrastructureClientComponent	Category	Observability_Pattern_Distributed_Tracing	NOT_RECOVERED
5	InfrastructurePatternComponent	Category	Observability_Pattern_Application_Metrics_Logging	NOT_RECOVERED
6	InfrastructurePatternComponent	Category	Observability_Pattern_Application_Metrics_Generation	NOT_RECOVERED
7	Endpoint	RequestURI	GET /hello	NOT_RECOVERED
8	ServiceMessage	MessageType	RESPONSE	NOT_RECOVERED
9	ServiceMessage	BodySchema	{"type":"array","items":{"type":”object”,”properties”:{“Title”…	NOT_RECOVERED
10	ServiceDependency	ProviderName	ts-travel-service	NOT_RECOVERED

Microservice “jaeger”: There is no documentation for the “jaeger” microservice in the wiki page. However, it is mentioned in the home page of TrainTicket’s github that a tracing system has been provided that is based on Jaeger. Therefore, the “jaeger” microservice provides tracing infrastructure. Hence, in the documented architecture diagram in Figure 8-2, the “jaeger” microservice is implied by the “Traffic Management” infrastructure at the bottom layer. No other information can be directly deduced by the available documentation. To facilitate the comparison, the documented architecture is described in terms of PIM elements, as illustrated in Table 8-14.

- **Consistent with documentation:** By comparing Table 8-7 for the recovered PIM instance for “jaeger” to Table 8-14 for the expected PIM elements, it is clear that MiSAR, out of three documented elements, correctly recovered the container element and inaccurately recovered the infrastructure microservice element as a microservice. It only captured the existence of this microservice with the help of the Docker Compose artefact and the name of the build directory acquired from the parser.

- **Missed elements:** Apparently, MiSAR failed to recover an infrastructure component for “jaeger” since it does not yet support Jaeger tracing technology in its current repository of mapping rules.

Table 8-14: Expected elements for “jaeger” as per the documentation vs MiSAR result.

ID	Element Name	Element Attribute	Attribute Value	MiSAR Output
1	Container	ContainerName	jaeger	RECOVERED
2	InfrastructureMicroservice	MicroserviceName	jaeger	MICROSERVICE
3	InfrastructureServerComponent	Category	Observability_Pattern_Distributed_Tracing	NOT_RECOVERED

8.3.5. Results

The results are presented according to research questions presented in section 8.3.1.

- RQ1 *(degree of completeness of the recovered microservice architecture model)*
- RQ2 *(degree of correctness of the recovered microservice architecture model)*

In order to answered RQ1 and RQ2, recall, precision and F1-score metrics were calculated to measure, respectively, the completeness, correctness and overall accuracy of the recovery model. The following metrics were applied for every PIM element to assess the overall effectiveness of MiSAR:

1) **Recall:** this metric measures how completely MiSAR captures the existing architectural elements. It is calculated as follows:

$$Recall = \frac{Correctly\ recovered\ architectural\ elements\ (TP)}{Total\ documented\ architectural\ elements\ (\mathrm{TP + FN})}$$

2) **Precision:** this metric measures how correct (valid) the elements recovered by MiSAR are. It is calculated as follows:

$$Precision = \frac{Correctly\ recovered\ architectural\ elements\ (\mathrm{TP})}{Total\ recovered\ architectural\ elements\ (TP + FP)}$$

3) **F-measure:** this metric measures the overall quality of MiSAR’s recovery performance. It is an average of both recall and precision. It is calculated as follows:

$$F_Measure = 2 \times \frac{Recall * Precision}{Recall + Precision}$$

Where TP is the number of True Positives (correctly recovered elements) and FP is the number of False Positive (incorrectly recovered elements).

These metrics were calculated for every PIM element, as shown in Table 8-15. It can be seen from the results in Table 8-15, the overall effectiveness of MiSAR based on the F-measure is 89.41% for this case study based on the documentation. MiSAR also achieved a precision score of 99.55% of correct elements. The recall score is also high but lower than precision. The recall score indicates that MiSAR has recovered 81.20% of architectural elements. The lower recall is due to the large number of missed elements which, in turn, is due to encountering microservices with artefacts that belong to non-JVM platforms or that were developed with unconventional implementation (a.k.a. developer-specific logic), as discussed in section 8.3.4.

To illustrate, one of the partially recovered microservices was the gateway microservice, i.e. “ts-ui-dashboard”, which is supposed to have at least 40 Service Dependency elements (because it routes requests to all of the 40 business microservices), and 83 Endpoint elements (because it exposes the main endpoints of all the 40 business microservices). The “ts-ui-dashboard” microservice is built with HTML/JS artefacts plus an NGINX configuration artefact. Both kinds of artefacts are not yet supported by the MiSAR repository. The second partially recovered microservice was the monitoring microservice, i.e. “ms-monitoring-core”, which is supposed to have at least 42 Service Dependency elements because it requests the health and metrics endpoints of all the 42 business microservices as well as pulling their logs for monitoring purposes. Such a large count of missed elements contributed to the recorded drop in recall.

The recall and precision score achieved 100% for container elements recovered from Docker Compose and POM artefacts. This indicates that MiSAR can capture the existence of all microservices but it might miss the underlying elements of those microservices, such as their infrastructure components, endpoints and dependencies. The next highest recall score of 96.20% was achieved for Endpoint element because their recovery is based on a Java Method from the standard Rest Template class available in the Spring Boot, Spring Cloud framework.

Table 8-15: Evaluation metrics for MiSAR recovery of TrainTicket system

PIM Element	Expected Elements	Correctly Recovered	Incorrectly Recovered	Missed Elements	Missed Classified	Recall	Precision	F-Measure
Container	69	69	0	0	0	100%	100%	100%
InfrastructureMicroservice	30	27	3	0	3	90%	90%	90%
FunctionalMicroservice	39	36	3	0	3	92.31%	92.31%	92.31%
InfrastructureServerComponent	32	27	0	5	0	84.38%	100%	91.53%
InfrastructurePatternComponent & InfrastructureClientComponent	208	131	0	77	0	62.98%	100%	77.29%
Endpoint	474	456	0	94	0	96.20%	100%	98.06%
ServiceDependency	792	589	0	203	0	74.37%	100%	85.30%
All	1644	1335	6	379	6	81.20%	99.55%	89.41%
Annotation	Correctly recovered for Infrastructure/Functional microservice recovered as a microservice without classify its type. Missed elements for Infrastructure server component belonging to "ts-ui-dashboard", "ms-monitoring-core" and "jaeger" microservices. Missed elements for Infrastructure pattern/client component belonging to tracing and service registry/discovery client infrastructure of all backend microservices. PIM Element (Endpoint) Including the count of service message elements (if any) associated with each endpoint. Missed elements for Endpoint Including the 83 endpoints of the "ts-ui-dashboard" microservice alone. PIM Element (ServiceDependency) separately counting the Provider Name and Provider Destination attributes for each service dependency. Missed elements for ServiceDependency including 40 ProviderName and 83 ProviderDestination values in "ts-ui-dashboard" as well as 42 ProviderName values in "ms-monitoring-core" and one Provider Name = "jaeger" in 37 backend microservices.								

- ***RQ3: Is the execution of the MiSAR transformation efficient or not?***

In order to answer this question, I measured the runtime at each stage of the MiSAR recovery for TrainTicket by using the runtime metric as shown in Table 8-16.

1) **Runtime**: this metric measures the efficiency of MiSARs static approach after the automation of parsing and QVTo transformation.

Table 8-16: Time spent at each stage of MiSAR recovery for TrainTicket

Artefact Collection	Parsing	QVTo Transformation	Total
~ 2minutes	~ 5 minutes	~ 2 minutes	~ 9 minutes

It can be seen from the results in Table 8-16 the overall automatic MiSAR static recovery process took less than 9 minutes to complete, which indicates the efficiency of MiSAR's static approach after the automation of parsing and QVTo transformation. This time equation includes the two minutes spent on selecting the path of all mandatory artefacts in TrainTicket via the MiSAR parser (including three Docker Compose files, 42 microservice build directories and 39 build files), the five minutes required to finish automatic parsing and generating the PSM model of TrainTicket artefacts, and finally the less than two minutes required to generate the architecture PIM model via the Eclipse QVTo project.

Generally, the count of steps necessary to execute a transformation is proportional to the count of elements in the source model as well as the count of transformation rules applicable to those elements. As for MiSAR, with the help of implementation in Eclipse QVTo, the recovery process took less than two minutes to transform the source model that belongs to TrainTicket with size of 13 MB into a target model with a size of 427 KB.

8.3.6. Updates to MiSAR's Repository

As a result of this study and after the recovery process was completed and the generated model had been analysed, some limitations were noticed. This led to necessary updates to MiSAR's repository (PIM metamodel, PSM metamodel and mapping rules). To illustrate, the following mapping rules were added in order to overcome limitations noticed in the recovery of architecture elements related to Jaeger open tracing technology identified in the previous step:

1) One Docker hub image container with image field value which contains "jaegertracing" indicates one infrastructure server component with category value: observability pattern distributed tracing.
2) One dependency library with image field value which contains "opentracing-spring-jaeger-web-starter" indicates one infrastructure client component with category value: observability pattern distributed tracing.
3) One configuration property with fully qualified property name value which contains "opentracing.jaeger.udp-sender.host" and property value {provider-name} indicates one service dependency with provider name value: {provider-name}.

Technical limitations that are related to the use of a new framework, infrastructure and even non-Java programming languages can be solved simply by extending MiSAR's PSM metamodel and transformation rules to include these new encounters. However, when a developer-specific logic is encountered, such as the case of recovering the "ms-monitoring-core" microservice, a different approach needs to be considered. One recommendation could be the design of an additional input metamodel (e.g. UML Sequence Diagram) where particular sets of elements map to particular sets of infrastructure microservices and/or architectural elements. To illustrate, monitoring microservices tend to implement a common functionality that collects logs periodically from other microservices, parses them and finally visualises metrics. Therefore, if such an algorithm is recovered as a UML sequence diagram model, whether statically by source code analysis or dynamically by analysing tracing packets, then this microservice will eventually be recovered as a monitoring infrastructure microservice.

8.4. Summary

In this chapter, I present an evaluation of MiSAR artefacts and the results obtained through its application to the recovery of the TrainTicket system's architecture. The objective of this chapter was to show the usefulness of the MiSAR through a case study by measure the correctness, completeness and efficiency of architecture elements recovered by MiSAR against documentation. This chapter also presented the architecture recovery process through a step-by-step recovery experiment involving the use of a set of QVT mapping rules. The case study reports that MiSAR's transformation is efficient and able to obtain architectural models effectively. This case study has enabled the improvement and refinement of the MiSAR artefacts.

Chapter 9

Discussion

9.1. Introduction

This chapter discusses the findings of the studies presented in chapters 5, 6 and 8. Three separate studies were carried out in order to build and evaluate the MiSAR artefacts. The first study explored the MiSAR artefacts from empirical data by defining the initial artefacts of MiSAR, the metamodel and mapping rules from the microservice systems analysed. The second study explored the incremental enhancement of these artefacts from analysis of more microservice systems, which led to a more refined version of the MiSAR artefacts. The third study applied the refined version of the MiSAR artefacts via a large-scale microservice system. Section 9.2 discusses the results of studies 1 and 2, which answer the four research questions of the thesis. Section 9.3 discusses study 3. Section 9.4 spotlights how the findings of these studies are similar to or different from previous studies. Section 9.5 presents consideration and positive aspects of MiSAR, divided into six sub-sections: the degree of recovery MiSAR attains, MiSAR's efficiency and reliability, MiSAR's architectural expressiveness, backtracking support, the ability of MiSAR to discover the existence of non-JVM applications, and model traceability support. Finally, section 9.6 presents lessons to research community be learned concerning microservice architecture recovery.

9.2. Discussion of Study 1 and Study 2: Empirical Studies

This section discusses the results of study 1 and study 2 in terms of each thesis research question. The main problem I wanted to address in this thesis is the following: *What are the architecture recovery processes that allow software engineers to recover the architecture of microservice systems?* In order to answer this main research question, I conducted two studies that answer the sub-questions, as depicted in Figure 9-1. The first study developed MiSAR artefacts by defining the metamodel and mapping rules which correspond to the artefacts of the Model-Driven Engineering (MDE) approach,

as presented in Chapter 5. The second study enhanced and refined the MiSAR artefacts, as presented in Chapter 6.

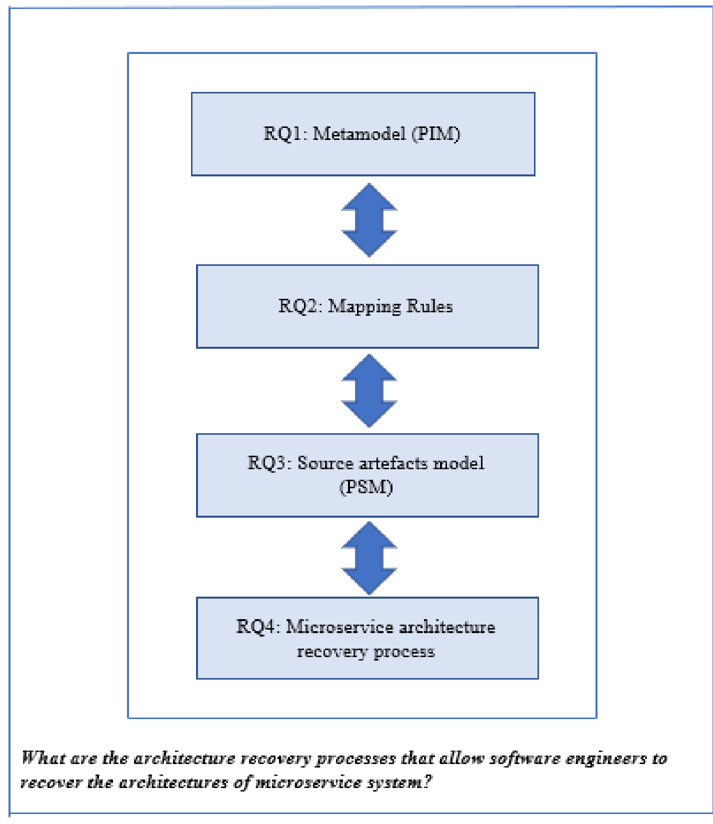


Figure 9-1: Research questions.

The research questions of this thesis are:

- **RQ1:** What are the microservice architectural elements/concepts that need to be present in metamodels in order to abstract microservice-based systems at a platform-independent model level?
- **RQ2:** What are the mapping rules that can transform microservice-based implementations into architectural models?
- **RQ3:** What are the suitable elements in the source model to be able to create a platform-specific model for the recovery of the architecture model?
- **RQ4:** What is an appropriate process/technique for microservice architecture recovery?

- **RQ1 and RQ2: metamodel and mapping rules**

These two questions were answered in parallel, first in study 1, which required understanding and defining the various concepts and elements that form a microservice architectural model and that define the characteristics of architectural models of a microservice system. This was followed by investigating the appropriate mapping rules between the source code of microservice implementation and the architectural model; these mapping rules transform the microservice system implemented in a technology-specific way to architectural elements in a technology-independent way. These questions were answered via the empirical study, which involved analysing eight open-source projects from the GitHub repository implemented in the Java and Spring Cloud frameworks that employed microservice architecture.

The aim of study 1 is to identify the concepts and elements needed to build metamodel concepts, that abstract a microservice-based system, and to develop mapping rules that derive a target model from the source model. To achieve this, study 1 addressed the following research questions:

- *What are the microservice architectural elements/concepts that are identified from the source code?*
- *What are the mapping rules between the source code of microservice implementations and the architectural model?*

Study 1 followed a manual recovery process. Initially, I wanted to follow the same steps as described in van Deursen et al. (2004), in terms of Recovery Design (RD) and Recovery Execution (RE). This kind of process begins by defining the architectural concepts, abstractions and concerns that can be recovered. Usually, abstractions and concerns are known beforehand, and architects extract and classify system data to map them to the architectural concepts. However, I noticed when I started this process that there is no standard metamodel for microservice architecture. Therefore, I opted to extract the data of the system and analyse it first, and then abstract the result into

architectural concepts. For the architectural concerns of microservices, there are standard ones.

From the results of the study, which included analysis of eight systems, I noticed that in the first case study analysed in the RD stage (Table 5-3, ID=1) I identified the architectural concepts **Microservice Architecture**, **Microservice**, **Functional Microservice**, **Infrastructure Microservice**, **Service Interface**, **Container**, **Service Operation**, **Load Balancer**, **Circuit Breaker**, **Endpoint**, **Data Store** and **Service Dependency**, but only two concepts, **Message Bus** and **Cache Store**, were discovered in the RE stage. In the RD stage, I identified 104 mapping rules. In the RE stage, I identified 164 new rules and refined 47 rules which were identified in the RD stage. These mapping rules use natural language, informal descriptions and a non-executable.

I noticed that the refinement of the mapping rules became less as I validated them with new case studies. The mapping of architectural elements from source artefact elements identified are not one-to-one; that is, many mapping rules can be applied to map one concept type. For example, nine mapping rules can be applied to generate the API Gateway architectural concept, as shown in Chapter 5, Table 5-5. Many of the newly added mapping rules were not related to new concepts but to the fact that different technologies can be used for the same architectural concepts.

To illustrate, the mapping rules presented in Table 5-5 support the recovery of the API Gateway concept implemented in three technologies, Apache HTTP, Netflix Zuul and Spring Sidecar. This makes the implementation of these mappings more complicated, and any future MiSAR tool should be able to identify and analyse a range of different technologies. All of the mapping rules use static code analysis. I also observed that the model recovered from these rules can be validated if a dynamic analysis is performed at system runtime. This is an important finding, as it demonstrates that many parts of a microservice architecture can be recovered statically, which is much easier than using dynamic analysis.

The initial MiSAR artefacts are: the PIM metamodel, which supports the creation of microservice architectural models and mapping rules, which map microservice

artefact elements (PSM) into the PIM metamodel in study 1. However, in study 2 a natural question to ask is "*To what extent do the current MiSAR artefacts such as the metamodel and the mapping rules satisfy microservice architecture recovery?*". I answered this question empirically by applying MiSAR artefacts to a new set of open-source microservice projects implemented in the Java, Docker and Spring Cloud frameworks. In this study, I aimed to validate that the existing MiSAR artefacts can recover an architectural model and that these artefacts can be enhanced. To achieve this, study 2 addressed the following research questions:

- *What are the enhancements that have to be performed to the existing MiSAR metamodel to represent more richly recovered architectural models of microservice systems?*
- *What enhancements have to be applied to the current MiSAR mapping rules that map Java and Spring Cloud microservice systems into architectural models?*

Nine additional open-source microservice systems were selected systematically, as I aimed in this study to address more complex systems that have more services than those addressed in study 1, such as systems that implement a synchronous/asynchronous inter-microservice interaction style, and integrate variation implementation of patterns. I manually applied the metamodel and the mapping rules presented in study 1 to every system incrementally and achieved improved artefacts.

In study 2, the initial PSM elements structure has been modified into PSM metamodel, this metamodel identifies the information that needs to be extracted from source artefacts. Like the PIM metamodel, the PSM metamodel consists of abstract artefact elements that can be easily extended in the future e.g. the abstract (Microservice Project) can have other sub-types than JavaSpringWebApplicationProject. For that purpose, I had to modify the structure of the mapping rules so that they can transform each particular PSM concept in the artefacts to particular PIM architecture concepts. Mapping rules become easy to implement with such a defined structure. This modification motivated the automation of the entire recovery process, presented in Chapter 7. As a result of this study, I identified requirements for a PIM metamodel that declares abstract microservice architecture elements, along with the new PSM

MiSAR metamodel and refinement of MiSAR mapping rules. The PIM metamodel concepts can be easily extended in the future e.g. the abstract element (Message Destination) can have other sub-types than Endpoint for synchronous service interface and Queue Listener for asynchronous service interface.

After that, I implemented the final version of the MiSAR PIM/PSM metamodels in Ecore format using the Eclipse Modelling Framework (EMF), and the mapping rules were implemented in the QVT operational transformation language using Eclipse M2M/QVTo, as presented in Chapter 7. Mapping rules were also improved further to transform architecture elements that do not have source artefacts available. Such mapping rules become augmented with hard-coded values for the attributes of these architectural elements. To illustrate, a microservice built from a Spring project having an Actuator library attached to it will expose production REST endpoints such as "/health" and "/metrics" at start-up. This information cannot be recovered statically from source artefacts as they are hidden (encoded) in the Actuator's library (executable). Therefore, I hard-coded in mapping rules the values of the Actuator's production endpoints from its public documentation. Some of these hard-coded mapping rules are demonstrated in listings 7-6 and 7-7.

As demonstrated in study 2, the majority of the current mapping rules were initially captured by study 1. Value added by the new case studies included new elements in the PIM metamodel, new mapping rules for new PIM elements, as well as new mapping rules that support different technologies and varying styles of implementation. There are variations in the implementation style of PIM elements; these variations could involve many PSM elements from different source artefacts, along with their mapping rules for defining the same PIM element. E.g. Reactive web application (WebFlux) or MVC web application, both have different architecture implementation style in the source code which are then abstracted and recovered later to an architectural element named Service operation concept. As an example of mapping rules variation that recovers same PIM concept, mapping rules in Table 6-7, Chapter 6, the rule R8 recovers the Service Operation concept, which is the exact output of R7 except that the input in R8 represents a reactive, non-blocking microservice. The nine case studies analysed in study 2 introduced the following new technologies to the current mapping rules: asynchronous message-based message

destinations and service dependencies, Neo4j Graph database, Spring Web Flux non-blocking reactive implementation, etc. MSAR at current stage recover the architecture of microservice based systems if developed with Spring Boot/Spring Cloud framework, due to the fact that all applications developed using Spring Boot/Spring Cloud framework were noticed to share a common core structure.

❖ **RQ3:** source artefacts model

In this section, I first discuss the particular information selected from the source artefacts in order to build a suitable PSM for the microservice-based system. Although this question has been answered in detail in study 2 by conducting the systematic steps (in Figure 6-2) for the identification of PSM metamodel (see Chapter 6), the analysis started early in study 1 at the stage of creating the initial mapping rules. At that time, I already noticed that not all implementation text carries information that indicates and expresses the architecture elements I identified in the PIM metamodel. Only the necessary information that contributed to the mapping rules to build the PSM metamodel (see Static Analysis in Appendix A, 2), while the unnecessary information was naturally filtered out and excluded. In other words, in order to create a concise representation of the source artefacts that best map to architecture elements of MiSAR's PIM metamodel, a partial coverage approach was employed. Later on, in study 3 (see Chapter 8), the resultant concise source model proved to perform faster and generated a more accurate architecture model.

Next, I discuss the primary differences between the two types of model transformation, text-to-model transformation and model-to-model transformation. Various abstraction levels can be used to transform text to model, as shown in Figure 9-2. It is possible to create a high-level PIM of the architecture from the code, which would then imply that the logic of the transformation will be of high complexity in terms of bridging the gap of details between the code and the PIM. On the other hand, a PSM can be created based on the textual code and then a PIM can be generated from the PSM. The model transformation from PSM to PIM contains less complexity. I conclude that with the help of PIM/PSM abstraction levels, it is possible to carry out textual transformations to any level of model abstraction. The main challenge lies in finding the level which is appropriate and then designing the transformation. The more abstract the model, the

more complexity will be involved in the transformation. Thus, the models diagram recovered at PSM to PIM level shows the suitable architectural models allowing microservice software to be properly recovered.

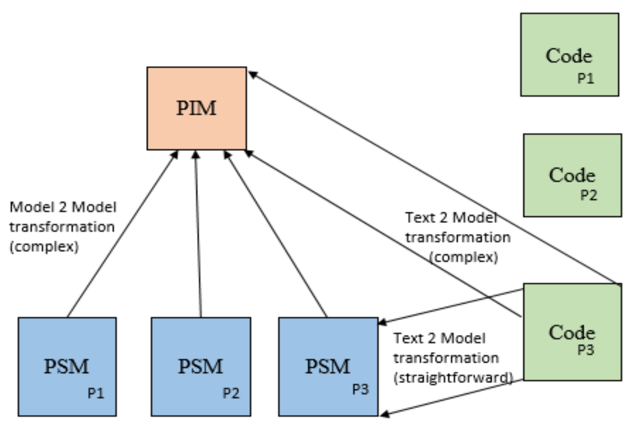


Figure 9-2: Transformations: text to model / model to model.

- **RQ4:** What is an appropriate process/technique for microservice architecture recovery?

In the first study, the architectural model was manually created from the system implementation, specifically in the recovery execution (RE) stage, by applying the first version of the metamodel and mapping rules, written in natural language, presented in Chapter 5 and Appendix B, 2 and 3, as I found that the manual application of mapping rules was difficult and inefficient in terms of time. After that, in the second study (Chapter 6 and 8), I developed and refined the recovery process into three steps to recover the architecture model of the microservice-based system, as depicted in Figure 8-1, Chapter 8.

The MiSAR recovery process aims to assist the activities of reverse engineering to recover the architecture of the software at different levels of abstraction. The current MiSAR recovery process consists of three steps that can be summarised as follows:

The first step is **artefact collection (semi-automatic)**; this step involves collecting artefacts (e.g. source code) and reviewing them to search for any artefact that may give information about the system. The second step is to **instantiate PSM instance (automatic)**; this step produces the required information to describe the software architecture. It extracts the static elements from the system's source code and other artefacts and eventually generates a PSM that conforms to the PSM metamodel. This step is executed either manually or using the MiSAR parser. The last step is to **recover PIM instance (automatic)**; this step populates the target model with a high-level abstraction of the system by applying the automated mapping rules implemented using the Eclipse QVTo project. The output of this stage is the architecture PIM that conforms to the PIM metamodel.

In addition, the architecture of the source microservice system has a graphical representation. Architecture models are mostly represented in an illustrative and graphical manner, unless they require some description, modelling or textual language. First, based on study 1, the final recovered model is visualised in two diagrams: the instance diagram and the architecture diagram. The instance diagram is equivalent to a UML object diagram that conforms to the PIM metamodel, where each PIM metamodel concept is equivalent to a UML class. The architecture diagram is equivalent to Newman's (2019) graphical notations diagram, as presented in Appendix B, 1. Second, in study 2, the recovered architecture in MiSAR is represented as an auto-generated PIM instance in XMI format and graphically represented at two levels: at an architectural level and at a microservice level, where each microservice has a more detailed view.

The architectural level, which reflects the recovered PIM instance, they include the high-level view of all microservices, their types and dependencies. In the architecture diagram at the architectural level, both the *Infrastructure Microservice* and *Functional Microservice* concepts are represented with a hexagon; the representation of the *Infrastructure Microservice* is distinguished by adding all of its *Infrastructure Server Components* as circles inside the hexagon in order to specify the composite category of that *Infrastructure Microservice* (see Figure 6-25 and Appendix B,1). A microservice-to-microservice *Service Dependency* is represented by a link connecting the two microservices, while a microservice-group-to-microservice *Service*

Dependency is represented by a box surrounding all the microservices that share the same *Service Dependency* with one microservice. The notations related to microservice level (see Figure 6-26 and Appendix B, 4) reflect the recovered PIM instance of an individual microservice, including its Service Interface, Service Operations, Messages Destinations (e.g. Endpoint and/or Queue Listener) and the Infrastructure Pattern Components.

After the recovery process is complete and the generated model is analysed, some limitations can be noticed. The main threat to MiSAR's validity is the implementation technology and implementation logic that are not supported in the current repository of mapping rules and metamodels. However, the Update MiSAR's Repository step comes to the rescue. In this case, the MiSAR repository (PIM/PSM metamodels and mapping rules) is updated with new elements found. However, a natural question to ask is "What parts of MiSAR repository need to be updated?". This question can be answered by providing the following scenarios:

- Limitations that are related to artefacts that are new to MiSAR, such as new languages (non-Java) or new source artefacts (non-Spring configuration files): in this case, the amendment will involve creating new PSM/PIM metamodel elements; these elements need to define the new language or new source artefact. Hence, mapping rules will also be added that map the PSM elements to PIM elements.
- Limitations that are related to new frameworks and infrastructure technologies not supported by MiSAR: if the source artefact of the technology is supported by MiSAR, then this can be resolved simply by adding mapping rules. However, if the source artefact of the technology not supported by MiSAR, then adding a new PSM metamodel for this new artefact and as well as new mapping rules is required.
- Limitations that are related to architectural concepts not supported by the MiSAR PIM metamodel: this can be resolved by adding new concepts to the PIM metamodel, amending/adding PSM metamodel elements that generate these PIM elements and defining mapping rules for them.

9.3. Discussion of Study 3: Evaluation

In study 3, I applied the final version of the MiSAR artefacts to a large-scale microservice system, involving a case study in an industry setting, to show the usefulness of the MiSAR elements and to evaluate the recovery approach. This study focuses on the integration of all MiSAR artefacts and applies them in order to obtain architectural models.

Through the first study in Chapter 5, it was possible to have an initial evaluation in the RE phase of the validity of architecture recovery based on initial MiSAR artefacts, i.e. the metamodel (first version) and mapping rules written in natural language. In an attempt to assess the applicability of MiSAR, I conducted manual architecture recovery for a case study named Piggy Metrics that has 17 microservices. The system was chosen with specific architectural patterns in mind. The recovered architecture that conforms to the metamodel (version 1) was represented using an instance diagram for a detailed description (e.g. see Figure 5-14) and a high-level architecture diagram (see Appendix B, 2) which only symbolises microservice type and RESTful endpoints that were involved in inter-service communication (dependencies). Those PIM elements recovered by MiSAR which belong to the high-level architecture diagram matched the documented architecture by 100%. However, when recovering infrastructure microservices such as "registry", their PIM instance models were missing the Service Operation elements but recovered the descendants (e.g. Circuit Breaker) disconnected as in Figure 5-14. The reason was that infrastructure microservices abstract the implementation of their operations, i.e. hidden implementation, due to the use of Spring Boot/Spring Cloud frameworks. This observation, in addition to other considerations, raised the need for MiSAR enhancements and hence led to the development of study 2.

Next, in the second study, as presented in Chapter 6, a second evaluation of MiSAR was conducted, with the aim of testing the performance of the automated architecture recovery using the new PSM metamodel, the enhanced PIM metamodel and the enhanced mapping rules implemented with Eclipse QVTo. The criteria for the selection of the case study concentrated mainly on the implementation of both synchronous and asynchronous inter-microservice communication styles (i.e. service

dependencies), the integration of polyglot technologies (e.g. multiple datastores, non-JVM applications, reactive programming, etc.), as well as multiple configuration profiles, in order to ensure that the validity and effectiveness of all enhancements made to MiSAR in this study are evaluated. The selected case study meeting these criteria was the Microservices Sample application, which consists of 14 microservices. The recovered architecture was represented using an instance diagram for detailed description, a high-level architecture diagram which only symbolises microservice type, infrastructure components, synchronous/asynchronous destinations that were involved in service dependencies, as well as the newly added microservice-level diagram which consists of the service interface, infrastructure pattern components, infrastructure client components, and a complete list of endpoints and queue listeners, along with their operations and service messages. The architecture model recovered by MiSAR demonstrated great consistency with the documentation, except for a few missed elements and, interestingly, there were some additional elements, as presented in Chapter 6. The missing elements were the infrastructure components and dependencies of only two infrastructure microservices with new technologies that are unfamiliar to the MiSAR repository. This was resolved simply by updating the mapping rules so that these technologies were included. Moreover, MiSAR managed to recover more information about the architecture and its elements than was documented, such as the recovery of one microservice as well as the service operations and business data messages of recovered microservices. This shows that MiSAR has the potential to provide expressive documentation for the recovered architecture elements, with the condition that mapping rules are comprehensive. Otherwise, human intervention comes to the rescue.

Finally, in the third study, as presented in Chapter 8, a third evaluation of MiSAR was conducted on a large, documented scale: a microservice-based application developed with a variety of technologies and implementations. The main goal of this test was to assess the effectiveness and efficiency of the final version of MiSAR's automated architecture recovery, as well as to highlight the threats to its validity. The selected case study, meeting the aforementioned test goals, was the TrainTicket application, which consists of 41 business microservices. TrainTicket offers comprehensive documentation in two means: an architecture diagram and a wiki page that describes the technologies used in the development of TrainTicket, as well as a complete list of

endpoints and invocations for every business microservice. However, I noticed that the available documentation corresponds to a subset of the total architecture elements represented by MiSAR's PIM metamodel. While the architecture diagram corresponds to Functional/Infrastructure microservices as well as Infrastructure Server/Client/Pattern components, it doesn't demonstrate, for instance, any Data Store containers, although they are defined in the Docker Compose artefact. Moreover, while the list of endpoints and invocations defined in the wiki page corresponds to the Endpoint and Service Dependency elements, respectively, it totally lacks representation of the Service Operation and Service Message elements. This encouraged me to add two undocumented architecture elements, namely the Data Store containers and endpoints' Service Messages, into the test cases to ensure more accurate evaluation of the MiSAR repository (i.e. mapping rules and metamodels). The overall automatic static recovery process took less than 9 minutes to complete.

In particular, MiSAR managed to capture the existence of all microservices in TrainTicket, as per the source artefacts (even those that are not documented); it was not able to recover the underlying elements of just six microservices. When analysing the six partially recovered microservices, I found that the artefacts that assisted in recovering their existence, represented by the Container, Microservice, Service Interface and Dependency elements, were the Docker Compose and/or POM build artefacts. The gateway microservice, i.e. "ts-ui-dashboard", and the monitoring service, i.e. "ms-monitoring-core", are an example of partially recovered microservices that contributed to the recorded drop in recall measurement. The "ms-monitoring-core" microservice differs from "ts-ui-dashboard" in that it meets the restrictions on the selected case studies I initially defined at the time of study 1. It has Java Spring artefacts that are supported by MiSAR. However, the monitoring task was implemented with a type of logic that does not follow the Spring Cloud standard logic, involving the use of specific keywords such as Java annotations and/or invocations to library methods.

Based on the previous discussion of partially recovered microservices, the main threat to MiSAR's validity is the technology and implementation that are not supported in the current repository of mapping rules and metamodels. However, the Update MiSAR's Repository step at the end of the architecture recovery process comes to the

rescue. Technical limitations that are related to the use of a new framework, infrastructure and even non-Java programming language can be solved simply by extending MiSAR's PSM metamodel and transformation rules to include these new encounters. However, when a developer-specific logic is encountered, such as the case of recovering the "ms-monitoring-core" microservice, a different approach needs to be considered. One recommendation is the design of an additional input metamodel (e.g. UML Sequence Diagram) where particular sets of elements map to particular sets of infrastructure microservices and/or architectural elements. To illustrate, monitoring microservices tend to implement a common functionality that collects logs periodically from other microservices, parses them, and finally visualises metrics. Therefore, if such an algorithm is recovered as a UML Sequence Diagram model, whether statically by source code analysis or dynamically by analysing tracing packets, then this microservice will eventually be recovered as a monitoring infrastructure microservice. On the other hand, after analysing the successful cases, such as "ts-auth-service", MiSAR was found to recover extra architecture elements when compared to the documented approaches. In particular, it provides information about the schema of request/response message(s) for every endpoint.

9.4. Previous Studies

This section spotlights how the findings of MiSAR are similar to or different from previous studies. Although significant software architecture recovery methods exist (Ducasse and Pollet, 2009; Raibulet et al., 2017), few of the current methods have their main focus on a system that specifically addresses microservice architecture. One of the few existing works related to the current project is MicroART (Granchelli, Cardarelli, Francesco, et al., 2017). MicroART is a microservice architecture recovery approach; similar to MiSAR approach, it uses model-driven engineering principles. MicroART was the only study found in the literature that tackles architecture recovery in microservices via model-driven engineering. In term of static analysis extraction, MicroART is similar to MiSAR, in that static extraction is mainly from source code repositories, where MicroART retrieves only information related to system name, developer information and service descriptors.

One key limitation of the MicroART approach is that it requires a software architect to manually refine the physical model generated in the first phase. The refining step resolves the consumer-to-proxy and proxy-to-provider interactions into consumer-to-provider interactions before the final logical architecture is generated. The main drawback of this approach is that the manual step slows down the recovery process. In MiSAR, the consumer-to-provider interactions (both in synchronous/asynchronous mode) can be directly recovered using MiSAR's mapping rules for the *Service Dependency* concept, as depicted in figures 9-3 and 9-4. Moreover, MicroART's DSL metamodel presented in Granchelli et al. (2017) does not define the asynchronous message-based input and output endpoints of the microservice, nor the structure of the business logic data of the architecture. MiSAR has defined mapping rules to recover asynchronous message destinations, named *Queue Listeners*, along with their inbound/outbound *Service Messages* (business data). The asynchronous *Service Dependency* mapping rules are a novel addition that adds value to the MiSAR mapping rule repository. None of the existing approaches in the literature address the asynchronous communication aspect; in an ideal world, communication between microservices should be asynchronous (Westheide, 2016). In addition, the DSL metamodel of MicroART allows for only one classification of infrastructure microservices, while one infrastructure microservice can have multiple infrastructure types – e.g. "consul" is an infrastructure microservice of three types together, service registry and discovery, asynchronous message brokering, and centralised configuration. MiSAR can support multiple-infrastructure patterns by introducing the concept of Infrastructure Pattern Component.

MicroLyze (Kleehaus et al., 2018) is another work which proposes an architecture recovery approach for microservices. MicroLyze, unlike MiSAR, does not adopt a model-driven recovery approach. Instead, it utilises a distributed tracing component that dynamically monitors simulated user requests. In term of service dependency, MicroLyze is similar to MiSAR, where MicroLyze considers intra-relationships among microservices in the application layer. However, MicroLyze does not recover information about the service interface of each microservice and its exposed synchronous endpoints or inbound asynchronous queues. In MiSAR this information corresponds to MiSAR's *Endpoint* and *Queue Listener* concepts. Moreover, MicroLyze does not recover the particular classification of an infrastructure

microservice rather, it considers only two classifications of microservice: functional or infrastructure.

Another study (Mayer and Weinreich, 2018) focuses mainly on the extraction approach of the architecture. Mayer and Weinreich's approach can continuously extract the REST-based architecture from microservice software in which the communication in the services is synchronous and based on HTTP. This work uses dynamic analysis, i.e. monitoring of simulated requests at runtime, to recover synchronous REST-based communications in microservice architecture, while MiSAR currently uses static analysis to recover the same information. Only REST-based and synchronous communication is captured in their work, while MiSAR's approach supports synchronous and asynchronous communication. On the other hand, MiSAR does not recover information about organisational or physical infrastructure aspects of microservice architecture.

Two other studies (Düllmann and van Hoorn, 2017; Rademacher, Sorgalla, et al., 2019) present work related to languages and metamodels for microservice architecture; however, they are not oriented towards architecture recovery. Düllmann and van Hoorn (2017) present the structure of the microservice environment from various viewpoints, such as microservice types, dependency and deployment, focusing on the area of application performance monitoring. The metamodel exhibits several similarities to MiSAR metamodel, First, it comprises explicit concepts for service endpoints and operations, support multiple configuration environment and it covers basic deployment modeling. In contrast, however, their proposed metamodel, unlike MiSAR's metamodel, (a) does not consider asynchronous operation or asynchronous dependencies. (b) the structure of business data offered by the services is also not covered, while in MiSAR, this information corresponds to Service Message. (c) no explicit modeling of infrastructure components e.g., API gateway, registry and discovery, load balancers or circuit breakers.

Rademacher, Sorgalla, et al. (2019) present a metamodel for model-driven development of microservice architecture. Their metamodel is structured into three distinct viewpoints. These comprise only those concepts relevant to domain-specific data, service and microservice architecture operation. Similar to MiSAR metamodel, it

comprises explicit concepts for service endpoint and operations, it covers deployment modeling, it covers the structure of business data and consider asynchronous and asynchronous dependencies. Unlike the MiSAR metamodel, the proposed service and operation metamodels do not define queue or message brokering concepts for asynchronous data exchange.

Rademacher et al. (2019) present a metamodel of a technology modelling language (a PSM) in the context of microservice architecture. Similar to MiSAR, their approach employs model-driven architecture in modelling microservice architecture. However, their approach differs from MiSAR in that it is a top-down method, in which PIMs are transformed into PSMs. The approach exhibits the PSM as more of a generalist point of view, and on the other end there is no concrete distinction between PIM and PSM. The main challenging of this approach is that it is general rather than technology-dependent, and thus introduces a level of complexity when the parser is used to extract the elements of the code that support the general PSM. The ideal scenario would be to have a specific PSM for each programming language.

Compared to the approaches listed in Table 3-5 in Chapter 3, MiSAR is semi-automatic architecture recovery that currently supports the Java language and Spring Framework. The objective of MiSAR is architecture recovery from microservice implementation systems. MiSAR performs static analysis on the source artefacts to recover the architecture model. The metamodel was developed and validated empirically from a set of open-source microservice systems, with the focus on the 'technical' concerns of microservice models, which refers to the building blocks and interconnection that statically describe the characteristic of a software system, and which carry most of the important information about the software systems to be recovered, rather than business process models and non-technical/business-related concerns, like organisation, structuring, culture, etc., which are not considered. Only program-related artefacts are considered, for instance source codes and existing documentation, based on their availability. The main emphasis of MiSAR is the platform-specific model (PSM) and the platform-independent model (PIM) levels of abstraction in modelling microservice architecture (MSA), which have a clearly defined purpose in relation to the model and microservice concepts. The mapping rules

that map between PSM and PIM were implemented using the QVTo transformation language. Manual intervention is required only in artefact collection.

With respect to other approaches, MiSAR includes well-defined concepts that are constructed based on empirical studies. To the best of my knowledge, there is a lack of empirical approaches in MDE and architecture recovery. The MiSAR metamodel is architecturally expressive and includes support for asynchronous message-based service interfaces and asynchronous message-based dependencies, as well as infrastructure microservices. In addition, it has well-developed and thoroughly tested mapping rules, which are essential when developing an automatic model transformations and abstracting hidden architectural complexities.

```
180⊖mapping MicroserviceProject :: MicroserviceProject2Microservice(): Microservice {
181
182     // some code has been removed
183
184     // map dependencies association
185     // synchronous interaction
186     JavaMethod.allInstances()->forEach(_method | _method.ParentProjectName = self.ProjectArtifactId
187                                              and _method.ElementIdentifier = 'exchange'
188                                              and _method.parent.ElementIdentifier = 'RestTemplate'
189                                              and _method.parent.PackageName = 'org.springframework.web.client') {
190         var operation := _method.RootCallingMethod;
191         var provider := '';
192         var destination := '';
193         _method.parameters->forOne(_parameter | _parameter.ParameterOrder = 1){
194             if _parameter.FieldValue <> 'NOT_AVAILABLE' then {
195                 provider := _parameter.FieldValue.substringAfter('//').substringBefore('/');
196                 destination := _parameter.FieldValue.substringAfter(provider) ;
197                 if provider.indexOf(':') <> 0 then {
198                     provider := provider.substringBefore(':');
199                 } endif;
200             } endif;
201         };
202         var http := 'HTTP_REQUEST';
203         _method.parameters->forOne(_parameter | _parameter.ParameterOrder = 2){
204             if _parameter.ElementIdentifier <> 'NOT_AVAILABLE' then {
205                 if _parameter.ElementIdentifier.indexOf('HttpMethod.') <> 0 then {
206                     http := _parameter.ElementIdentifier.substringAfter('HttpMethod.');
207                 } endif;
208             } endif;
209         };
210         destination := 'Endpoint[RequestURI:' + http + ' ' +  destination + ']';
211         dependencies += _method.map JavaMethod2ServiceDependency(operation, provider, destination, false);
212     };
213
217⊖mapping JavaMethod :: JavaMethod2ServiceDependency(operation: String, provider: String, destination: String, wrapped: Boolean): ServiceDependency {
218
219     // if a Dependency with the same details was recovered before, update it, otherwise, create a new one.
220     init{
221         var dependency := ServiceDependency.allInstances()->selectOne(d | d.container().oclAsType(Microservice).MicroserviceName = self.ParentProjectName
222                                                                        and d.ProviderName = provider
223                                                                        and (d.ProviderDestination = destination
224                                                                             or d.ProviderDestination = null));
225         if dependency <> null then {
226             result:= dependency;
227         } endif;
228     }
229
230     ConsumerOperation:= operation;
231     ProviderName:= provider;
232     ProviderDestination:= destination;
233     isWrappedByCircuitBreaker := wrapped;
234     if dependency = null then {
235         Environment := self.ElementProfile;
236     } endif;
237     GeneratingPSM += 'JavaMethod[ElementIdentifier:'+self.ElementIdentifier+'()]';
238
239 }
```

Figure 9-3: Eclipse QVTo implementation of an example mapping rule that recovers synchronous *Service Dependency*

```
180⊖mapping MicroserviceProject :: MicroserviceProject2Microservice(): Microservice {
181
182     // some code has been removed
183
184     // map dependencies association
185     // some code has been removed
186     // asynchronous interaction
186     JavaMethod.allInstances()->forEach(_invoked | _invoked.ElementIdentifier = 'convertAndSend'
187                                                  and ((_invoked.parent.ElementIdentifier = 'AmqpTemplate'
188                                                        and _invoked.parent.PackageName = 'org.springframework.amqp.core')
189                                                        or (_invoked.parent.ElementIdentifier = 'RabbitTemplate'
190                                                            and _invoked.parent.PackageName = 'org.springframework.amqp.rabbit.core'))) {
191
192         var _consumer := Microservice.allInstances()->selectOne( __microservice | __microservice.MicroserviceName = _invoked.ParentProjectName);
193         var _parameterorder := _invoked.parameters->size() - 1;
194         var _routingkey := _invoked.parameters->selectOne(_parameter | _parameter.ParameterOrder = _parameterorder
195                                                               and _parameter.type.ElementIdentifier = 'String').FieldValue.replace('*', '').replace('"', '');
196         var _operation := JavaMethod.allInstances()->selectOne(_method | _method.ParentProjectName = _consumer.MicroserviceName
197                                                                 and _method.ElementIdentifier = _invoked.RootCallingMethod
198                                                                 and _method.annotations->exists(_annotation1 | _annotation1.AnnotationName.endsWith('Mapping')
199                                                                                                         or _annotation1.AnnotationName.endsWith('Listener')));
200         QueueListener.allInstances()->forEach(_queue | _queue.QueueName.indexOf(_routingkey) <> 0 ) {
201             var _provider := _queue.container().oclAsType(ServiceInterface).container().oclAsType(Microservice);
202             _consumer.dependencies += _invoked.map JavaMethod2ServiceDependency(_operation.ElementIdentifier, _provider.MicroserviceName, 'QueueListener[QueueName:' + _queue.QueueName + ']', false);
203
204         };
205     };
206
207     // some code has been removed
208 }
209
210⊖mapping JavaMethod :: JavaMethod2ServiceDependency(operation: String, provider: String, destination: String, wrapped: Boolean): ServiceDependency {
211
212     // if a Dependency with the same details was recovered before, update it, otherwise, create a new one.
213     init{
214         var dependency := ServiceDependency.allInstances()->selectOne(d | d.container().oclAsType(Microservice).MicroserviceName = self.ParentProjectName
215                                                                   and d.ProviderName = provider
216                                                                   and (d.ProviderDestination = destination
217                                                                        or d.ProviderDestination = null));
218         if dependency <> null then {
219             result:= dependency;
220         } endif;
221     }
222
223     ConsumerOperation:= operation;
224     ProviderName:= provider;
225     ProviderDestination:= destination;
226     isWrappedByCircuitBreaker := wrapped;
227     if dependency = null then {
228         Environment := self.ElementProfile;
229     } endif;
230     GeneratingPSM += 'JavaMethod[ElementIdentifier:'+self.ElementIdentifier+'()]';
231
232 }
```
Figure 9-4: Eclipse QVTo implementation of an example mapping rule that recovers asynchronous message-based *Service Dependency*.

9.5. Considerations and Positive Aspects of MiSAR

In this section, the positive aspects of MiSAR are discussed. The discussion will be carried out based on six major arguments, representing the main components of the findings from the studies.

9.1.1. Degree of MiSAR Recovery

Even though MiSAR adopts the static approach to recovery, it can recover the service interaction of an architecture as well as the structure of service data. It has been observed that the Configuration Property and Java Method are the two artefact elements responsible for the success of MiSAR in the recovery of the dynamic aspects of a microservice architecture. This success is realised in practice for TrainTicket case study by the exceptionally high recall score of 96.20% for the recovery of Endpoints, which themselves are recovered from Java Method elements, compared to the overall recall score of 81.20%. To illustrate, with the mapping rules that transform Service Dependency associated to Message Destination (Endpoint/Queue Listener) PIM elements, MiSAR becomes able to automatically resolve the consumer-to-proxy (proxy examples are gateway, discovery, broker, etc.) and proxy-to-provider service interactions into consumer-to-provider service interactions (synchronous/asynchronous). Moreover, the Body Schema attribute of the Service Message PIM element defines the structure of business data exchanged between consumer and provider in the synchronous/asynchronous service interactions.

9.1.2. MiSAR's Reliability

The current MiSAR recovery approach is based on static analysis of source artefacts. The results confirmed that the static methodology of MiSAR is a good alternative to dynamic analysis in many scenarios. In the dynamic recovery approach, e.g. using Zipkin,[33] the application needs to be in a running state so if it fails to run due to expensive resources needed or due to existing bugs then the recovery process will not

[33] https://zipkin.io/.

start at all. Despite that bugs will produce some logical false results, the static method will still work under these scenarios.

In addition, with dynamic recovery which is based on tracing, there is an added effort to design trace requests that can capture the entire behaviour of the application. This effort is eliminated in the static method since it recovers all specifications of the architecture as long as there is a mapping rule matching the source statements implementing the specifications. It can be noticed from Table 8-15, Chapter 8 that MiSAR achieved high correctness in the case study. This adds reliability to MiSAR's static approach. The recall score is also high but lower than precision. This indicates that MiSAR for TrainTicket case study can recover 81.27% of existing architectural elements. This particular figure is limited by artefact terms, implementation and technologies of microservice applications that are currently supported by MiSAR's extendible repository. However, in the future if MiSAR concepts are extended to include performance metrics such as response time dynamic analysis could be needed.

9.1.3. Architectural Expressiveness

MiSAR is architecturally expressive in terms of its metamodel concepts, which are reusable, extendible and comprehensive to the core components in microservice architecture, including asynchronous message-based service interfaces, asynchronous message-based dependencies, service destination, which represents both the provider's Endpoint and the provider's Queue, and Infrastructure pattern components with its categories. For illustration, "ts-auth-service" is one of the successful cases in the recovery process, as presented in Chapter 8. MiSAR was found to recover extra architecture elements compared to the documentation presented by the TrainTicket team. In particular, it provided information about the service operation names as well as the schema of request/response data message(s) associated with every endpoint (see additional elements in Section 8.3.4 in Chapter 8). This extra information recovered is considered an added value to the current documentation of TrainTicket. MiSAR was also able to recover the correct paths for "ts-order-service" endpoints even though they were incorrectly documented. This advantage, when added to the high overall score in precision, indicates that MiSAR has the potential to provide comprehensive and

implemented microservice architecture models. This is because the expressiveness of the metamodel is higher than the informal models drawn by the developers.

9.1.4. Backtracking Support

In order to check the validity of the recovered elements, especially in the case of generating erroneous or undocumented elements, MiSAR includes an attribute named GeneratingPSM to every concept in the PIM metamodel in order to backtrack the PSM source element that generated it, by checking the specific lines in the artefact that generated those particular PSM elements (as discussed in additional elements in Section 8.3.3). This attribute records all PSM elements that are involved in the transformation of one target PIM element. The more PSM elements involved in the transformation, the more certain the existence of a generated PIM element in the architecture is. One exception is the JavaMethod PSM. If a target PIM element is transformed from one JavaMethod element, then its existence in the architecture is 100% certain. This attribute assists in the validation and update stages after the recovery stage.

9.1.5. The Ability of MiSAR to Discover the Existence of Non-JVM Applications

MiSAR was built via analysis of Java applications developed with the Spring Boot/Spring Cloud frameworks, hence a common infrastructure, common technologies and common artefact terms were expected. However, some non-JVM applications were encountered, such as in study 3: TrainTicket developed with Python (ts-voucher-service), Node.js (ts-office-ticket-service), HTML/JS (ts-ui-dashboard) and Go (ts-news-service). MiSAR managed to capture and recover their existence at the container level, namely the elements of Container, Microservice, Service Interface and Dependency from the Docker Compose and/or POM build files. However, it was not able to recover the underlying elements. This indicates the significance of the Docker Compose and POM build artefacts to the static approach of architecture recovery in general as well as to MiSAR in particular.

9.1.6. Model Traceability Support

MiSAR mapping rules are implemented with Eclipse QVTo, which accomplishes model traceability by means of the resolve() function and its variants (see Table 7-1). The resolve() function returns a set of target objects. These objects are the result of an earlier mapping rules from the source object on which the resolving is being applied. These traces can assist the developers in analysing the orders in which mappings rule were invoked.

9.6. Observations and Lessons Learned

The research community can learn several lessons concerning microservice architecture recovery. A number of observation points also emerged from studies presented in this thesis.

1) Among reverse engineering approaches, the MDE approach proved to have great potential for microservice architecture recovery. Models are used to bridge the gap between software implementation and architecture using systematic transformation between the software implementation concepts and the architecture level, as they define the complex systems at multiple abstraction levels through a variety of viewpoints. The suitable abstraction levels recommended for architecture recovery are observed to be the PSM as the source and the PIM as the target.

2) Microservice architecture recovery approaches must be linked to assessing the runtime platform configuration, which influences the runtime behaviour of the service: I learned that an assessing source code alone, e.g. Java source file (low-level artefact), is not sufficient to reverse engineer an architecture which is based on microservices. In order to successfully carry out the process of reverse engineering, different aspects of architectures must be analysed. These aspects include the characteristics of the platform leveraged by the microservice, the platform on which each microservice is deployed, and the interaction of the microservice with other applications external to the architecture being analysed. It is critically important to analyse and understand

the configurations, build and deployment artefacts files (high-level artefacts) that the microservice application uses for development and deployment, since they reflect the abstract and reusable structure of the application, its components and its configurations.

3) The most essential artefacts that participated in the success of a static approach for architecture recovery were the high-level artefacts, including the containerisation (i.e. Docker Compose) and project build (i.e. POM) files. They were observed to be used by a wide span of software development frameworks. As observed from the results in study 3, MiSAR was able to recover all microservices deployed remotely as Docker Hub images, as well as those built locally as non-JVM applications. One Docker Container Definition PSM element is transformed by MiSAR into at least three essential PIM elements: the Microservice, Container and Service Interface elements. In other cases, with the help of hard-coded mapping rules, it can also transform into Infrastructure Microservice, Infrastructure Server Component, Endpoint, Service Dependency and Service Dependency PIM elements.

4) A static approach for architecture recovery can reveal the service interaction (dynamic aspect) of an architecture as well as the structure of service data (business model), which are usually recovered with dynamic approaches. A lot of information on the possible runtime architecture for a distributed system is accessible from its implementation when the mapping rules reflect a high-level understanding and simulating of the required aspects (dynamic and business), and the source model represents the source code in way that simulates the application execution.

   To illustrate, the resolving of the consumer-to-proxy (proxy examples are gateway, discovery, broker, etc.) and proxy-to-provider service interactions into consumer-to-provider service interactions (synchronous/asynchronous) was performed manually in the MicroART approach (Granchelli, Cardarelli, Francesco, et al., 2017), while it was performed easily in other dynamic approaches. In order to accomplish this in MiSAR's static approach, the

mapping rules that transform Service Dependency PIM elements need to reflect understanding of the interaction mechanisms at a higher level than the source code. In practice, Java implementation of AMQP asynchronous interaction transmits the consumer's data message to the broker (rather than the provider) with the routing key attached to the request. Direct transformation of PSM to PIM elements will not accomplish the required resolution. Instead, the MiSAR mapping rule presented in Figure 9-4 simulates the AMQP protocol and hence connects the consumer to a recovered Microservice element with a Queue Listener element matching the routing key. Moreover, the Body Schema attribute of the Service Message PIM element defines the structure of business data exchanged between consumer and provider in the synchronous/asynchronous service interaction.

5) MDE is supported with languages and plugins that aid the semi-automatic generation and manipulation of models. This allows models to be validated against precise metamodels. For example, the implementation of MiSAR's transformation mapping rules with QVTo (MDA transformation language) utilises the Ecore implementation of MiSAR's PSM/PIM metamodels and facilitates the construction of precise, non-error-prone design models. The elements in the source model and their attribute values are always checked prior to the execution of corresponding transformation rules. The automation of the metamodels and model transformations together with the model-driven development principles make it possible to reuse the models involved in reverse engineering processes, and to check and automate the mapping rules, and maintains traceability between codes and models.

9.7. Summary

This chapter has discussed the overall findings that emerged from the studies presented in chapters 5, 6 and 8. I highlighted the main considerations and positive aspects of MiSAR. Based on the discussion of the findings, I then outlined possible lessons to research community be learned concerning microservice architecture recovery.

Chapter 10

Conclusions and Further Work

10.1. Introduction

This thesis reports the development of a novel approach for the recovery of the software architecture of existing microservice software systems, called MiSAR. I present an in-depth empirical investigation into microservice-based systems for defining and evaluation MiSAR. It has been shown that a bottom-up approach using model-driven engineering can be adapted painlessly to reverse engineer microservice-based systems. This chapter summarizes the main ideas presented in this thesis and discusses briefly the research stages to define MiSAR, followed by research contributions. Finally, the limitations of the current study and future research directions in regard to architecture recovery of microservice-based systems are addressed and explained.

10.2. Major Topics Addressed

Chapter 1 presented an overview of software architecture and architecture recovery in the context of microservices, a gap that has been addressed in this research. The chapter highlighted the major research aims and objectives which helped in answering the research question.

Chapter 2 presented a mapping study that looked into available studies on microservice architecture. The outcomes of this analysis assisted in finding the gaps and the prevailing trends in previous research. Microservice architecture recovery was identified as a gap in the microservice field.

Chapter 3 presented useful information and necessary background concerning the field of microservice architecture, model-driven engineering, reverse engineering and software architecture recovery. This chapter also analysed the existing gaps in the available approaches and techniques that are related to the topic.

Chapter 4 discussed the research methodology used to conduct the research. It also discussed the systems selected for the different studies conducted in this thesis.

Chapter 5 presented a description of microservice architecture recovery (MiSAR), an approach that follows a model-driven engineering (MDE) framework. An empirical study was conducted to define MiSAR based on empirical data. The findings in this chapter led to an initial version of the MiSAR artefacts: a metamodel at the platform-independent model (PIM) level and mapping rules.

Chapter 6 presented the second empirical study, which incrementally evaluated and enhanced the initial MiSAR artefacts in order to achieve improved artefacts. This chapter defined the final version of the MiSAR artefacts in order to be able to generate architectural models of implemented microservice systems.

Chapter 7 presented the implementation of the MiSAR artefacts developed with the help of the Eclipse framework, the implementation of the metamodel using Ecore and the implementation of the mapping rules using QVT.

The prime aim of Chapter 8 was to evaluate the usefulness of the MiSAR elements (metamodels and mapping rules) through a case study. In this regard, the TrainTicket system was applied. The efficiency and effectiveness of the MiSAR technique were measured in the case study via the applied criteria. This chapter presents a process for recovering the software architecture of microservice-based systems, which takes platform-specific models (PSMs) concerning the system and obtains PIMs to represent the target model.

Chapter 9 presented a discussion of the studies conducted in the thesis and their results. This chapter also included different considerations regarding the main components of the findings.

10.3. Research Stages

This research involved three stages of exploration and evaluation of the MiSAR artefacts: metamodels, which are microservice architectural elements/concepts that are identified from the source code, and mapping rules between the source code of microservice implementations and the architectural model, which assist the architecture model recovery process. Each stage has methods of collecting and analysing data from microservice systems. Empirical studies were conducted to build and evaluate the approach from empirical data. The following subsections provide a brief overview of the studies conducted in each stage.

Stage One

In this stage, the aim was to identify the concepts and elements needed to build a metamodel and a specific-purpose abstraction of a microservice-based system, and to develop mapping rules that derive a target model from the source model. Two main steps were followed: recovery design (RD) and recovery execution (RE). Eight microservice systems were analysed to identify the architectural concepts, abstractions and concerns that can be recovered. The microservice source systems were then mapped into the metamodel. The outcomes of this stage included initial MiSAR artefacts, the metamodel and the mapping rules, which are artefacts that are used to manually recover architectures of microservice systems. Finally, the metamodel and mapping rules were applied to create architectural models manually (Chapter 5).

Stage Two

The purpose of this stage was a final version of the MiSAR artefacts, including the PIM metamodel, PSM metamodel and mapping rules, which are artefacts that are used to automatically recover architectures of microservice systems. This stage analysed nine microservice systems to validate that the existing artefacts of MiSAR can recover an architectural model. This stage included activities (application to metamodels, application to mapping rules) that helped enhance and refine MiSAR in increments. The mapping rules were then implemented using the QVTo Operational language, while metamodels were implemented as Ecore models using the Eclipse Modeling Framework (EMF). Finally, the implemented MiSAR artefacts (mapping rules and metamodels) were validated by generating a software architecture model of an

unanalysed software system and checking its conformance to the documentation (Chapter 6).

Stage Three

Finally, in the last study, I applied the final version of the MiSAR artefacts via a large-scale microservice system for evaluation purpose, involving a case study in an industry setting, to show the usefulness of the MiSAR elements and evaluate the recovery approach. This study focuses on the integration of all MiSAR artefacts and applies them to obtain architectural models (Chapter 8).

10.4. Summary of Contributions

The main research contributions of this thesis are as follows:

- MiSAR, a novel microservice architecture recovery approach, based on static analysis, to recovering microservice architectural models from existing microservice applications. The approach follows a bottom-up model-driven engineering approach. I have presented an in-depth empirical investigation into microservice-based systems for the purposes of defining and evaluating MiSAR.
- MiSAR metamodels (PSM and PIM): I developed a well-defined concept constructed based on static analysis, reusable, extendible and comprehensive in regard to the core components in a microservice architecture. A systematic approach was used in developing a metamodel. This approach was developed empirically to support comprehension of recovered architectural models. To the best of my knowledge, there are no empirical studies on the evaluation of microservice modelling approaches.
- MiSAR mapping rules: I developed a set of mapping rules that transform microservice implementations into architectural concepts, as a part of the architecture recovery process, which is essential to abstracting the hidden complexities of software architecture. MiSAR's mapping rules allow us to recover a software system's architecture. Mapping rule features are automatic

and implemented using Eclipse Operational QVTo, uni-directional from source to target, and traceable.

- A detailed architecture recovery of a large-scale microservice system has been presented. This contribution is twofold:
  1) It serves to validate the MiSAR architecture recovery process and verify that the recovery process is efficient, effective and applicable to larger software systems.
  2) It enlarges the body of knowledge about the microservice architecture recovery process.

10.5. Threats to Validity

There are several limitations and threats to validity intrinsic to the study and results reported in this thesis. Internal threats of validity concern factors that impact the integrity of the study results. There are three major threats to the internal validity of the empirical study.

- **T1. Absence of Applicable Concepts:** The first one lies in the absence of applicable concepts. Even though the architectural concepts, which are essential and applicable to microservice architecture, are acquired confidently, the research was conducted manually (extracted, compiled and analyzed), which implies that there could be unintentionally a few concepts that are missed out. To mitigate this threat, I performed architecture recovery automatically and compared the recovered model with the documented one.

- **T2. Extraction data from artefacts:** The second one lies in the extraction data from artefacts. As long as the parsing task generating the PSM instance model is automatic, there is a threat that the PSM instance model will either be exposed to errors or will take an extremely long time to complete before the recovery process even starts. The parsing task involves mainly the evaluation of fields and methods' arguments, as well as the tracing of nested methods' invocations, i.e. when a method invokes another method, as well as nested property fields. Such complexity is doubled when the number of artefacts is a

lot and the content is lengthy. To mitigate this threat, I performed manual parsing to validate the results.

- **T3. Bugs in Source Code:** The third one lies in the bugs found in the source code. Even though the static recovery process can still run if there are bugs in the source code, unlike dynamic recovery, the resultant model might represent some incorrect information.

Threats to external validity concerning this study are related to the generalization of results.

- **T4. Recovery scope**: MiSAR, at the current stage, effectively recovers the architecture microservice-based applications if developed in Java and Spring Boot/Spring Cloud framework. The popularity of the Spring Cloud framework in MSA is the reason for supporting it in MiSAR. Furthermore, the recovery approach is limited to static analysis for architecture recovery that produces static information, service interaction as well as the structure of service data. Dynamic analysis and test cases methodologies or expensive static techniques e.g. (data-flow analysis), would derive and extract increased information. Nonetheless, it is still necessary to investigate these costlier methods.

- **T5. Mapping rules scope**: The mapping rules generated were exclusively from applications developed in Spring Framework, the Java language and Spring configuration artefacts (Bootstrap Yaml/Application Yaml). Mapping rules are limited to the version of the framework's and technologies' specifications used to develop the source applications, i.e. libraries, annotations, methods, etc. To mitigate this threat, MiSAR designed as extendible artefacts that might include a new framework version, new technologies and new languages.

  In addition, MiSAR mapping rules were extracted from specific lines in the source artefacts that hold deterministic information/indicators about the output PIM elements. As one architecture element can be declared across multiple artefacts, one PIM element can be generated from multiple mapping rules

having distinct PSM elements. If a particular PIM element is instantiated once, the subsequent mapping rules that generate the same PIM element will update it. However, conflicts in the output of some mapping rules can result from bugs or misleading text existing in the source artefacts before the recovery process. To mitigate this threat, a strategy to find issues in the lines of source artefacts needs to be designed to exclude the possibility of invoking the mapping rules initiating from them.

- **T6. Selecting case studies:** The present empirical studies were based on systems considered to represent basic and best practices. Although the study results have been validated by many case studies, the mapping rules and architectural concepts could be identified by analyzing more projects. Furthermore, empirical reliability, which refers to the consistency of data capture and interpretation, is relevant to this type of study as data extracted and analysed is qualitative and can be interpreted differently. To minimize this, the validation was conducted with systems that have illustrative architecture diagrams and documentation.

- **T7. Evaluation measurements:** The completeness of the MiSAR recovery process is measured mathematically by recall, which, in turn, is determined by the amount of architecture elements extracted from the available application artefacts. Precision, on the other hand, measures the correctness of those extracted architecture elements. As demonstrated in the evaluation Chapter 8, MiSAR achieved results (greater than 80%) for both recall and precision when evaluated against a benchmark case study that meets the selection criteria. A threat to the validity of these measurements is directly related to the test cases on which the calculation is based. When only the documented elements are compared to the extracted elements, the recall score, in particular, is likely to drop, since it will not reflect the undocumented elements existing in the architecture and expressed by the MiSAR metamodel. To mitigate this threat, the test cases included those additional architecture elements that are expressed by MiSAR and expected elements to be recovered.

- **T8. Scalability to larger systems:** It has been observed by empirical study and evaluation of cases of various size that the scalability of a microservice-based application could be a threat to the validity of MiSAR's recovery process if new technologies/implementation languages are encountered that are not supported by the metamodel and/or the mapping rules. Otherwise, MiSAR's performance in terms of parsing time and recovery time will increase as the microservice-based application scales up. MiSAR's performance and efficiency were evaluated through the time spent on the recovery as well as scalability to larger systems. While the threat to performance is unavoidable, the former threat is mitigated by user intervention in enhancing the output model, as well as updates to MiSAR artefacts by extending the metamodel and mapping rules to support the new technologies/implementation languages.

- **T9. Replication of MiSAR**: In principle, MiSAR recovery process is based on rules that perform a direct mapping from source PSM model to target PIM model. A generated model of other systems might introduce some limitations. MiSAR's effectiveness, precision and recall values will vary from one system to another, although in train ticket evaluation the value of precision was higher than recall. This means that MiSAR retrieves a great number of architectural elements that are correct, but a few of them could be erroneous. In addition, a small set of activities could not be retrieved. The main threat to MiSAR's validity is the implementation technology and implementation logic that are not supported in the current repository of mapping rules and metamodels. This indicates that MiSAR may be limited by the artefact terms, implementation languages and technologies of the microservice applications that are currently supported by MiSAR's extendible repository. To mitigate this threat, the MiSAR repository (PIM/PSM metamodels and mapping rules) is must continuous updating with the new-found elements. In this sense, the recovery process is said to be deterministic rather than stochastic. In the case of new recovery cases, the precision results will not encounter major change while the recall results of the process may drop if the specifications of the new application are not supported yet by MiSAR artefacts (rules and metamodels).

10.6. Future Directions

As already discussed, the need to adequately understand existing software architectures is evident. Software architecture recovery needs to be regarded as a proactive approach, as it is always undertaken for prospective progression of the system in question, and is the point of initiation for engineering activities. This research is an initial step in the research field of software re-engineering, with the focus on resolving the issues associated with the early stages of the software lifecycle, including validation, specification, system knowledge, maintenance and impact analysis. The MiSAR approach is still ongoing. The following points present certain areas for future research that could broaden the work covered in this thesis:

- **MiSAR tool:** In order to decrease the requirement for manual support to architecture recovery, a set of efficacious tools and tactics capable of supporting the process needs to be developed. However, this is an extensive question to answer in this field, and beyond the scope of this thesis. The mapping rules automated through the QVT language and the metamodels were developed with effective usage of Ecore. Thus, architectural recovery requires tools for the purpose of integrating all MiSAR artefacts (metamodels, mapping rules). A future intention is the utilising MiSAR tool in an industrial setting and obtains feedback from practitioners in the usefulness of the architectural models recovered and the user-friendliness of the tool. For example, that would help to acknowledge challenges related to user-friendliness and/or understandability of the MiSAR concepts to microservice practitioners.

- **Enlarging the coverage of MiSAR:** Currently, MiSAR architecture recovery provides a specific solution for Java software and spring OSS framework. The intention is to enhance MiSAR's coverage of artefacts by considering different frameworks and languages beyond Java. Along with this, the aim is to extend MiSAR to support components which are positioned in deployment platforms and the public cloud (e.g. Vagrant).

- **A graphical modelling tool for MiSAR:** Currently, the graphical process of MISAR architectural models still encompasses mainly manual work, which involves a repetitive process, and is labour intensive and time-consuming. A future intention is to develop a graphical tool for the MiSAR graphical model diagram that has a dedicated parser which takes the Ecore PIM as input and performs both parsing and graphical model diagram generation tasks. The former is in charge of extracting information from the PIM (in XML format), while the latter generates the graphical diagram/documentation. A future intention is to investigate more about the graphical notations that best fit to properly represent the microservice architecture model.

- **Microservice architectural inconsistency tool:** The findings drawn from architecture recovery are used prominently for identifying inconsistencies between two aspects, microservice-based systems and architectural models. In this respect, the study (Buckley et al., 2015) claimed that consistency checking and combining architecture recovery are becoming a crucial approach to identifying architecture erosion/drift. A future intention is to develop a tool that allowing software engineers to recover and check inconsistency of microservice-based systems and their planned architecture.

- Future studies should compare architecture recovery based on static analysis and architecture recovery based on dynamic analysis.

- Future research should consider what the effect of architecture recovery is in identifying microservice ‘bad smells’. A further extension would be to add detection strategies to automatically detect microservice smells such as shared persistence, hard-coded endpoints and cyclic dependency, in projects developed with microservices.

References

Abi-Antoun, M. and Aldrich, J. (2008), "A field study in static extraction of runtime architectures", *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ACM Press, New York, New York, USA, pp. 22–28.

Agrawal, A. and Aditya. (2003), "Metamodel based model transformation language to facilitate domain specific model driven architecture", *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA '03*, ACM Press, New York, New York, USA, p. 118.

Akehurst, D. and Kent, S. (2002), "A Relational Approach to Defining Transformations in a Metamodel", Springer, Berlin, Heidelberg, pp. 243–258.

Akkiraju, R., Mitra, T. and Thulasiram, U. (2012), "Reverse engineering platform independent models from business software applications.", *In Reverse Engineering-Recent Advances and Applications. InTech,* p. 64.

Ali, N., Baker, S., O'Crowley, R., Herold, S. and Buckley, J. (2018), "Architecture consistency: State of the practice, challenges and requirements", *Empirical Software Engineering*, Springer New York LLC, Vol. 23 No. 1, pp. 224–258.

Ali, N., Rosik, J. and Buckley, J. (2012), "Characterizing real-time reflexion-based architecture recovery: an in-vivo multi-case study", *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*, pp. 23–32.

Alpers, S., Becker, C., Oberweis, A. and Schuster, T. (2015), "Microservice based tool support for business process modelling", *Proceedings of the 2015 IEEE 19th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, EDOCW 2015*, Institute of Electrical and Electronics Engineers Inc., pp. 71–78.

Alshuqayran, N., Ali, N. and Evans, R. (2016), "A Systematic Mapping Study in Microservice Architecture", *IEEE International Conference on Service-Oriented Computing and Applications*, not yet published.

Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M. and Steinder, M. (2016), "Performance evaluation of microservices architectures using containers", *Proceedings - 2015 IEEE 14th International Symposium on Network Computing and Applications, NCA 2015*, Institute of Electrical and Electronics Engineers Inc., pp. 27–34.

Ameller, D., Burgués, X., Collell, O., Costal, D., Franch, X. and Papazoglou, M.P. (2015), "Development of service-oriented architectures using model-driven development: A mapping study", *Information and Software Technology*, Elsevier B.V., Vol. 62 No. 1, pp. 42–66.

AndroMDA.org. (2003), "AndroMDA Model Driven Architecture Framework", available at: https://www.andromda.org/ (accessed 25 April 2019).

Arnold, R.S. (1993), *Software Reengineering*, IEEE Computer Society Press.

Bak, P., Melamed, R., Moshkovich, D., Nardi, Y., Ship, H. and Yaeli, A. (2015), "Location and Context-Based Microservices for Mobile and Internet of Things Workloads", *Proceedings - 2015 IEEE 3rd International Conference on Mobile Services, MS 2015*, Institute of Electrical and Electronics Engineers Inc., pp. 1–8.

Barendrecht, P.J. (2010), *Modeling Transformations Using QVT Operational*

Mappings, available at: http://redpanda.nl/BEP_P.J.Barendrecht.pdf (accessed 11 March 2019).
Bass, L., Clements, P. and Kazman, R. (2003), *Software Architecture in Practice*, *Vasa*, Vol. 2nd, Addison-Wesley Professional, available at:https://doi.org/10.1024/0301-1526.32.1.54.
El Beggar, O., Bousetta, B. and Gadi, T. (2013), “Comparative Study between Clustering and Model Driven Reverse Engineering Approaches”, *Lecture Notes on Software Engineering, 1(2), p.135.*, Vol. 1 No. 2, pp. 135–140.
Benoit, C., Robert, F., Jean-Marc, J., Bernhard, R., James, S. and Didier, V. (2016), *Engineering Modeling Languages: Turning Domain Knowledge into Tools*, CRC Press.
Bex, G.J., Maneth, S. and Neven, F. (2002), “A formal model for an expressive fragment of XSLT”, *Information Systems*, Vol. 27 No. 1, pp. 21–39.
Bézivin, J. (2005), “On the Unification Power of Models”, *Software & Systems Modeling*, pp. 1–31.
Bezivin, J. and Gerbe, O. (2001), “Towards a precise definition of the OMG/MDA framework”, *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pp. 273–280.
Bibi, M. and Maqbool, O. (2011), “Version information support for software architecture recovery”, *2011 7th International Conference on Emerging Technologies*, pp. 1–6.
Biehl, M. (2010), “Literature study on model transformations”, *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*, No. July, pp. 1–28.
Brambilla, M., Cabot, J. and Wimmer, M. (2017), *Model-Driven Software Engineering in Practice: Second Edition*, *Synthesis Lectures on Software Engineering*, Vol. 3, Morgan & Claypool Publishers LLC, available at:https://doi.org/10.2200/s00751ed2v01y201701swe004.
Brereton, P., Kitchenham, B., Budgen, D. and Li, Z.H. (2008), “Using a Protocol Template for Case Study Planning”, BCS Learning & Development, available at:https://doi.org/10.14236/EWIC/EASE2008.5.
Brown, S. (2018), “The C4 model for visualising software architecture”, available at: https://c4model.com/ (accessed 22 July 2020).
Brunelière, H., Cabot, J., Dupé, G. and Madiot, F. (2014), “MoDisco: A model driven reverse engineering framework”, *Information and Software Technology*, Vol. 56 No. 8, pp. 1012–1032.
Buckley, J., Ali, N., English, M., Rosik, J. and Herold, S. (2015), “Real-Time Reflexion Modelling in architecture reconciliation: A multi case study”, *Information and Software Technology*, Elsevier, Vol. 61, pp. 107–123.
Buckley, J., Mooney, S., Rosik, J. and Ali, N. (2013), “JITTAC: A Just-in-Time tool for architectural consistency”, *Proceedings - International Conference on Software Engineering*, pp. 1291–1294.
Budgen, D., Turner, M., Brereton, P. and Kitchenham, B. (2008), “Using Mapping Studies in Software Engineering”, *PPIG*, Vol. 8 No. 2, pp. 195–204.
Burson, S., Kotik, G.B. and Markosian, L.Z. (1990), “A program transformation approach to automating software re-engineering”, *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International*, pp. 314–322.
Cerny, T., Donahoo, M.J. and Trnka, M. (2018), “Contextual understanding of microservice architecture”, *ACM SIGAPP Applied Computing Review*, Association for Computing Machinery (ACM), Vol. 17 No. 4, pp. 29–45.

Chikofsky, E.J. and Cross, J.H. (1990), “Reverse engineering and design recovery: A taxonomy. IEEE software”, No. January, pp. 13–17.
Ciuffoletti, A. (2015), “Automated Deployment of a Microservice-based Monitoring Infrastructure”, *Procedia Computer Science*, Elsevier B.V., Vol. 68, pp. 163–172.
Cosentino, V., Cabot, J., Albert, P., Bauquel, P. and Perronnet, J. (2012), “A model driven reverse engineering framework for extracting business rules out of a java application”, *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 7438 LNCS, pp. 17–31.
Czarnecki, K. and Helsen, S. (2003), “"Classification of model transformation approaches”, *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Vol. 45, pp. 1–17.
Czarnecki, K. and Helsen, S. (2006), “Feature-based survey of model transformation approaches”, *IBM Systems Journal*, IBM Corporation, Vol. 45 No. 3, pp. 621–645.
Dave, S., Budinsky, F., Paternostro, M. and Merks, E. (2008), *EMF: Eclipse Modeling Framework*, Addison-Wesley Professional., available at: https://www.eclipse.org/modeling/emf/ (accessed 5 December 2019).
Daya, S., Duy, N. Van, Eati, K., Ferreira, C.M., Glozic, D., Gucer, V., Gupta, M., et al. (2015), “Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach”, August.
van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L. and Riva, C. (2004), “Symphony: view-driven software architecture reconstruction”, *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, IEEE Comput. Soc, pp. 122–132.
Devanbu, P.T. (1992), “GENOA - A Customizable, Language- And Front-end Independent Code Analyzer”, *International Conference on Software Engineering*, pp. 307–317.
Ding, L. and Medvidovic, N. (2001), “Focus: A light-weight, incremental approach to software architecture recovery and evolution”, *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference On*, pp. 191–200.
Dirckze, R. (2002), *Java Metadata Interface (JMI) Specification*, *Java Community Process*.
Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L. (2017), “Microservices: Yesterday, today, and tomorrow”, *Present and Ulterior Software Engineering*, pp. 195–216.
Ducasse, S. and Pollet, D. (2009), “Software architecture reconstruction: A process-oriented taxonomy”, *IEEE Transactions on Software Engineering*, Vol. 35 No. 4, pp. 573–591.
Düllmann, T.F. and van Hoorn, A. (2017), “Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches”, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion - ICPE ’17 Companion*, ACM Press, New York, New York, USA, pp. 171–172.
Eberhard, W. (2016), *Microservices: Flexible Software Architecture*, 1st ed., Boston: Addison-Wesley Professional.
Esposito, C., Castiglione, A. and Choo, K.K.R. (2016), “Challenges in Delivering Software in the Cloud as Microservices”, *IEEE Cloud Computing*, Institute of Electrical and Electronics Engineers Inc., Vol. 3 No. 5, pp. 10–14.

Fakeeh, R. and Alshuqayran, N. (2019), "Misar Parser", available at: https://github.com/RanaFakeeh-87/MisarParser (accessed 27 July 2020).
Falleri, J.-R., Huchard, M. and Nebut, C. (2006), *Towards a Traceability Framework for Model Transformations in Kermeta*.
Feijs, L., Krikhaar, R. and Ommering, R. Van. (1998), "A relational approach to support software architecture analysis", *Software: Practice and Experience*, John Wiley & Sons, Ltd, Vol. 28 No. 4, pp. 371–400.
Fernandez, A., Insfran, E. and Abrahão, S. (2011), "Usability evaluation methods for the web: A systematic mapping study", *Information and Software Technology*, Elsevier B.V., Vol. 53 No. 8, pp. 789–817.
Finnigan, P.J., Holt, R.C., Kalas, I., Kerr, S., Kontogiannis, K., Muller, H.A., Mylopoulos, J., et al. (1997), "The software bookshelf", *IBM Systems Journal*, Vol. 36 No. 4, pp. 564–593.
Fleurey, F., Breton, E., Baudry, B., Nicolas, A. and Jézéquel, J.-M. (2007), "Model-Driven Engineering for Software Migration in a Large Industrial Context", *Lecture Notes in Computer Science*, Vol. 4735, pp. 482–497.
Di Francesco, P., Malavolta, I. and Lago, P. (2017), "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption", *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017*, IEEE, pp. 21–30.
François, A., Richard, R. and John, H. (2015), "Migrating Healthcare Applications to the Cloud through Containerization and Service Brokering", *In HEALTHINF (Pp. 164-171).*
Gadea, C., Trifan, M., Ionescu, D. and Ionescu, B. (2016), "A reference architecture for real-time microservice API consumption", *3rd Workshop on CrossCloud Infrastructures and Platforms, CrossCloud 2016 - Colocated with EuroSys 2016*, Association for Computing Machinery, Inc, New York, New York, USA, pp. 1–6.
Gall, H., Jazayeri, M., Klösch, R., Lugmayr, W. and Trausmuth, G. (1996), "Architecture recovery in ARES", *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints' 96) on SIGSOFT'96 Workshops*, pp. 111–115.
Garlan, D. (2000), "Software architecture: a roadmap", *Proceedings of the Conference on the Future of Software Engineering*, pp. 91–101.
Gaševic, D., Djuric, D. and Devedžic, V. (2009), "Model Driven Engineering", *Model Driven Engineering and Ontology Development*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 125–155.
Granchelli, G., Cardarelli, M., Francesco, P. Di, Malavolta, I., Iovino, L. and Salle, A. Di. (2017), "Towards recovering the software architecture of microservice-based systems", *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, No. November, pp. 46–53.
Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L. and Di Salle, A. (2017), "MicroART: A software architecture recovery tool for maintaining microservice-based systems", *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*.
Gruman, G. and Morrison, A. (2014), "Microservices : The resurgence of SOA principles and an alternative to the monolith", *PwC Technology Forecast*, No.

1, pp. 1–8.
Guo, D., Wang, W., Zeng, G. and Wei, Z. (2016), "Microservices architecture based cloudware deployment platform for service computing", *Proceedings - 2016 IEEE Symposium on Service-Oriented System Engineering, SOSE 2016*, Institute of Electrical and Electronics Engineers Inc., pp. 358–364.
Guo, G.Y., Atlee, J.M. and Kazman, R. (1999), "A Software Architecture Reconstruction Method", Springer, Boston, MA, pp. 15–33.
Hassan, S., Ali, N. and Bahsoon, R. (2017), "Microservice Ambients: An Architectural Meta-Modelling Approach for Microservice Granularity", *2017 IEEE International Conference on Software Architecture (ICSA)*, IEEE, pp. 1–10.
Hassan, S. and Bahsoon, R. (2016), "Microservices and their design trade-offs: A self-adaptive roadmap", *Proceedings - 2016 IEEE International Conference on Services Computing, SCC 2016*, Institute of Electrical and Electronics Engineers Inc., pp. 813–818.
Hassan, S., Bahsoon, R. and Kazman, R. (2019), "Microservice Transition and its Granularity Problem: A Systematic Mapping Study", available at: http://arxiv.org/abs/1903.11665 (accessed 15 April 2020).
Hevner, A.R. (2007), "A Three Cycle View of Design Science Research", *Scandinavian Journal of Information Systems*, Vol. 19 No. 2, pp. 87–92.
Hevner, March, Park and Ram. (2004), "Design Science in Information Systems Research", *MIS Quarterly*, Vol. 28 No. 1, p. 75.
Ibryam, B. (2016), "Spring Cloud for Microservices Compared to Kubernetes - RHD Blog", *Redhat*, available at: https://developers.redhat.com/blog/2016/12/09/spring-cloud-for-microservices-compared-to-kubernetes/ (accessed 12 February 2019).
Jambunathan, B. and Kalpana, Y. (2016), "Multi cloud deployment with containers", *International Journal of Engineering and Technology*, Vol. 8 No. 1, pp. 421–428.
Jilani, A.A.A., Aftab, A., Jilani, A., Usman, M. and Halim, Z. (2010), "Model Transformations in Model Driven Architecture", *Universal Journal of Computer Science and Engineering Technology*, Vol. 1 No. 1, pp. 50–54.
K. Petersen, R. Feldt, S.M. (2008), "Systematic mapping studies in software engineering", *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, No. February 2015, pp. 1-10.
Kang, H., Le, M. and Tao, S. (2016), "Container and microservice driven design for cloud infrastructure DevOps", *Proceedings - 2016 IEEE International Conference on Cloud Engineering, IC2E 2016: Co-Located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016*, Institute of Electrical and Electronics Engineers Inc., pp. 202–211.
Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H. and Carriere, J. (1998), "The architecture tradeoff analysis method", *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193)*, IEEE Comput. Soc, pp. 68–78.
Kazman, R., O'Brien, L. and Verhoef, C. (2002), *Architecture Reconstruction Guidelines*.
Kent, S. (2002), "Model Driven Engineering", *International Conference on Integrated Formal Methods*, pp. 286–298.
Killalea, T. (2016), "The hidden dividends of microservices", *Communications of the ACM*, Association for Computing Machinery, Vol. 59 No. 8, pp. 42–45.

Kitchenham, B. and Charters, S. (2007), *Guidelines for Performing Systematic Literature Reviews in Software Engineering*, *Technical Report*, Vol. 2.

Kleehaus, M., Uludağ, Ö., Schäfer, P. and Matthes, F. (2018), "MICROLYZE: A framework for recovering the software architecture in microservice-based environments", *Lecture Notes in Business Information Processing*, Vol. 317, pp. 148–162.

Kleppe, A.G., Warmer, J.B. and Bast, W. (2003), *MDA Explained : The Model Driven Architecture : Practice and Promise*, Addison-Wesley.

Knoche, H. (2016), "Sustaining runtime performance while incrementally modernizing transactional monolithic software towards microservices", *ICPE 2016 - Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering*, Association for Computing Machinery, Inc, New York, New York, USA, pp. 121–124.

Koskimies, K., Systä, T., Tuomi, J. and Männistö, T. (1998), "Automated support for modeling OO software", *IEEE Software*, Vol. 15 No. 1, pp. 87–94.

Kreger, H. and Estefan, J. (2009), *Navigating the SOA Open Standards Landscape Around Architecture*, available at: www.opengroup.org (accessed 23 April 2020).

Krikhaar, R.L. (1997), "Reverse architecting approach for complex systems", *Proceedings International Conference on Software Maintenance*, pp. 4–11.

Krylovskiy, A., Jahn, M. and Patti, E. (2015), "Designing a Smart City Internet of Things Platform with Microservice Architecture", *Proceedings - 2015 International Conference on Future Internet of Things and Cloud, FiCloud 2015 and 2015 International Conference on Open and Big Data, OBD 2015*, Institute of Electrical and Electronics Engineers Inc., pp. 25–30.

Le, V.D., Neff, M.M., Stewart, R. V., Kelley, R., Fritzinger, E., Dascalu, S.M. and Harris, F.C. (2015), "Microservice-based architecture for the NRDC", *Proceeding - 2015 IEEE International Conference on Industrial Informatics, INDIN 2015*, Institute of Electrical and Electronics Engineers Inc., pp. 1659–1664.

Lewis, J. and Fowler, M. (2014), "Microservices", available at: http://martinfowler.com/articles/microservices.html (accessed 10 August 2016).

Liu, D., Zhu, H., Xu, C., Bayley, I., Lightfoot, D., Green, M. and Marshall, P. (2016), "CIDE: An Integrated Development Environment for Microservices", *2016 IEEE International Conference on Services Computing (SCC)*, IEEE, pp. 808–812.

LONG, J. (2016), "This Year in Spring - 2016 edition", available at: https://spring.io/blog/2016/12/28/this-year-in-spring-2016-edition (accessed 12 February 2019).

Mackenzie, C.M., Mccabe, F., Brown, P.F., Net, P., Metz, R. and Hamilton, A. (2006), *Reference Model for Service Oriented Architecture 1.0*, available at: http://docs.oasis-open.org/soa-rm/v1.0/ (accessed 23 April 2020).

Malavalli, D. and Sathappan, S. (2015), "Scalable microservice based architecture for enabling DMTF profiles", *Proceedings of the 11th International Conference on Network and Service Management, CNSM 2015*, Institute of Electrical and Electronics Engineers Inc., pp. 428–432.

Maqbool, O. and Babri, H. (2007a), "Hierarchical clustering for software architecture recovery", *IEEE Transactions on Software Engineering*, IEEE, Vol. 33 No. 11, pp. 759–780.

Maqbool, O. and Babri, H.A. (2007b), "Bayesian Learning for Software Architecture

Recovery", *2007 International Conference on Electrical Engineering*, pp. 1–6.
Marru, S., Pamidighantam, S., Pierce, M. and Wimalasena, C. (2015), "Apache airavata as a laboratory: Architecture and case study for component-based gateway middleware", *SCREAM 2015 - Proceedings of the 2015 Workshop on the Science of Cyberinfrastructure: Research, Experience, Applications and Models, Part of HPDC 2015*, Association for Computing Machinery, Inc, New York, New York, USA, pp. 19–26.
Mayer, B. and Weinreich, R. (2018), "An Approach to Extract the Architecture of Microservice-Based Software Systems", *Proceedings - 12th IEEE International Symposium on Service-Oriented System Engineering, SOSE 2018 and 9th International Workshop on Joint Cloud Computing, JCC 2018*, Institute of Electrical and Electronics Engineers Inc., pp. 21–30.
Meinke, K. and Nycander, P. (2015), "Learning-based testing of distributed microservice architectures: Correctness and fault injection", *In SEFM 2015 Collocated Workshops*, Vol. 9509, Springer Verlag, pp. 3–10.
Mendonça, N.C. and Kramer, J. (1996), "Requirements for an effective architecture recovery framework", *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development*, pp. 101–105.
Mendonça, N.C. and Kramer, J. (2001), "An approach for recovering distributed system architectures", *Automated Software Engineering*, Springer, Vol. 8 No. 3–4, pp. 311–354.
Mens, T. and Van Gorp, P. (2006), "A taxonomy of model transformation", *Electronic Notes in Theoretical Computer Science*, Vol. 152 No. 1–2, pp. 125–142.
Mens, T., Magee, J. and Rumpe, B. (2010), "Evolving Software Architecture Descriptions of Critical Systems Distributed software engineering View project SECO-ASSIST View project", available at:https://doi.org/10.1109/MC.2010.136.
Miller, J., Mukerji, J. and Belaunde France, M. (2003), *MDA Guide Version 1.0.1*, available at: https://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf (accessed 29 January 2019).
Mitchell, B.S. and Mancoridis, S. (2006), "On the automatic modularization of software systems using the bunch tool", *IEEE Transactions on Software Engineering*, Vol. 32 No. 3, pp. 193–208.
Müller, H.A., Tilley, S.R. and Wong, K. (1993), "Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project", *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, pp. 217–226.
Murphy, G.C., Notkin, D. and Sullivan, K. (1995), "Software reflexion models", *ACM SIGSOFT Software Engineering Notes*, Vol. 20 No. 4, pp. 18–28.
Murphy, G.C., Notkin, D. and Sullivan, K.J. (2001), "Software reflexion models: Bridging the gap between design and implementation", *IEEE Transactions on Software Engineering*, Vol. 27 No. 4, pp. 364–380.
Myers, M.D. (1997), "Qualitative research in information systems", *Management Information Systems Quarterly*, Vol. 21 No. June, pp. 1–18.
Namiot, D. and Sneps-sneppe, M. (2014), "On Micro-services Architecture", Vol. 2 No. 9, pp. 24–27.
Newman, S. (2015), *Building Microservices: Designing Fine Grained System*,

O'Reilly Media, Inc.
Newman, S. (2019), *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*.
Nicolaescu, P., Toubekis, G. and Klamma, R. (2015), "A microservice approach for near real-time collaborative 3D objects annotation on the web", *In International Conference on Web-Based Learning*, Vol. 9412, Springer Verlag, pp. 187–196.
Nycander, P. (2015), "Learning-Based Testing of Microservices: An Exploratory Case Study Using LBTest", available at: http://www.diva-portal.org/smash/record.jsf?pid=diva2:847215.
O'Brien, L., Smith, D. and Lewis, G. (2005), "Supporting Migration to Services using Software Architecture Reconstruction", *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*, IEEE, pp. 81–91.
OMG. (2014), *MDA Guide Rev.2.0*, available at: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.
OMG. (2016), "MOF Query/View/Transformation", available at: https://www.omg.org/spec/QVT/About-QVT/ (accessed 16 January 2020).
Pahl, C. and Jamshidi, P. (2015), "Microservices : a Systematic Mapping Study", No. January, available at:https://doi.org/10.5220/0005785501370146.
Pahl, C., Jamshidi, P. and Zimmermann, O. (2018), "Architectural Principles for Cloud Software", *ACM Transactions on Internet Technology*, ACM, Vol. 18 No. 2, pp. 1–23.
Paige, R. (2006), "The Meta-Object Facility (MOF)", available at: https://slideplayer.com/slide/5922445/ (accessed 4 March 2019).
Pashov, I. and Riebisch, M. (2004), "Using feature modeling for program comprehension and software architecture recovery", *Proceedings - 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, ECBS 2004*, pp. 406–417.
Patanjali, S., Truninger, B., Harsh, P. and Bohnert, T.M. (2015), "CYCLOPS: A micro service based approach for dynamic rating, charging & billing for cloud", *Proceedings of the 13th International Conference on Telecommunications, ConTEL 2015*, Institute of Electrical and Electronics Engineers Inc., available at:https://doi.org/10.1109/ConTEL.2015.7231226.
Pérez-Castillo, R., Fernández-Ropero, M., Guzmán, I.G.R. De and Piattini, M. (2011), "MARBLE. A business process archeology tool", *IEEE International Conference on Software Maintenance, ICSM*, pp. 578–581.
Pérez-Castillo, R., De Guzmán, I.G.R. and Piattini, M. (2011), "Business process archeology using MARBLE", *Information and Software Technology*, Vol. 53 No. 10, pp. 1023–1044.
Pires, L.., Hammoudi, S. and Selic, B. eds. (2018), *Model-Driven Engineering and Software Development: 5th International* ., MODELSWARD 2017, Porto, Portugal, February 19-21, 2017, Revised Selected Papers (Vol. 880). Springer.
Popma, R. (2004), "Introduction to JET", available at: https://www.eclipse.org/articles/Article-JET/jet_tutorial1.html (accessed 25 April 2019).
Quilici, A. and Chin, D.N. (1995), "DECODE: a cooperative environment for reverse-engineering legacy software", *Proceedings of 2nd Working Conference on Reverse Engineering*, pp. 156–165.
Rademacher, F., Sachweh, S. and Zundorf, A. (2017), "Differences between model-driven development of service-oriented and microservice architecture",

Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings, pp. 38–45.
Rademacher, F., Sachweh, S. and Zundorf, A. (2019), “Aspect-oriented modeling of technology heterogeneity in microservice architecture”, *Proceedings - 2019 IEEE International Conference on Software Architecture, ICSA 2019*, Institute of Electrical and Electronics Engineers Inc., pp. 21–30.
Rademacher, F., Sorgalla, J., Sachweh, S. and Zundorf, A. (2019), “Viewpoint-specific model-driven microservice development with interlinked modeling languages”, *Proceedings - 13th IEEE International Conference on Service-Oriented System Engineering, SOSE 2019, 10th International Workshop on Joint Cloud Computing, JCC 2019 and 2019 IEEE International Workshop on Cloud Computing in Robotic Systems, CCRS 2019*, Institute of Electrical and Electronics Engineers Inc., pp. 57–66.
Rademacher, F., Sorgalla, J., Sachweh, S. and Zündorf, A. (2018), “Towards a Viewpoint-specific Metamodel for Model-driven Development of Microservice Architecture”, *ArXiv Preprint ArXiv:1804.09948.*, available at: http://arxiv.org/abs/1804.09948 (accessed 1 November 2019).
Rahman, M. and Gao, J. (2015a), “A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development”, *2015 IEEE Symposium on Service-Oriented System Engineering*, pp. 321–325.
Rahman, M. and Gao, J. (2015b), “A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development”, *Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium On*, pp. 321–325.
Raibulet, C., Arcelli Fontana, F. and Zanoni, M. (2017), “Model-driven reverse engineering approaches: A systematic literature review”, *IEEE Access*, Institute of Electrical and Electronics Engineers Inc., 29 July.
Richards, M. (2015), *Microservices vs. Service-Oriented Architecture*, edited by Nan Barber, R.R., First Edition., O’Reilly Media, available at: http://174.129.160.22/WilliamDengTechRef/Microservices_vs_SOA_NGINX.pdf (accessed 29 January 2019).
Richardson, C. (2018), “Microservice Architecture pattern”, available at: https://microservices.io/patterns/microservices.html (accessed 12 February 2019).
Riva, C. (2000), “Reverse architecting: an industrial experience report”, *Proceedings Seventh Working Conference on Reverse Engineering*, pp. 42–50.
Rory, O., Peter, E. and Paul, M.C. (2016), “Exploring the Impact of Situational Context — A Case Study of a Software Development Process for a Microservices Architecture - IEEE Conference Publication”, *IEEE/ACM International Conference on Software and System Processes (ICSSP).*
Di Ruscio, D., Eramo, R. and Pierantonio, A. (2012), “Model Transformations”, *In International School on Formal Methods for the Design of Computer, Communication and Software Systems (Pp. 91-136). Springer, Berlin, Heidelberg.*
Safina, L., Mazzara, M., Montesi, F. and Rivera, V. (2016), “Data-Driven Workflows for Microservices: Genericity in Jolie”, *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, IEEE, pp. 430–437.
Sartipi, K. (2003), “Software architecture recovery based on pattern matching”, *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference On*, pp. 293–296.

Sartipi, K., Lingdong, Y. and Safyallah, H. (2006), "Alborz: An interactive toolkit to extract static and dynamic views of a software system", *IEEE International Conference on Program Comprehension*, Vol. 2006, pp. 256–259.

Savchenko, D. and Radchenko, G. (2015), "Microservices validation: Methodology and implementation", *CEUR Workshop Proceedings*, Vol. 1513, pp. 21–28.

Savchenko, D.I., Radchenko, G.I. and Taipale, O. (2015a), "Microservices validation: Mjolnirr platform case study", *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015 - Proceedings*, pp. 235–240.

Savchenko, D.I., Radchenko, G.I. and Taipale, O. (2015b), "Microservices validation: Mjolnirr platform case study", *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention On*, pp. 235–240.

Schmidt, D.C. (2006), *Model-Driven Engineering*, *IEEE Computer*, Vol. 39, available at: http://www.computer.org/portal/site/computer/menuitem.e533b16739f5... (accessed 11 March 2019).

Schwanke, R.W. (1991), "An intelligent tool for re-engineering software modularity", *Software Engineering, 1991. Proceedings., 13th International Conference On*, pp. 83–92.

Selic, B. (2003), "The pragmatics of model-driven development", *IEEE Software*, Vol. 20 No. 5, pp. 19–25.

Sorgalla, J., Rademacher, F., Sachweh, S. and Zündorf, A. (2018), "On collaborative model-driven development of microservices", *In Federation of International Conferences on Software Technologies: Applications and Foundations* , Vol. 11176 LNCS, Springer Verlag, pp. 596–603.

SourceForge. (2003), "JAMDA Java Model Driven Architecture", available at: https://sourceforge.net/projects/jamda/ (accessed 25 April 2019).

Steinberg, D., Budinsky, F., Merks, E. and Paternostro, M. (2008), *EMF: Eclipse Modeling Framework*, Person Education, 2008.

Stubbs, J., Moreira, W. and Dooley, R. (2015), "Distributed Systems of Microservices Using Docker and Serfnode", *Proceedings - 7th International Workshop on Science Gateways, IWSG 2015*, Institute of Electrical and Electronics Engineers Inc., pp. 34–39.

Systä, T. (2000), *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*, available at: http://acta.uta.fi (accessed 6 May 2020).

Thones, J. (2015), "Microservices", *IEEE Software*, Vol. 32 No. 1, pp. 116–116.

Tip, F. (1995), "A Survey of Program Slicing Techniques", *Journal of Programming Languages*, Vol. 5399 No. 3, pp. 1–65.

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F. and Edmonds, A. (2015), "An architecture for self-managing microservices", *Proceedings: AIMC 2015 - Automated Incident Management in Cloud, 1st International Workshop, in Conjunction with EuroSYS 2015*, Association for Computing Machinery, Inc, New York, New York, USA, pp. 19–24.

Vaughan-Nichols, S. (2017), "What is Docker and why is it so darn popular? | ZDNet", *ZDNet*, available at: https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/ (accessed 12 February 2019).

Vianden, M., Lichter, H. and Steffens, A. (2014), "Experience on a Microservice-based reference architecture for measurement systems", *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, Vol. 1, IEEE Computer

Society, pp. 183–190.
Viennot, N., Lécuyer, M., Bell, J., Geambasu, R. and Nieh, J. (2015), “Synapse: A microservices architecture for heterogeneous-database web applications”, *Proceedings of the 10th European Conference on Computer Systems, EuroSys 2015*, Association for Computing Machinery, Inc, New York, New York, USA, pp. 1–16.
Vijayendra, M. (2019), “Microservices Sample”, available at: https://github.com/microservices-sample (accessed 5 September 2019).
Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R. and Gil, S. (2015), “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”, *Computing Colombian Conference (10CCC), 2015 10th*, pp. 583–590.
Westheide, D. (2016), “Why RESTful communication between microservices can be perfectly fine – INNOQ”, available at: https://www.innoq.com/en/blog/why-restful-communication-between-microservices-can-be-perfectly-fine/ (accessed 28 January 2020).
Wieringa, R., Maiden, N., Mead, N. and Rolland, C. (2006), “Requirements engineering paper classification and evaluation criteria: A proposal and a discussion”, *Requirements Engineering*, Vol. 11 No. 1, pp. 102–107.
Wong, K. (1998), “Rigi user’s manual”, *Department of Computer Science, University of Victoria*, available at: http://www.rigi.cs.uvic.ca/downloads/pdf/rigi-5_4_4-manual.pdf.
Woods, D. (2015), “Building Microservices with Spring Boot”, *InfoQ*, available at: https://www.infoq.com/articles/boot-microservices (accessed 12 February 2019).
Woods, E. (2016), “Software Architecture in a Changing World”, *IEEE Software*, IEEE Computer Society, Vol. 33 No. 6, pp. 94–97.
Yuqiong, S., Susanta, N. and Trent, J. (2015), “Security-as-a-Service for Microservices-Based Cloud Applications ”, *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, available at: https://ieeexplore.ieee.org/abstract/document/7396137 (accessed 31 March 2020).
Zeiner, H., Goller, M., Expósito Jiménez, V.J., Salmhofer, F. and Haas, W. (2016), “SeCoS: Web of Things platform based on a microservices architecture and support of time-awareness”, *Elektrotechnik Und Informationstechnik*, Springer-Verlag Wien, Vol. 133 No. 3, pp. 158–162.
Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C. and Zhao, W. (2018), “Benchmarking microservice systems for software engineering research”, *Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings - ICSE ’18*, pp. 323–324.

Appendices

Appendix-A: 1- Recovery Design Phase: Source artefacts collected.

Docker Compose Files	
Filename	URL
docker-compose.dev.yml	https://github.com/sqshq/PiggyMetrics
docker-compose.yml	https://github.com/sqshq/PiggyMetrics
docker-compose.yml [34]	https://github.com/sqshq/ELK-docker/blob/master/docker-compose.yml

List of Service Projects						
account-service	config	gateway	auth-service	kibana	elasticsearch	rabbitmq
statistics-service	registry	monitoring	turbine-stream service		notification-service	logstash

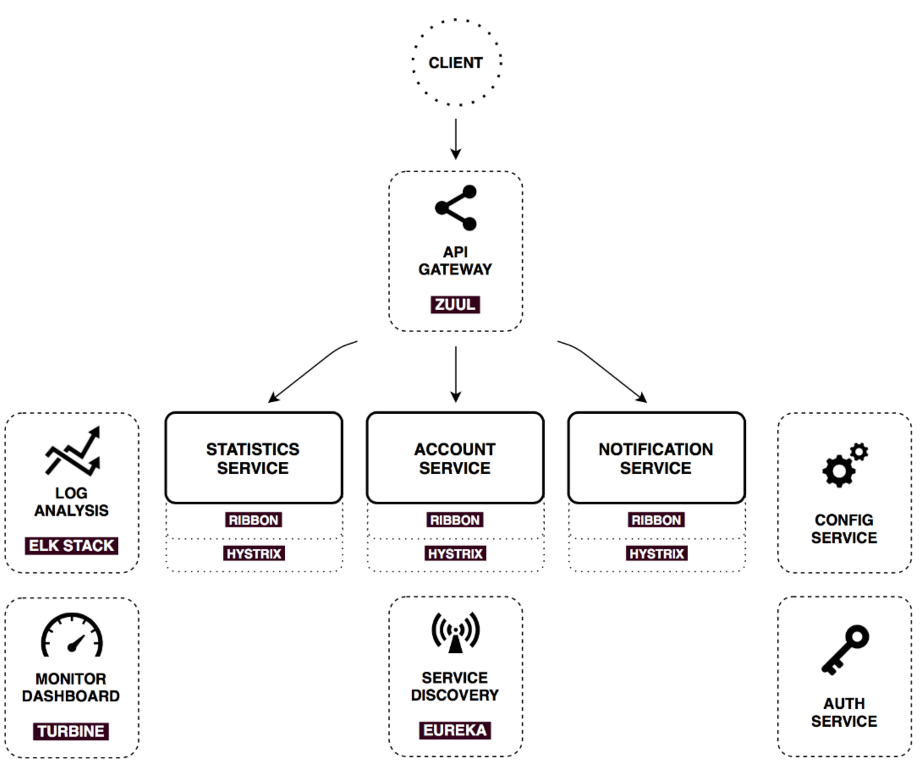
System Project Build Files	
Filename	URL
pom.xml	https://github.com/sqshq/PiggyMetrics

Service Project Build Files		
Service Project Name	Filename	URL
account-service	pom.xml	https://github.com/sqshq/PiggyMetrics/tree/master/account-service
auth-service	pom.xml	https://github.com/sqshq/PiggyMetrics/tree/master/auth-service
config	pom.xml	https://github.com/sqshq/PiggyMetrics/tree/master/config
gateway	pom.xml	https://github.com/sqshq/PiggyMetrics/tree/master/gateway
monitoring	pom.xml	https://github.com/sqshq/PiggyMetrics/tree/master/monitoring
notification-service	pom.xml	https://github.com/sqshq/PiggyMetrics/tree/master/notification-service
registry	pom.xml	https://github.com/sqshq/PiggyMetrics/tree/master/registry
statistics-service	pom.xml	https://github.com/sqshq/PiggyMetrics/tree/master/statistics-service
turbine-stream-service	pom.xml	https://github.com/sqshq/PiggyMetrics/tree/master/turbine-stream-service

Count of Configuration Files for Service Project[35]		
Service Name	Count	URL and Location
account-service	3	Local https://github.com/sqshq/PiggyMetrics/blob/master/account-service/src/main/resources/bootstrap.yml Remote https://github.com/sqshq/PiggyMetrics/blob/master/config/src/main/resources/shared/application.yml https://github.com/sqshq/PiggyMetrics/blob/master/config/src/main/resources/shared/account-service.yml
config	2	Local https://github.com/sqshq/PiggyMetrics/blob/master/config/src/main/resources/application.yml Remote https://github.com/sqshq/PiggyMetrics/blob/master/config/src/main/resources/shared/application.yml

[34] For log analysis microservices.
[35] For account-service and config microservices.

Count of Service Java Source Files	
Service Project Name	Java Source File Count
account-service	24
auth-service	9
config	2
gateway	1
monitoring	1
notification-service	17
registry	1
statistics-service	23
turbine-stream-service	1
Architecture Diagram	



Appendix-A: 2- Recovery Design Phase: Extracted information from both static and dynamic analysis.

Analysis Type	Extracted Source File	Extracted Information	Type of Extraction (Manual/Tool)
Static Analysis	Docker file	❖ **Commands**: For example, command EXPOSE 6000 as in Docker file[36] indicates the exposed port number at which the service's container account-service is running.	Manual
	Docker compose file	❖ **Service definition keys**: For example, keys rabbitmq, config, statistics-service, account-service, notification-service, auth-service, gateway, auth-mongodb, account-mongodb, statistics-mongodb, notification-mongodb, monitoring and registry keys in docker-compose.yml indicate services deployed to Docker containers. ❖ **Configuration keys and their options**: For example, the configuration key ports under service definition rabbitmq indicates the exposed port number at which the service's container is running, while the key *depends* on under service definition registry indicates its dependencies, i.e. which services should be running first.	Manual
	Java source code	❖ **Imported libraries and classes**: For example, importing the package 'org.springframework.data' indicates that this service implements some sort of data storage, as in line 5 and 6 of Account.java.[37]	Manual GitHub-metadat a

[36] https://github.com/sqshq/PiggyMetrics/blob/master/account-service/Dockerfile.
[37] https://github.com/sqshq/PiggyMetrics/blob/master/account-service/src/main/java/com/piggymetrics/account/domain/Account.java.

		❖ **Java annotations** (class-level, method-level, field-level): For example, the POJO class, e.g. Account.java, annotated with @Document indicates that a this class is mapped to a Mongo NoSQL data store document named ‘Account’ while the field level annotation @Id, at line 17 of Account.java, identifies the primary key for this document, e.g. the field name. ❖ **Java annotations’ parameters**: For example, the path parameter of @RequestMapping in a controller class, e.g. AccountController.java,[38] indicates the URL of the REST endpoint exposed by the service while the method parameter indicates the HTTP method type of the exposed request, e.g. GET as in line 20 of AccountController.java. ❖ **Java user-defined types’ super types**: For example, defining a class that extends AuthorizationServerConfigurerAdapter indicates that this service is an OAuth2 security server, as in OAuth2AuthorizationConfig.java.[39] ❖ **Java field types**: For example, fields of type OAuth2RestOperations and/or OAuth2RestTemplate as in line 40 and 119, respectively, of CustomUserInfoTokenServices.java[40] indicate that this service has a client component of authorisation pattern.	Enterprise architecture tool, visual paradigm tool

[38] https://github.com/sqshq/PiggyMetrics/blob/master/account-service/src/main/java/com/piggymetrics/account/controller/AccountController.java.
[39] https://github.com/sqshq/PiggyMetrics/blob/master/auth-service/src/main/java/com/piggymetrics/auth/config/OAuth2AuthorizationConfig.java.
[40] https://github.com/sqshq/PiggyMetrics/blob/master/account-service/src/main/java/com/piggymetrics/account/service/security/CustomUserInfoTokenServices.java.

		❖ **Java methods** (definition, invocation): For example, an invocation to the method RestTemplate::getForEntity() as in line 129 of CustomUserInfoTokenServices.java indicates that this service is a client that will request data from a remote service.	
	Maven build file (system level)	❖ **Project-XML-element**: Forexample, <artifactId>piggymetrics</artifactId> of <project> element at line 6 of pom.xml[41] indicates the name of the architecture is piggymetrics. ❖ **Module XML element**: For example, the value of <module> elements at lines 37-45 of pom.xml indicate the name of microservices composing the architecture.	Manual
	Maven build file (service level)	❖ **Project XML element**: For example, <artifactId> account-service </artifactId> of <project> element at line 6 of pom.xml[42] indicates the name of the microservice is account-service. ❖ **Dependency XML element**: For example, <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>of<dependency> element at line 41 of pom.xml indicates that this service uses client components of service registry and discovery pattern.	Manual

[41] https://github.com/sqshq/PiggyMetrics/blob/master/pom.xml.
[42] https://github.com/sqshq/PiggyMetrics/blob/master/account-service/pom.xml.

	Spring Boot configurations file (service level)	❖ **Configuration properties**: For example, property “feign:hystrix:enabled: true” at lines 24-26 of account-service.yml[43] indicates that this service uses the client component of monitoring pattern.	Manual
Dynamic Analysis	Traces	❖ Traces provide us information about how microservices are communicating with another, a sequence of various calls between different microservices. Traces provide us information about a single message call from an external system or UI to a microservice which in turn calls other microservices. I was able to construct service call graphs that describe runtime sequence and dependencies between different interconnected microservices.	Zipkin tool
	Logs	❖ Logs provide us information about how the system is behaving and if there are any errors or exceptions that occurred while system operated.	Docker logs command
	Containers	❖ Image name, IP addresses.	Docker inspect command
	Network trace/logs	❖ Confirmation of the connectivity between various microservices. This information is redundant to traces which provide similar but higher-level information. The difference between Zipkin traces and network logs is that Zipkin traces are designed specifically for microservice communication whereas	Tcpdump

[43] https://github.com/sqshq/PiggyMetrics/blob/master/config/src/main/resources/shared/account-service.yml.

		network traces provide low-level TCP connection and communication information.

Appendix-A: 3- A Summary of microservice systems used in Recovery Extraction phase

Ref.	System Name	Microservices	Technologies Used	Corresponding Architecture Element from Technology
[1]	piggymetrics	• account-service • statistics-service • notification-service • auth-service • config • registry • gateway • auth-mongodb • account-mongodb • statistics-mongodb • notification-mongodb • monitoring • rabbitmq	Docker	Containerization
			Spring Boot / Cloud	Development Framework
			Netflix Zuul	API Gateway
			Spring Cloud Config	Configuration
			Netflix Eureka	Registry and Discovery
			Netflix Ribbon	Load Balancing
			Netflix Feign	Microservice Discovery Client
			Spring Cloud Security with OAuth2	Security
			Netflix Hystrix	Circuit Breaker
			Netflix Hystrix Dashboard with Turbine	Monitoring
			Elasticsearch, Logstash and Kibana (ELK)	Log Analysis
			MongoDB	Data Store
			RabbitMQ Message Broker	Asynchronous Communication
[2]	microservice-blog	• employee • reward	Docker	Containerization
			Spring Boot / Cloud	Development Framework
			Tutom / HAProxy	Load Balancing

		• mission • mongodb • ha_employee • ha_mission • ha_reward	MongoDB	Data Store
[3]	spmia-chapter10	• authenticationservice • configserver • eurekaserver • licensingservice • organizationservice • zuulserver • kafkaserver	Docker	Containerization
			Spring Boot / Cloud	Development Framework
			Spring Cloud Config	Configuration
			Netflix Zuul	API Gateway
			Netflix Eureka	Registry and Discovery
			Netflix Ribbon	Load Balancing
			Netflix Feign	Microservice Discovery Client
			Spring Cloud Security with OAuth2	Security
			Netflix Hystrix	Circuit Breaker
			Spring Sleuth with Zipkin	Tracing
			MongoDB	Data Store
			Redis	Cache Store
			Kafka Message Broker	Asynchronous Communication
[4]	microservice-consul	• customer • catalog • order • consul • apache • hystrix-dashboard • zipkin • prometheus • filebeat • elasticsearch • kibana	Docker	Containerization
			Spring Boot / Cloud	Development Framework
			Apache HTTP	API Gateway
			Consul Discovery	Registry and Discovery
			Netflix Ribbon	Load Balancing
			Netflix Hystrix	Circuit Breaker
			Spring Sleuth with Zipkin	Tracing
			Netflix Hystrix Dashboard with Prometheus	Monitoring
			Elasticsearch, Filebeat and Kibana (ELK)	Log Analysis
			HSQLDB	Data Store
[5]	spring-cloud-consul-example	• service-a • service-b • consul • zuul • admin-dashboard • hystrix-dashboard • zipkin	Docker	Containerization
			Spring Boot / Cloud	Development Framework
			Consul Config	Configuration
			Netflix Zuul	API Gateway
			Consul Discovery	Registry and Discovery
			Netflix Ribbon	Load Balancing
			Netflix Feign	Microservice Discovery Client
			Netflix Hystrix	Circuit Breaker

			Netflix Hystrix Dashboard with Turbine	Monitoring
			Spring Sleuth with Zipkin	Tracing
			RabbitMQ Message Broker	Asynchronous Communication
[6]	spring-cloud-netflix-example	• service-a • service-b • config-server • eureka-server • zuul • admin-dashboard • hystrix-dashboard • zipkin • rabbitmq	Docker	Containerization
			Spring Boot / Cloud	Development Framework
			Netflix Zuul	API Gateway
			Spring Cloud Config	Configuration
			Netflix Eureka	Registry and Discovery
			Netflix Ribbon	Load Balancing
			Netflix Feign	Microservice Discovery Client
			Netflix Hystrix Dashboard with Turbine	Monitoring
			Spring Sleuth with Zipkin	Tracing
			RabbitMQ Message Broker	Asynchronous Communication
[7]	microservices-sidecar-example	• simple1 • simple2 • railsdemo • eureka • zuul	Docker	Containerization
			Spring Boot / Cloud	Development Framework
			Netflix Zuul	API Gateway
			Netflix Eureka	Registry and Discovery
			Netflix Ribbon	Load Balancing
			Netflix Sidecar	Non-JVM API Gateway
[8]	blog-microservices	• discovery • config • auth-server • product-service • recommendation-service • review-service • composite-service • monitor-dashboard • edge-server • zipkin-server • rabbitmq • logstash • elasticsearch • kibana	Docker	Containerization
			Spring Boot / Cloud	Development Framework
			Netflix Zuul	API Gateway
			Spring Cloud Config	Configuration
			Netflix Eureka	Registry and Discovery
			Netflix Ribbon	Load Balancing
			Netflix Feign	Microservice Discovery Client
			Netflix Hystrix Dashboard with Turbine	Monitoring
			Spring Sleuth with Zipkin	Tracing
			RabbitMQ Message Broker	Asynchronous Communication

Appendix-A: 4- Mapping Rules (104 in total) extracted from case study 1 (PiggyMetrics)

ID	Artefact Type (PSM)	PIM Concept (Source)	PIM Concept (Destination)	Mapping Rule (PSM -> PIM)
1	GitHub Repository	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the name of the root GitHub Repository which contains all artifacts of the application's project.
2	Build File	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the value of <project><artifactId> key in the Build File of the application's project.
3	Build File	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the value of <project><parent><artifactId> key in the Build File of the microservice's project.
4	Build File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of <project><modules><module> key in the Build File of the application's project.
5	Build File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of <project><artifactId> key in the Build File of the microservice's project.
6	Build File	Microservice	Container	The name of Container concept is indicated by the value of <project><modules><module> key in the Build File of the application's project.
7	Build File	Microservice	Container	The name of Container concept is indicated by the value of <project><artifactId> key in the Build File of the microservice's project.
8	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-zuul' in the Build File of the microservice's project.
9	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-server' in the Build File of the microservice's project.
10	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-client' in the Build File of the microservice's project.

11	Build File	Microservice	Service Interface	The server path of Service Interface concept is indicated by the value of <project><modules><module> key in the Build File of the application's project.
12	Build File	Microservice	Service Interface	The server path of Service Interface concept is indicated by the value of <project><artifactId> key in the Build File of the microservice's project.
13	Build File	Microservice	Service Dependency	A 'MongoDB' Infrastructure Microservice provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-boot-starter-data-mongodb' in the Build File of the microservice's project.
14	Build File	Microservice	Service Dependency	A 'Netflix Turbine' Monitoring provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-hystrix-dashboard' in the Build File of the microservice's project.
15	Build File	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-config' in the Build File of the microservice's project.
16	Build File	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-client' in the Build File of the microservice's project.
17	Build File	Microservice	Service Dependency	A 'RabbitMQ' Infrastructure Microservice provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-stream-rabbit' or 'spring-cloud-starter-bus-amqp' in the Build File of the microservice's project.
18	Build File	API Gateway	-	An API Gateway concept with technology of 'Netflix Zuul' is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-zuul' in the Build File of the microservice's project.
19	Build File	Configuration	-	A 'Spring Cloud Config' Configuration concept is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-config-server' in the Build File of the microservice's project.
20	Build File	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-server' in the Build File of the microservice's project.

21	Build File	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-client' in the Build File of the microservice's project.
22	Build File	Monitoring	-	A 'Netflix Hystrix Dashboard' Monitoring is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-hystrix-dashboard' in the Build File of the microservice's project.
23	Build File	Monitoring	-	A 'Netflix Turbine' Monitoring is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-turbine-stream' in the Build File of the microservice's project.
24	Build File	Monitoring	Service Dependency	A 'Netflix Turbine' Monitoring provider to a 'Netflix Hystrix Dashboard' Monitoring is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-turbine-stream' in the Build File of the microservice's project.
25	Build File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-hystrix' or 'spring-cloud-netflix-hystrix-stream' in the Build File of the microservice's project.
26	Build File	Tracing	Service Dependency	A Microservice provider to a 'Zipkin' Tracing is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-sleuth' in the Build File of the microservice's project.
27	Configurations File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice's project.
28	Configurations File	Microservice	Container	The name of Container conceptis indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice's project.
29	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.
30	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property name that starts with 'zuul.routes.' and ends with '.serviceId:' in the Configurations File of the microservice's project.

31	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'ribbon.ReadTimeout:' or 'ribbon.ConnectTimeout:' with nonzero values in the Configurations File of the microservice's project.
32	Configurations File	Microservice	Service Interface	The server path of Service Interface concept is indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice's project.
33	Configurations File	Microservice	Service Interface	A prefix to server path of Service Interface concept is indicated by the value of property 'server.servlet.contextPath:' in the Configurations File of the microservice's project.
34	Configurations File	Microservice	Service Dependency	A 'MongoDB' Infrastructure Microservice provider to a Microservice is indicated by the value of property 'spring.data.mongodb.host:' in the Configurations File of the microservice's project.
35	Configurations File	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by the hostname section of the url value of the property 'spring.cloud.config.uri:' or 'spring.cloud.config.failFast: true' in the Configurations File of the microservice's project.
36	Configurations File	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by the hostname section of the url value of the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.
37	Configurations File	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by the hostname section of the url value of the property 'security.oauth2.resource.userInfoUri:' or 'security.oauth2.client.accessTokenUri:' in the Configurations File of the microservice's project.
38	Configurations File	Microservice	Service Dependency	A 'RabbitMQ' Infrastructure Microservice provider to a Microservice is indicated by the value of the property 'spring.rabbitmq.host:' in the Configurations File of the microservice's project.
39	Configurations File	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the value of the property 'spring.mail.host:' in the Configurations File of the microservice's project.
40	Configurations File	API Gateway	Service Dependency	A Microservice provider to a 'Netflix Zuul' API Gateway is indicated by the hostname section of the url value of the property that starts with 'zuul.routes.' and ends with '.url:' in the Configurations File of the microservice's project.
41	Configurations File	API Gateway	Service Dependency	A Microservice provider to a 'Netflix Zuul' API Gateway is indicated by the value of the property that starts with 'zuul.routes.' and ends with '.serviceId:' in the Configurations File of the microservice's project.

42	Configurations File	API Gateway	-	An API Gateway concept with technology of 'Netflix Zuul' is indicated by the property name that starts with 'zuul.routes.' in the Configurations File of the microservice's project.
43	Configurations File	Configuration	-	A 'Spring Cloud Config' Configuration concept is indicated by the property 'spring.cloud.config.server.native.searchLocations:' and the property 'profiles.active:' with value 'native' in the Configurations File of the microservice's project.
44	Configurations File	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by the two properties 'eureka.client.registerWithEureka: false' and 'eureka.client.fetchRegistry: false' in the Configurations File of the microservice's project.
45	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the two properties 'eureka.client.registerWithEureka: false' and 'eureka.client.fetchRegistry: false' in the Configurations File of the microservice's project.
46	Configurations File	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.
47	Configurations File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by the non-zero property 'hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:' or the property 'feign.hystrix.enabled: true' in the Configurations File of the microservice's project.
48	Configurations File	Log Analysis	Service Dependency	A Microservice provider to a Log Analysis is indicated by the property name that starts with 'logging.level.' in the Configurations File of the microservice's project.
49	Configurations File	Service Interface	Endpoint	An Endpoint to Service Interface is indicated by the value of the property that starts with 'zuul.routes.' and ends with '.path:' in the Configurations File of the microservice's project.
50	Configurations File	Service Operation	Circuit Breaker	A 'Netflix Hystrix' Circuit Breaker to Service Operation is indicated by the non-zero property 'hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:' or the property 'feign.hystrix.enabled: true' in the Configurations File of the microservice's project.
51	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'eureka.server.waitTimeInMsWhenSyncEmpty:' in the Configurations File of the microservice's project.

52	Configurations File	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by the property 'eureka.server.waitTimeInMsWhenSyncEmpty:' in the Configurations File of the microservice's project.
53	Source Code File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by last section of package name of a Java Class with '@SpringBootApplication' annotation in the Source Code File of the microservice's project.
54	Source Code File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by last section of package name of a Java Class with Java Method with identifier of 'main' that invockes another Java Method with identifier of 'SpringApplication.run' in the Source Code File of the microservice's project.
55	Source Code File	Microservice	Container	The name of Container conceptis indicated by last section of package name of a Java Class with '@SpringBootApplication' annotation in the Source Code File of the microservice's project.
56	Source Code File	Microservice	Container	The name of Container concept is indicated by last section of package name of a Java Class with Java Method with identifier of 'main' that invockes another Java Method with identifier of 'SpringApplication.run' in the Source Code File of the microservice's project.
57	Source Code File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a Java Class with '@EnableZuulProxy' annotation in the Source Code File of the microservice's project.
58	Source Code File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a Java Class with '@EnableEurekaServer' annotation in the Source Code File of the microservice's project.
59	Source Code File	Microservice	Service Interface	The server path of Service Interface concept is indicated by last section of package name of a Java Class with '@SpringBootApplication' annotation in the Source Code File of the microservice's project.
60	Source Code File	Microservice	Service Interface	The server path of Service Interface concept is indicated by last section of package name of a Java Class with Java Method with identifier of 'main' that invockes another Java Method with identifier of 'SpringApplication.run' in the Source Code File of the microservice's project.
61	Source Code File	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by a Java Class with '@EnableResourceServer' or 'EnableOAuth2Client' annotation in the Source Code File of the microservice's project.

62	Source Code File	Microservice	Service Dependency	A Registry and Discovery provider to a Microservice is indicated by a Java Class with '@EnableDiscoveryClient' annotation in the Source Code File of the microservice's project.
63	Source Code File	Microservice	Service Dependency	A 'Netflix Turbine' Monitoring provider to a Microservice is indicated by a Java Class with '@EnableHystrixDashboard' annotation in the Source Code File of the microservice's project.
64	Source Code File	Microservice	Service Dependency	A 'MongoDB' Infrastructure Microservice provider to a Microservice is indicated by a Java Method 'save()' or 'findById()' or any Java Method that starts with 'find' of a Java Interface that extends 'CrudRepository' Java Interface of package 'org.springframework.data.repository' in the Source Code File of the microservice's project.
65	Source Code File	Microservice	Service Dependency	A ' MongoDB' Infrastructure Microservice provider to a Microservice is indicated by a Java Interface that extends 'CrudRepository' Java Interface of package 'org.springframework.data.repository' in the Source Code File of the microservice's project.
66	Source Code File	Microservice	Service Dependency	A 'MongoDB' Infrastructure Microservice provider to a Microservice is indicated by a Java Class with '@Document' annotation in the Source Code File of the microservice's project.
67	Source Code File	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by a Java Class that implements 'ResourceServerTokenServices' Java Interface of package 'org.springframework.security.oauth2.provider.token' in the Source Code File of the microservice's project.
68	Source Code File	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by a Java Class that extends 'ResourceServerConfigurerAdapter' Java Class of package 'org.springframework.security.oauth2.provider.token' in the Source Code File of the microservice's project.
69	Source Code File	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the URL in the first argument of a Java Method 'getForEntity()' of a Java Interface 'OAuth2RestOperations' of package 'org.springframework.security.oauth2.config.annotation.web.configuration' in the Source Code File of the microservice's project.
70	Source Code File	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by a Java Method 'getForEntity()' of a Java Interface 'OAuth2RestOperations' of package 'org.springframework.security.oauth2.client' in the Source Code File of the microservice's project.

71	Source Code File	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by a Java Method with '@ConfigurationProperties' or '@Value' annotation in the Source Code File of the microservice's project.
72	Source Code File	Microservice	Service Dependency	A Microservice provider name to a Microservice is indicated by the value of 'name' parameter of '@FeignClient' Java Interface annotation in the Source Code File of the microservice's project.
73	Source Code File	Microservice	Service Dependency	A Microservice provider operation to a Microservice is indicated by a Java Method with '@RequestMapping' annotation that belongs to a Java Interface with '@FeignClient' annotation in the Source Code File of the microservice's project.
74	Source Code File	API Gateway	-	A 'Netflix Zuul' API Gateway is indicated by a Java Class with '@EnableZuulProxy' annotation in the Source Code File of the microservice's project.
75	Source Code File	Configuration	-	A 'Spring Cloud Config' Configuration concept is indicated by a Java Class with '@EnableConfigServer' annotation in the Source Code File of the microservice's project.
76	Source Code File	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by a Java Class with '@EnableEurekaServer' annotation in the Source Code File of the microservice's project.
77	Source Code File	Registry and Discovery	Service Dependency	A Microservice provider to a Registry and Discovery is indicated by a Java Class with '@EnableDiscoveryClient' annotation in the Source Code File of the microservice's project.
78	Source Code File	Security	-	A 'Spring Cloud OAuth2' Security is indicated by a Java Class with '@EnableAuthorizationServer' annotation in the Source Code File of the microservice's project.
79	Source Code File	Security	-	A 'Spring Cloud OAuth2' Security is indicated by a Java Class that extends a Java Class 'AuthorizationServerConfigurerAdapter' of package 'org.springframework.security.oauth2.config.annotation.web.configuration' and overrides a Java Method 'configure()' in the Source Code File of the microservice's project.
80	Source Code File	Monitoring	-	A 'Netflix Hystrix Dashboard' Monitoring is indicated by a Java Class with '@EnableHystrixDashboard' annotation in the Source Code File of the microservice's project.
81	Source Code File	Monitoring	-	A 'Netflix Turbine' Monitoring is indicated by a Java Class with '@EnableTurbineStream' annotation in the Source Code File of the microservice's project.

82	Source Code File	Monitoring	Service Dependency	A 'Netflix Turbine' Monitoring provider to a 'Netflix Hystrix Dashboard' Monitoring is indicated by a Java Class with '@EnableTurbineStream' annotation in the Source Code File of the microservice's project.
83	Source Code File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by a Java Class with '@EnableCircuitBreaker' annotation in the Source Code File of the microservice's project.
84	Source Code File	Log Analysis	Service Dependency	A Microservice provider to a Log Analysis is indicated by a Java Method 'info()' or 'debug()' or 'error()' of a Java Class 'Logger' of package 'org.slf4j' or a Java Class 'Log' of package 'org.apache.commons.logging' in the Source Code File of the microservice's project.
85	Source Code File	Service Interface	Endpoint	A prefix to path of Endpoint concept is indicated by the value of '@RequestMapping' Java Class annotation in the Source Code File of the microservice's project.
86	Source Code File	Service Interface	Endpoint	The path of Endpoint concept is indicated by the value of 'method' parameter and 'value' or 'path' parameter in '@RequestMapping' Java Method annotation that belongs to a Java Class with '@RestController' annotation in the Source Code File of the microservice's project.
87	Source Code File	Service Interface	Service Operation	A Service Operation concept is indicated by a Java Method with '@RequestMapping' or '@ResponseStatus' annotation that belongs to a Java Class with '@RestController' or '@ControllerAdvice' annotation respectively in the Source Code File of the microservice's project.
88	Source Code File	Service Interface	Service Operation	A Service Operation concept is indicated by a Java Method with '@Scheduled' annotation in the Source Code File of the microservice's project.
89	Source Code File	Service Operation	Data Store	A Data Store to a Service Operation is indicated by a Java Method 'save()' or 'findById()' or any Java Method that starts with 'find' of a Java Interface that extends 'CrudRepository' Java Interface of package 'org.springframework.data.repository' in the Source Code File of the microservice's project.
90	Source Code File	Service Operation	Circuit Breaker	A Circuit Breaker to a Service Operation is indicated by a Java Method with '@RequestMapping' annotation that belongs to a Java Interface with '@FeignClient' annotation in the Source Code File of the microservice's project.

91	Source Code File	Microservice	Service Dependency	A 'Consul' Configuration provider to a Microservice is indicated by a Java Class with '@RefreshScope' annotation in the Source Code File of the microservice's project.
92	Container Build File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the JAR application name argument of 'ADD' command in the Container Build File of the microservice's project.
93	Container Build File	Microservice	Container	The name of Container concept is indicated by the JAR application name argument of 'ADD' command in the Container Build File of the microservice's project.
94	Container Build File	Microservice	Service Interface	The server path of Service Interface concept is indicated by the JAR application name argument of 'ADD' command in the Container Build File of the microservice's project.
95	Container Build File	Infrastructure Microservice	-	A 'MongoDB Data Store' Infrastructure Microservice concept is indicated by a 'FROM' command with argument value that starts with 'mongo:' in the Container Build File of the microservice's project.
96	Container Build File	Log Analysis	-	A Log Analysis concept is indicated by a 'FROM' command with argument value that starts with 'logstash:' in the Container Build File of the microservice's project.
97	Container Orchestration File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.
98	Container Orchestration File	Microservice	Container	The name of Container concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.
99	Container Orchestration File	Microservice	Service Interface	The server path of Service Interface concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.
100	Container Orchestration File	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the service container name of 'depends_on' or 'links' key in the Container Orchestration File of the application's project.
101	Container Orchestration File	Infrastructure Microservice	-	An Infrastructure Microservice concept is indicated by a service container definition that does not have 'build' key in the Container Orchestration File of the application's project.
102	Container Orchestration File	Infrastructure Microservice	-	A 'RabbitMQ' Infrastructure Microservice concept is indicated by an 'image:' key with value that starts wth 'rabbitmq:' in the Container Orchestration File of the application's project.

103	Container Orchestration File	Log Analysis	-	A Log Analysis concept is indicated by an 'image:' key with value that starts wth 'elasticsearch:' or 'kibana:' in the Container Orchestration File of the application's project.
104	Container Orchestration File	Log Analysis	Service Dependency	A Microservice provider to a Log Analysis is indicated by the key name of service container definition that has 'logging' or 'log_opt' key in the Container Orchestration File of the application's project.

Appendix-A: 5- Refined mapping Rules (268 in total) extracted from case studies 1 to 8

ID	CS[44]	Artefact Type (PSM)	PIM Concept (Source)	PIM Concept (Destination)	Mapping Rule (PSM -> PIM)
1	[1]	GitHub Repository	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the name of the root GitHub Repository which contains all artifacts of the application's project.
2	[1]	Build File	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the value of <project><artifactId> key in the Build File of the application's project.
3	[1]	Build File	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the value of <project><parent><artifactId> key in the Build File of the microservice's project.
4	[1]	Build File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of <project><modules><module> key in the Build File of the application's project.
5	[1]	Build File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of <project><artifactId> key in the Build File of the microservice's project.
6	[1]	Build File	Microservice	Container	The name of Container concept is indicated by the value of <project><modules><module> key in the Build File of the application's project.

[44] Case study.

7	[1]	Build File	Microservice	Container	The name of Container concept is indicated by the value of <project><artifactId> key in the Build File of the microservice's project.
8	[1]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-zuul' in the Build File of the microservice's project.
9	[1]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-server' in the Build File of the microservice's project.
10	[1]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-client' in the Build File of the microservice's project.
11	[1]	Build File	Microservice	Service Interface	The server path of Service Interface concept is indicated by the value of <project><modules><module> key in the Build File of the application's project.
12	[1]	Build File	Microservice	Service Interface	The server path of Service Interface concept is indicated by the value of <project><artifactId> key in the Build File of the microservice's project.
13	[1]	Build File	Microservice	Service Dependency	A 'MongoDB' Infrastructure Microservice provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-boot-starter-data-mongodb' in the Build File of the microservice's project.
14	[1]	Build File	Microservice	Service Dependency	A 'Netflix Turbine' Monitoring provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-hystrix-dashboard' in the Build File of the microservice's project.
15	[1]	Build File	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-config' in the Build File of the microservice's project.
16	[1]	Build File	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value

					'spring-cloud-starter-netflix-eureka-client' in the Build File of the microservice's project.
17	[1]	Build File	Microservice	Service Dependency	A 'RabbitMQ' Infrastructure Microservice provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-stream-rabbit' or 'spring-cloud-starter-bus-amqp' in the Build File of the microservice's project.
18	[1]	Build File	API Gateway	-	An API Gateway concept with technology of 'Netflix Zuul' is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-zuul' in the Build File of the microservice's project.
19	[1]	Build File	Configuration	-	A 'Spring Cloud Config' Configuration concept is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-config-server' in the Build File of the microservice's project.
20	[1]	Build File	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-server' in the Build File of the microservice's project.
21	[1]	Build File	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-eureka-client' in the Build File of the microservice's project.
22	[1]	Build File	Monitoring	-	A 'Netflix Hystrix Dashboard' Monitoring is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-hystrix-dashboard' in the Build File of the microservice's project.
23	[1]	Build File	Monitoring	-	A 'Netflix Turbine' Monitoring is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-turbine-stream' in the Build File of the microservice's project.
24	[1]	Build File	Monitoring	Service Dependency	A 'Netflix Turbine' Monitoring provider to a 'Netflix Hystrix Dashboard' Monitoring is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-turbine-stream' in the Build File of the microservice's project.

25	[1]	Build File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-hystrix' or 'spring-cloud-netflix-hystrix-stream' in the Build File of the microservice's project.
26	[1]	Build File	Tracing	Service Dependency	A Microservice provider to a 'Zipkin' Tracing is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-sleuth' in the Build File of the microservice's project.
27	[1]	Configurations File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice's project.
28	[1]	Configurations File	Microservice	Container	The name of Container conceptis indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice's project.
29	[1]	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.
30	[1]	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property name that starts with 'zuul.routes.' and ends with '.serviceId:' in the Configurations File of the microservice's project.
31	[1]	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'ribbon.ReadTimeout:' or 'ribbon.ConnectTimeout:' with nonzero values in the Configurations File of the microservice's project.
32	[1]	Configurations File	Microservice	Service Interface	The server path of Service Interface concept is indicated by the value of the property 'spring.application.name:' in the Configurations File of the microservice's project.
33	[1]	Configurations File	Microservice	Service Interface	A prefix to server path of Service Interface concept is indicated by the value of property 'server.servlet.contextPath:' in the Configurations File of the microservice's project.
34	[1]	Configurations File	Microservice	Service Dependency	A 'MongoDB' Infrastructure Microservice provider to a Microservice is indicated by the value of property 'spring.data.mongodb.host:' in the Configurations File of the microservice's project.
35	[1]	Configurations File	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by the hostname section of the url value of the property

					'spring.cloud.config.uri:' or 'spring.cloud.config.failFast: true' in the Configurations File of the microservice's project.
36	[1]	Configurations File	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by the hostname section of the url value of the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.
37	[1]	Configurations File	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by the hostname section of the url value of the property 'security.oauth2.resource.userInfoUri:' or 'security.oauth2.client.accessTokenUri:' in the Configurations File of the microservice's project.
38	[1]	Configurations File	Microservice	Service Dependency	A 'RabbitMQ' Infrastructure Microservice provider to a Microservice is indicated by the value of the property 'spring.rabbitmq.host:' in the Configurations File of the microservice's project.
39	[1]	Configurations File	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the value of the property 'spring.mail.host:' in the Configurations File of the microservice's project.
40	[1]	Configurations File	API Gateway	Service Dependency	A Microservice provider to a 'Netflix Zuul' API Gateway is indicated by the hostname section of the url value of the property that starts with 'zuul.routes.' and ends with '.url:' in the Configurations File of the microservice's project.
41	[1]	Configurations File	API Gateway	Service Dependency	A Microservice provider to a 'Netflix Zuul' API Gateway is indicated by the value of the property that starts with 'zuul.routes.' and ends with '.serviceId:' in the Configurations File of the microservice's project.
42	[1]	Configurations File	API Gateway	-	An API Gateway concept with technology of 'Netflix Zuul' is indicated by the property name that starts with 'zuul.routes.' in the Configurations File of the microservice's project.
43	[1]	Configurations File	Configuration	-	A 'Spring Cloud Config' Configuration concept is indicated by the property 'spring.cloud.config.server.native.searchLocations:' and the property 'profiles.active:' with value 'native' in the Configurations File of the microservice's project.
44	[1]	Configurations File	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by the two properties 'eureka.client.registerWithEureka: false' and 'eureka.client.fetchRegistry: false' in the Configurations File of the microservice's project.

45	[1]	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the two properties 'eureka.client.registerWithEureka: false' and 'eureka.client.fetchRegistry: false' in the Configurations File of the microservice's project.
46	[1]	Configurations File	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by the property 'eureka.client.serviceUrl.defaultZone:' in the Configurations File of the microservice's project.
47	[1]	Configurations File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by the non-zero property 'hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:' or the property 'feign.hystrix.enabled: true' in the Configurations File of the microservice's project.
48	[1]	Configurations File	Log Analysis	Service Dependency	A Microservice provider to a Log Analysis is indicated by the property name that starts with 'logging.level.' in the Configurations File of the microservice's project.
49	[1]	Configurations File	Service Interface	Endpoint	An Endpoint to Service Interface is indicated by the value of the property that starts with 'zuul.routes.' and ends with '.path:' in the Configurations File of the microservice's project.
50	[1]	Configurations File	Service Operation	Circuit Breaker	A 'Netflix Hystrix' Circuit Breaker to Service Operation is indicated by the non-zero property 'hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds:' or the property 'feign.hystrix.enabled: true' in the Configurations File of the microservice's project.
51	[1]	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'eureka.server.waitTimeInMsWhenSyncEmpty:' in the Configurations File of the microservice's project.
52	[1]	Configurations File	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by the property 'eureka.server.waitTimeInMsWhenSyncEmpty:' in the Configurations File of the microservice's project.
53	[1]	Source Code File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by last section of package name of a Java Class with '@SpringBootApplication' annotation in the Source Code File of the microservice's project.
54	[1]	Source Code File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by last section of package name of a Java Class with Java Method with identifier of 'main' that invockes another Java Method with identifier of

					'SpringApplication.run' in the Source Code File of the microservice's project.
55	[1]	Source Code File	Microservice	Container	The name of Container conceptis indicated by last section of package name of a Java Class with '@SpringBootApplication' annotation in the Source Code File of the microservice's project.
56	[1]	Source Code File	Microservice	Container	The name of Container concept is indicated by last section of package name of a Java Class with Java Method with identifier of 'main' that invockes another Java Method with identifier of 'SpringApplication.run' in the Source Code File of the microservice's project.
57	[1]	Source Code File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a Java Class with '@EnableZuulProxy' annotation in the Source Code File of the microservice's project.
58	[1]	Source Code File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a Java Class with '@EnableEurekaServer' annotation in the Source Code File of the microservice's project.
59	[1]	Source Code File	Microservice	Service Interface	The server path of Service Interface concept is indicated by last section of package name of a Java Class with '@SpringBootApplication' annotation in the Source Code File of the microservice's project.
60	[1]	Source Code File	Microservice	Service Interface	The server path of Service Interface concept is indicated by last section of package name of a Java Class with Java Method with identifier of 'main' that invockes another Java Method with identifier of 'SpringApplication.run' in the Source Code File of the microservice's project.
61	[1]	Source Code File	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by a Java Class with '@EnableResourceServer' or 'EnableOAuth2Client' annotation in the Source Code File of the microservice's project.
62	[1]	Source Code File	Microservice	Service Dependency	A Registry and Discovery provider to a Microservice is indicated by a Java Class with '@EnableDiscoveryClient' annotation in the Source Code File of the microservice's project.
63	[1]	Source Code File	Microservice	Service Dependency	A 'Netflix Turbine' Monitoring provider to a Microservice is indicated by a Java Class with '@EnableHystrixDashboard' annotation in the Source Code File of the microservice's project.
64	[1]	Source Code File	Microservice	Service Dependency	A 'MongoDB' Infrastructure Microservice provider to a Microservice is indicated by a Java Method 'save()' or 'findById()' or any Java Method that starts with 'find' of a Java Interface that

					extends 'CrudRepository' Java Interface of package 'org.springframework.data.repository' in the Source Code File of the microservice's project.
65	[1]	Source Code File	Microservice	Service Dependency	A ' MongoDB' Infrastructure Microservice provider to a Microservice is indicated by a Java Interface that extends 'CrudRepository' Java Interface of package 'org.springframework.data.repository' in the Source Code File of the microservice's project.
66	[1]	Source Code File	Microservice	Service Dependency	A 'MongoDB' Infrastructure Microservice provider to a Microservice is indicated by a Java Class with '@Document' annotation in the Source Code File of the microservice's project.
67	[1]	Source Code File	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by a Java Class that implements 'ResourceServerTokenServices' Java Interface of package 'org.springframework.security.oauth2.provider.token' in the Source Code File of the microservice's project.
68	[1]	Source Code File	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by a Java Class that extends 'ResourceServerConfigurerAdapter' Java Class of package 'org.springframework.security.oauth2.provider.token' in the Source Code File of the microservice's project.
69	[1]	Source Code File	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the URL in the first argument of a Java Method 'getForEntity()' of a Java Interface 'OAuth2RestOperations' of package 'org.springframework.security.oauth2.config.annotation.web.configuration' in the Source Code File of the microservice's project.
70	[1]	Source Code File	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by a Java Method 'getForEntity()' of a Java Interface 'OAuth2RestOperations' of package 'org.springframework.security.oauth2.client' in the Source Code File of the microservice's project.
71	[1]	Source Code File	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by a Java Method with '@ConfigurationProperties' or '@Value' annotation in the Source Code File of the microservice's project.
72	[1]	Source Code File	Microservice	Service Dependency	A Microservice provider name to a Microservice is indicated by the value of 'name' parameter of '@FeignClient' Java Interface annotation in the Source Code File of the microservice's project.

73	[1]	Source Code File	Microservice	Service Dependency	A Microservice provider operation to a Microservice is indicated by a Java Method with '@RequestMapping' annotation that belongs to a Java Interface with '@FeignClient' annotation in the Source Code File of the microservice's project.
74	[1]	Source Code File	API Gateway	-	A 'Netflix Zuul' API Gateway is indicated by a Java Class with '@EnableZuulProxy' annotation in the Source Code File of the microservice's project.
75	[1]	Source Code File	Configuration	-	A 'Spring Cloud Config' Configuration concept is indicated by a Java Class with '@EnableConfigServer' annotation in the Source Code File of the microservice's project.
76	[1]	Source Code File	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by a Java Class with '@EnableEurekaServer' annotation in the Source Code File of the microservice's project.
77	[1]	Source Code File	Registry and Discovery	Service Dependency	A Microservice provider to a Registry and Discovery is indicated by a Java Class with '@EnableDiscoveryClient' annotation in the Source Code File of the microservice's project.
78	[1]	Source Code File	Security	-	A 'Spring Cloud OAuth2' Security is indicated by a Java Class with '@EnableAuthorizationServer' annotation in the Source Code File of the microservice's project.
79	[1]	Source Code File	Security	-	A 'Spring Cloud OAuth2' Security is indicated by a Java Class that extends a Java Class 'AuthorizationServerConfigurerAdapter' of package 'org.springframework.security.oauth2.config.annotation.web.configuration' and overrides a Java Method 'configure()' in the Source Code File of the microservice's project.
80	[1]	Source Code File	Monitoring	-	A 'Netflix Hystrix Dashboard' Monitoring is indicated by a Java Class with '@EnableHystrixDashboard' annotation in the Source Code File of the microservice's project.
81	[1]	Source Code File	Monitoring	-	A 'Netflix Turbine' Monitoring is indicated by a Java Class with '@EnableTurbineStream' annotation in the Source Code File of the microservice's project.
82	[1]	Source Code File	Monitoring	Service Dependency	A 'Netflix Turbine' Monitoring provider to a 'Netflix Hystrix Dashboard' Monitoring is indicated by a Java Class with '@EnableTurbineStream' annotation in the Source Code File of the microservice's project.
83	[1]	Source Code File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by a Java Class with

					'@EnableCircuitBreaker' annotation in the Source Code File of the microservice's project.
84	[1]	Source Code File	Log Analysis	Service Dependency	A Microservice provider to a Log Analysis is indicated by a Java Method 'info()' or 'debug()' or 'error()' of a Java Class 'Logger' of package 'org.slf4j' or a Java Class 'Log' of package 'org.apache.commons.logging' in the Source Code File of the microservice's project.
85	[1]	Source Code File	Service Interface	Endpoint	A prefix to path of Endpoint concept is indicated by the value of '@RequestMapping' Java Class annotation in the Source Code File of the microservice's project.
86	[1]	Source Code File	Service Interface	Endpoint	The path of Endpoint concept is indicated by the value of 'method' parameter and 'value' or 'path' parameter in '@RequestMapping' Java Method annotation that belongs to a Java Class with '@RestController' annotation in the Source Code File of the microservice's project.
87	[1]	Source Code File	Service Interface	Service Operation	A Service Operation concept is indicated by a Java Method with '@RequestMapping' or '@ResponseStatus' annotation that belongs to a Java Class with '@RestController' or '@ControllerAdvice' annotation respectively in the Source Code File of the microservice's project.
88	[1]	Source Code File	Service Interface	Service Operation	A Service Operation concept is indicated by a Java Method with '@Scheduled' annotation in the Source Code File of the microservice's project.
89	[1]	Source Code File	Service Operation	Data Store	A Data Store to a Service Operation is indicated by a Java Method 'save()' or 'findById()' or any Java Method that starts with 'find' of a Java Interface that extends 'CrudRepository' Java Interface of package 'org.springframework.data.repository' in the Source Code File of the microservice's project.
90	[1]	Source Code File	Service Operation	Circuit Breaker	A Circuit Breaker to a Service Operation is indicated by a Java Method with '@RequestMapping' annotation that belongs to a Java Interface with '@FeignClient' annotation in the Source Code File of the microservice's project.
91	[1]	Source Code File	Microservice	Service Dependency	A 'Consul' Configuration provider to a Microservice is indicated by a Java Class with '@RefreshScope' annotation in the Source Code File of the microservice's project.
92	[1]	Container Build File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the JAR application name argument of 'ADD' command in the Container Build File of the microservice's project.

93	[1]	Container Build File	Microservice	Container	The name of Container concept is indicated by the JAR application name argument of 'ADD' command in the Container Build File of the microservice's project.
94	[1]	Container Build File	Microservice	Service Interface	The server path of Service Interface concept is indicated by the JAR application name argument of 'ADD' command in the Container Build File of the microservice's project.
95	[1]	Container Build File	Infrastructure Microservice	-	A 'MongoDB Data Store' Infrastructure Microservice concept is indicated by a 'FROM' command with argument value that starts with 'mongo:' in the Container Build File of the microservice's project.
96	[1]	Container Build File	Log Analysis	-	A Log Analysis concept is indicated by a 'FROM' command with argument value that starts with 'logstash:' in the Container Build File of the microservice's project.
97	[1]	Container Orchestration File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.
98	[1]	Container Orchestration File	Microservice	Container	The name of Container concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.
99	[1]	Container Orchestration File	Microservice	Service Interface	The server path of Service Interface concept is indicated by the key name of service container definition in the Container Orchestration File of the application's project.
100	[1]	Container Orchestration File	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the service container name of 'depends_on' or 'links' key in the Container Orchestration File of the application's project.
101	[1]	Container Orchestration File	Infrastructure Microservice	-	An Infrastructure Microservice concept is indicated by a service container definition that does not have 'build' key in the Container Orchestration File of the application's project.
102	[1]	Container Orchestration File	Infrastructure Microservice	-	A 'RabbitMQ' Infrastructure Microservice concept is indicated by an 'image:' key with value that starts wth 'rabbitmq:' in the Container Orchestration File of the application's project.
103	[1]	Container Orchestration File	Log Analysis	-	A Log Analysis concept is indicated by an 'image:' key with value that starts wth 'elasticsearch:' or 'kibana:' in the Container Orchestration File of the application's project.
104	[1]	Container Orchestration File	Log Analysis	Service Dependency	A Microservice provider to a Log Analysis is indicated by the key name of service container definition that has 'logging' or 'log_opt' key in the Container Orchestration File of the application's project.

105	[2]	Build File	Microservice	Service Dependency	A 'MongoDB' Infrastructure Microservice provider to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.boot:spring-boot-starter-data-mongodb' in the Build File of the microservice's project.
106	[2]	Source Code File	Microservice	Service Dependency	A 'MongoDB' Infrastructure Microservice provider to a Microservice is indicated by a Java Method 'save()' or 'findById()' or any Java Method that starts with 'find' of a Java Interface that extends 'MongoRepository' Java Interface of package 'org.springframework.data.mongodb.repository' in the Source Code File of the microservice's project.
107	[2]	Source Code File	Service Operation	Data Store	A Data Store to a Service Operation is indicated by a Java Method 'save()' or 'findById()' or any Java Method that starts with 'find' of a Java Interface that extends 'MongoRepository' Java Interface of package 'org.springframework.data.mongodb.repository' in the Source Code File of the microservice's project.
108	[2]	Source Code File	Microservice	Service Dependency	A 'MongoDB' Infrastructure Microservice provider to a Microservice is indicated by a Java Interface that extends 'MongoRepository' Java Interface of package 'org.springframework.data.mongodb.repository' in the Source Code File of the microservice's project.
109	[2]	Container Orchestration File	Infrastructure Microservice	-	A 'MongoDB Data Store' Infrastructure Microservice concept is indicated by an 'image:' key with value that starts wth 'mongo' in the Container Orchestration File of the application's project.
110	[2]	Container Orchestration File	Infrastructure Microservice	-	An 'HAProxy Load Balancer' Infrastructure Microservice concept is indicated by an 'image:' key with value that starts wth 'tutum/haproxy' in the Container Orchestration File of the application's project.
111	[2]	Container Orchestration File	Microservice	Load Balancer	An 'HAProxy' Load Balancer to a Microservice is indicated by an 'HAProxy Load Balancer' Infrastructure Microservice that has a 'links' or 'depends_on' key that points to it in the Container Orchestration File of the application's project.
112	[3]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-eureka' in the Build File of the microservice's project.
113	[3]	Build File	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value

					'spring-cloud-starter-eureka' in the Build File of the microservice's project.
114	[3]	Build File	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-eureka' in the Build File of the microservice's project.
115	[3]	Build File	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-config-client' in the Build File of the microservice's project.
116	[3]	Build File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-hystrix' in the Build File of the microservice's project.
117	[3]	Build File	Microservice	Service Dependency	An 'PostgreSQL' Infrastructure Microservice provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'postgresql' in the Build File of the microservice's project.
118	[3]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-eureka-server' in the Build File of the microservice's project.
119	[3]	Build File	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-eureka-server' in the Build File of the microservice's project.
120	[3]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-zuul' in the Build File of the microservice's project.
121	[3]	Build File	API Gateway	-	An API Gateway concept with technology of 'Netflix Zuul' is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-zuul' in the Build File of the microservice's project.

122	[3]	Build File	Microservice	Service Dependency	A 'Kafka' Infrastructure Microservice provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-stream-kafka' in the Build File of the microservice's project.
123	[3]	Build File	Microservice	Service Dependency	A 'Redis' Infrastructure Microservice provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-data-redis' or 'jedis' in the Build File of the microservice's project.
124	[3]	Build File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is is indicated by a <project><dependencies><dependency><artifactId> key with value 'hystrix-javanica' in the Build File of the microservice's project.
125	[3]	Configurations File	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by the property 'spring.cloud.config.enabled: true' in the Configurations File of the microservice's project.
126	[3]	Configurations File	Microservice	Service Dependency	A 'PostgreSQL' Infrastructure Microservice provider to a Microservice is indicated by the property 'spring.datasource.url:' and a value that starts with 'jdbc:postgresql://' in the Configurations File of the microservice's project.
127	[3]	Configurations File	Microservice	Service Dependency	A 'PostgreSQL' Infrastructure Microservice provider to a Microservice is indicated by the property 'spring.database.driverClassName:' and a value 'org.postgresql.Driver' in the Configurations File of the microservice's project.
128	[3]	Configurations File	Configuration	-	A 'Spring Cloud Config' Configuration concept is indicated by the property 'spring.cloud.config.server.git.uri:' and/or 'spring.cloud.config.server.git.searchPaths:' in the Configurations File of the microservice's project.
129	[3]	Configurations File	Service Interface	Endpoint	A prefix to an Endpoint is indicated by the value of the property 'zuul.prefix:' in the Configurations File of the microservice's project.
130	[3]	Configurations File	Service Interface	Endpoint	An Endpoint to Service Interface is indicated by the value of the property that starts with 'zuul.routes.' and ends with the microservice name in the Configurations File of the microservice's project.
131	[3]	Configurations File	API Gateway	Service Dependency	A Microservice provider to a 'Netflix Zuul' API Gateway is indicated by the property that starts with 'zuul.routes.' and ends with the microservice name in the Configurations File of the microservice's project.

132	[3]	Configurations File	Microservice	Service Dependency	A 'Redis' Infrastructure Microservice provider to a Microservice is indicated by the value of the property 'redis.server' in the Configurations File of the microservice's project.
133	[3]	Configurations File	Microservice	Service Dependency	A 'Kafka' Infrastructure Microservice provider to a Microservice is indicated by the value of the property that starts with 'spring.cloud.stream.bindings.kafka.binder.brokers' in the Configurations File of the microservice's project.
134	[3]	Source Code File	Microservice	Service Dependency	A Registry and Discovery provider to a Microservice is indicated by a Java Class with '@EnableEurekaClient' annotation in the Source Code File of the microservice's project.
135	[3]	Source Code File	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by a Java Class with '@EnableEurekaClient' annotation in the Source Code File of the microservice's project.
136	[3]	Source Code File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a Java Class with '@EnableEurekaClient' annotation in the Source Code File of the microservice's project.
137	[3]	Source Code File	Microservice	Service Dependency	A 'PostgreSQL' Infrastructure Microservice provider to a Microservice is indicated by a Java Class with '@Entity' annotation in the Source Code File of the microservice's project.
138	[3]	Source Code File	Microservice	Service Dependency	A 'PostgreSQL' Infrastructure Microservice provider to a Microservice is indicated by a Java Class with '@Table' annotation in the Source Code File of the microservice's project.
139	[3]	Source Code File	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the URL in the first argument of a Java Method 'exchange()' of a Java Interface 'RestTemplate' of package 'org.springframework.web.client' in the Source Code File of the microservice's project.
140	[3]	Source Code File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a Java Method with '@LoadBalanced' annotation in the Source Code File of the microservice's project.
141	[3]	Source Code File	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the URL in the first argument of a Java Method 'exchange()' of a Java Interface 'OAuth2RestTemplate' of package 'org.springframework.security.oauth2.client' in the Source Code File of the microservice's project.
142	[3]	Source Code File	Microservice	Service Dependency	A 'Spring Cloud OAuth2' Security provider to a Microservice is indicated by a Java Method 'exchange()' of a Java Interface 'OAuth2RestTemplate' of package

					'org.springframework.security.oauth2.client' in the Source Code File of the microservice's project.
143	[3]	Source Code File	Service Operation	Message Bus	A Message Bus to a Service Operation reader is indicated by a Java Method with '@StreamListener' annotation that belongs to a Java Class with '@EnableBinding' annotation in the Source Code File of the microservice's project.
144	[3]	Source Code File	Service Operation	Cash Store	A 'Redis' Cash Store to a Service Operation reader is indicated by a Java Method 'put()' of a Java Interface 'HashOperations' of package 'org.springframework.data.redis.core' in the Source Code File of the microservice's project.
145	[3]	Source Code File	Service Operation	Cash Store	A 'Redis' Cash Store to a Service Operation reader is indicated by a Java Method 'get()' of a Java Interface 'HashOperations' of package 'org.springframework.data.redis.core' in the Source Code File of the microservice's project.
146	[3]	Source Code File	Service Operation	Cash Store	A 'Redis' Cash Store to a Service Operation writer is indicated by a Java Method 'put()' or 'delete()' of a Java Interface 'HashOperations' of package 'org.springframework.data.redis.core' in the Source Code File of the microservice's project.
147	[3]	Source Code File	Service Operation	Cash Store	A 'Redis' Cash Store to a Service Operation writer is indicated by a Java Method 'opsForHash()' of a Java Class 'RedisTemplate' of package 'org.springframework.data.redis.core' in the Source Code File of the microservice's project.
148	[3]	Source Code File	Microservice	Service Dependency	A Registry and Discovery provider to a Microservice is indicated by a Java Method 'getServices()' or 'getInstances()' of a Java Interface 'DiscoveryClient' of package 'org.springframework.cloud.client.discovery' in the Source Code File of the microservice's project.
149	[3]	Source Code File	Service Operation	Circuit Breaker	A 'Netflix Hystrix' Circuit Breaker to a Service Operation is indicated by a Java Method with '@HystrixCommand' annotation in the Source Code File of the microservice's project.
150	[3]	Source Code File	Service Operation	Message Bus	A Message Bus to a Service Operation writer is indicated by a Java Method 'output()' of a Java Interface 'Source' of package 'org.springframework.cloud.stream.messaging' in the Source Code File of the microservice's project.
151	[3]	Source Code File	Microservice	Service Dependency	A 'PostgreSQL' Infrastructure Microservice provider to a Microservice is indicated by a Java Method 'save()' or 'findById()' or any Java Method that starts with 'find' of a Java Interface that extends 'CrudRepository' Java Interface of package

					'org.springframework.data.repository' in the Source Code File of the microservice's project.
152	[3]	Source Code File	Microservice	Service Dependency	A 'PostgreSQL' Infrastructure Microservice provider to a Microservice is indicated by a Java Interface that extends 'CrudRepository' Java Interface of package 'org.springframework.data.repository' in the Source Code File of the microservice's project.
153	[3]	Container Orchestration File	Infrastructure Microservice	-	A 'Kafka' Infrastructure Microservice concept is indicated by an 'image:' key with value that contains 'spotify/kafka' in the Container Orchestration File of the application's project.
154	[4]	Build File	Microservice	Service Dependency	A 'Consul' Registry and Discovery provider to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-consul-discovery' in the Build File of the microservice's project.
155	[4]	Build File	Registry and Discovery	Service Dependency	A Microservice provider to a 'Consul' Registry and Discovery is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-consul-discovery' in the Build File of the microservice's project.
156	[4]	Build File	Tracing	Service Dependency	A Microservice provider to a 'Zipkin' Tracing is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-zipkin' in the Build File of the microservice's project.
157	[4]	Build File	Log Analysis	Service Dependency	A Microservice provider to a 'Logstash' Log Analysis is indicated by a <project><dependencies><dependency><artifactId> key with value 'logstash-logback-encoder' in the Build File of the microservice's project.
158	[4]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a <project><dependencies><dependency><artifactId> key with value 'spring-cloud-starter-netflix-ribbon' in the Build File of the microservice's project.
159	[4]	Build File	Monitoring	Service Dependency	A Microservice provider to a 'Prometheus' Monitoring is indicated by a <project><dependencies><dependency><artifactId> key with value 'micrometer-registry-prometheus' in the Build File of the microservice's project.
160	[4]	Configurations File	Tracing	Service Dependency	A Microservice provider to a 'Zipkin' Tracing is indicated by non-zero value of the property 'spring.sleuth.sampler.percentage' in the Configurations File of the microservice's project.

161	[4]	Configurations File	Tracing	Service Dependency	A Microservice provider to a 'Zipkin' Tracing is indicated by the property 'spring.zipkin.enabled: true' in the Configurations File of the microservice's project.
162	[4]	Configurations File	Microservice	Service Dependency	A 'Consul' Registry and Discovery provider to a Microservice is indicated by the property 'spring.cloud.consul.host' or starts with 'spring.cloud.consul.discovery' in the Configurations File of the microservice's project.
163	[4]	Configurations File	Registry and Discovery	Service Dependency	A Microservice provider to a 'Consul' Registry and Discovery is indicated by the property 'spring.cloud.consul.host' or starts with 'spring.cloud.consul.discovery' in the Configurations File of the microservice's project.
164	[4]	Configurations File	Monitoring	Service Dependency	A Microservice provider to a 'Prometheus' Monitoring is indicated by the property 'management.endpoint.prometheus.enabled: true' in the Configurations File of the microservice's project.
165	[4]	Configurations File	Service Interface	Endpoint	The 'GET /prometheus' path of Endpoint concept is indicated by the property 'management.endpoint.prometheus.enabled: true' in the Configurations File of the microservice's project.
166	[4]	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'spring.cloud.consul.ribbon.enabled: true' in the Configurations File of the microservice's project.
167	[4]	Source Code File	Service Interface	Endpoint	The path of Endpoint concept is indicated by the value of 'method' parameter and 'value' or 'path' parameter in '@RequestMapping' Java Method annotation that belongs to a Java Class with '@Controller' annotation in the Source Code File of the microservice's project.
168	[4]	Source Code File	Service Interface	Service Operation	A Service Operation concept is indicated by a Java Method with '@RequestMapping' annotation that belongs to a Java Class with '@Controller' annotation respectively in the Source Code File of the microservice's project.
169	[4]	Source Code File	Service Operation	Data Store	A Data Store to a Service Operation is indicated by a Java Method 'save()' or 'findById()' or any Java Method that starts with 'find' of a Java Interface that extends 'PagingAndSortingRepository' Java Interface of package 'org.springframework.data.repository' in the Source Code File of the microservice's project.
170	[4]	Source Code File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a Java Class with '@RibbonClient' annotation in the Source Code File of the microservice's project.
171	[4]	Source Code File	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the URL in the first argument of a Java Method 'getForObject()' or

					'getForEntity()' of a Java Interface 'RestTemplate' of package 'org.springframework.web.client' in the Source Code File of the microservice's project.
172	[4]	Source Code File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a Java Method 'choose()' of a Java Interface 'LoadBalancerClient' of package 'org.springframework.cloud.client.loadbalancer' in the Source Code File of the microservice's project.
173	[4]	Source Code File	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the value of the first argument of a Java Method 'choose()' of a Java Interface 'LoadBalancerClient' of package 'org.springframework.cloud.client.loadbalancer' in the Source Code File of the microservice's project.
174	[4]	Source Code File	Log Analysis	Service Dependency	A Microservice provider to a Log Analysis is indicated by a Java Method 'trace()' of a Java Class 'Logger' of package 'org.slf4j' in the Source Code File of the microservice's project.
175	[4]	Container Build File	API Gateway	-	A 'Apache HTTP' API Gateway concept is indicated by a 'RUN' command with argument value that contains 'apache2', 'proxy_http' and 'proxy_balancer' in the Container Build File of the microservice's project.
176	[4]	Container Build File	Microservice	Load Balancer	A 'Apache HTTP' Load Balancer to a Microservice is indicated by a 'RUN' command with argument value that contains 'apache2' and 'proxy_balancer' in the Container Build File of the microservice's project.
177	[4]	Container Build File	Log Analysis	-	A Log Analysis concept is indicated by a 'FROM' command with argument value that starts with 'docker.elastic.co/elasticsearch/elasticsearch:' or 'docker.elastic.co/beats/filebeat:' in the Container Build File of the microservice's project.
178	[4]	Container Build File	Monitoring	-	A 'Prometheus' Monitoring concept is indicated by a 'FROM' command with argument value that starts with 'prom/prometheus:' in the Container Build File of the microservice's project.
179	[4]	Container Orchestration File	Registry and Discovery	-	A 'Consul' Registry and Discovery concept is indicated by an 'image:' key with value that starts wth 'consul:' in the Container Orchestration File of the application's project.
180	[4]	Container Orchestration File	Configuration	-	A 'Consul' Configuration concept is indicated by an 'image:' key with value that starts wth 'consul:' in the Container Orchestration File of the application's project.

181	[4]	Container Orchestration File	Tracing	-	A 'Zipkin' Tracing concept is indicated by an 'image:' key with value that starts wth 'openzipkin/zipkin:' in the Container Orchestration File of the application's project.
182	[4]	Container Orchestration File	Log Analysis	-	A 'Kibana' Log Analysis concept is indicated by an 'image:' key with value that starts wth 'docker.elastic.co/kibana/kibana:' in the Container Orchestration File of the application's project.
183	[5]	Build File	Microservice Architecture	-	The name of Microservice Architecture concept is indicated by the value of 'rootProject.name' Gradle command in the Build File of the application's project.
184	[5]	Build File	Microservice Architecture	Microservice	The name of Microservice concept is indicated by the value of 'include' Gradle command in the Build File of the application's project.
185	[5]	Build File	Microservice	Service Dependency	A 'Consul' Registry and Discovery provider to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-consul-all' in the Build File of the microservice's project.
186	[5]	Build File	Registry and Discovery	Service Dependency	A Microservice provider to a 'Consul' Registry and Discovery is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-consul-all' in the Build File of the microservice's project.
187	[5]	Build File	Microservice	Service Dependency	A 'Consul' Configuration provider to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-consul-all' in the Build File of the microservice's project.
188	[5]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-zuul' in the Build File of the microservice's project.
189	[5]	Build File	API Gateway	-	An API Gateway concept with technology of 'Netflix Zuul' is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-zuul' in the Build File of the microservice's project.
190	[5]	Build File	Tracing	Service Dependency	A Microservice provider to a 'Zipkin' Tracing is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-zipkin' in the Build File of the microservice's project.
191	[5]	Build File	Monitoring	-	A 'Netflix Hystrix Dashboard' Monitoring is indicated by a 'compile' Gradle command with an argument

					'org.springframework.cloud:spring-cloud-starter-hystrix-dashboard' in the Configurations File of the microservice's project.
192	[5]	Build File	Microservice	Service Dependency	A 'Netflix Hystrix Dashboard' Monitoring is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-hystrix-dashboard' in the Build File of the microservice's project.
193	[5]	Build File	Monitoring	-	A 'Netflix Turbine' Monitoring is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-netflix-turbine' in the Build File of the microservice's project.
194	[5]	Build File	Monitoring	Service Dependency	A 'Netflix Turbine' Monitoring provider to a 'Netflix Hystrix Dashboard' Monitoring is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-netflix-turbine' in the Build File of the microservice's project.
195	[5]	Build File	Monitoring	-	A 'Spring Boot Admin' Monitoring is indicated by a 'compile' Gradle command with an argument 'de.codecentric:spring-boot-admin-server' in the Build File of the microservice's project.
196	[5]	Build File	Monitoring	Microservice	A Microservice provider to a 'Spring Boot Admin' Monitoring is indicated by a 'compile' Gradle command with an argument 'org.springframework.boot:spring-boot-starter-actuator' in the Build File of the microservice's project.
197	[5]	Build File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-hystrix' in the Build File of the microservice's project.
198	[5]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-ribbon' in the Build File of the microservice's project.
199	[5]	Configurations File	Tracing	Service Dependency	A Microservice provider to a 'Zipkin' Tracing is indicated by the URL value of the property 'spring.zipkin.baseUrl:' in the Configurations File of the microservice's project.
200	[5]	Configurations File	Monitoring	-	A 'Netflix Turbine' Monitoring is indicated by the property 'turbine.appConfig' in the Build File of the microservice's project.
201	[5]	Configurations File	Monitoring	Service Dependency	A 'Netflix Turbine' Monitoring provider to a 'Netflix Hystrix Dashboard' Monitoring is indicated by the property 'turbine.appConfig' in the Build File of the microservice's project.

202	[5]	Configurations File	Monitoring	Service Dependency	A Microservice provider to 'Netflix Turbine' Monitoring provider is indicated by the value of the property 'turbine.appConfig' in the Configurations File of the microservice's project.
203	[5]	Configurations File	Monitoring	-	A 'Spring Boot Admin' Monitoring is indicated by a property that starts with 'spring.boot.admin.discovery' in the Configurations File of the microservice's project.
204	[5]	Configurations File	Microservice	Service Dependency	A 'Consul' Configuration provider to a Microservice is indicated by the property 'spring.cloud.consul.config.enabled: true' in the Configurations File of the microservice's project.
205	[5]	Source Code File	Monitoring	-	A 'Netflix Turbine' Monitoring is indicated by a Java Class with '@EnableTurbine' annotation in the Source Code File of the microservice's project.
206	[5]	Source Code File	Monitoring	Service Dependency	A 'Netflix Turbine' Monitoring provider to a 'Netflix Hystrix Dashboard' Monitoring is indicated by a Java Class with '@EnableTurbine' annotation in the Source Code File of the microservice's project.
207	[5]	Source Code File	Monitoring	-	A 'Spring Boot Admin' Monitoring is indicated by a Java Class with '@EnableAdminServer' annotation in the Source Code File of the microservice's project.
208	[5]	Source Code File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by a Java Class with '@EnableHystrix' annotation in the Source Code File of the microservice's project.
209	[6]	Build File	Monitoring	-	A 'Spring Boot Admin' Monitoring is indicated by a 'compile' Gradle command with an argument 'de.codecentric:spring-boot-admin-starter-server' in the Build File of the microservice's project.
210	[6]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client' in the Build File of the microservice's project.
211	[6]	Build File	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client' in the Build File of the microservice's project.
212	[6]	Build File	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-client' in the Build File of the microservice's project.

213	[6]	Build File	Configuration	-	A 'Spring Cloud Config' Configuration concept is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-config-server' or 'org.springframework.cloud:spring-cloud-config-monitor' in the Build File of the microservice's project.
214	[6]	Build File	Microservice	Service Dependency	A 'RabbitMQ' Infrastructure Microservice provider to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-bus-amqp' in the Build File of the microservice's project.
215	[6]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-server' in the Build File of the microservice's project.
216	[6]	Build File	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-netflix-eureka-server' in the Build File of the microservice's project.
217	[6]	Build File	Monitoring	-	A 'Netflix Hystrix Dashboard' Monitoring is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-netflix-hystrix-dashboard' in the Configurations File of the microservice's project.
218	[6]	Build File	Microservice	Service Dependency	A 'Netflix Hystrix Dashboard' Monitoring is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-netflix-hystrix-dashboard' in the Build File of the microservice's project.
219	[6]	Build File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-netflix-hystrix' in the Build File of the microservice's project.
220	[6]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-netflix-ribbon' in the Build File of the microservice's project.
221	[6]	Build File	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-config' in the Build File of the microservice's project.

222	[6]	Configurations File	Service Interface	Endpoint	The 'POST /shutdown' path of Endpoint concept is indicated by the property 'management.endpoint.shutdown.enabled: true' in the Configurations File of the microservice's project.
223	[6]	Configurations File	Service Interface	Endpoint	The 'GET /health' path of Endpoint concept is indicated by the non-never value of property 'management.endpoint.health.showDetails:' in the Configurations File of the microservice's project.
224	[6]	Configurations File	Tracing	Service Dependency	A Microservice provider to a 'Zipkin' Tracing is indicated by the property 'spring.zipkin.sender.type:' in the Configurations File of the microservice's project.
225	[6]	Configurations File	Microservice	Service Dependency	A 'Spring Cloud Config' Configuration provider to a Microservice is indicated by the property 'spring.cloud.config.discovery.enabled: true' or the value of 'spring.cloud.config.discovery.serviceId:' in the Configurations File of the microservice's project.
226	[7]	Build File	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-eureka' in the Build File of the microservice's project.
227	[7]	Build File	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-eureka' in the Build File of the microservice's project.
228	[7]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-eureka-server' in the Build File of the microservice's project.
229	[7]	Build File	Registry and Discovery	-	A Registry and Discovery concept with technology of 'Netflix Eureka' is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-eureka-server' in the Build File of the microservice's project.
230	[7]	Build File	API Gateway	-	A 'Netflix Sidecar' API Gateway is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-netflix-sidecar' in the Build File of the microservice's project.
231	[7]	Build File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-netflix-sidecar' in the Build File of the microservice's project.
232	[7]	Build File	Microservice	Service Dependency	A Microservice provider to a Registry and Discovery is indicated by a 'compile' Gradle command with an argument

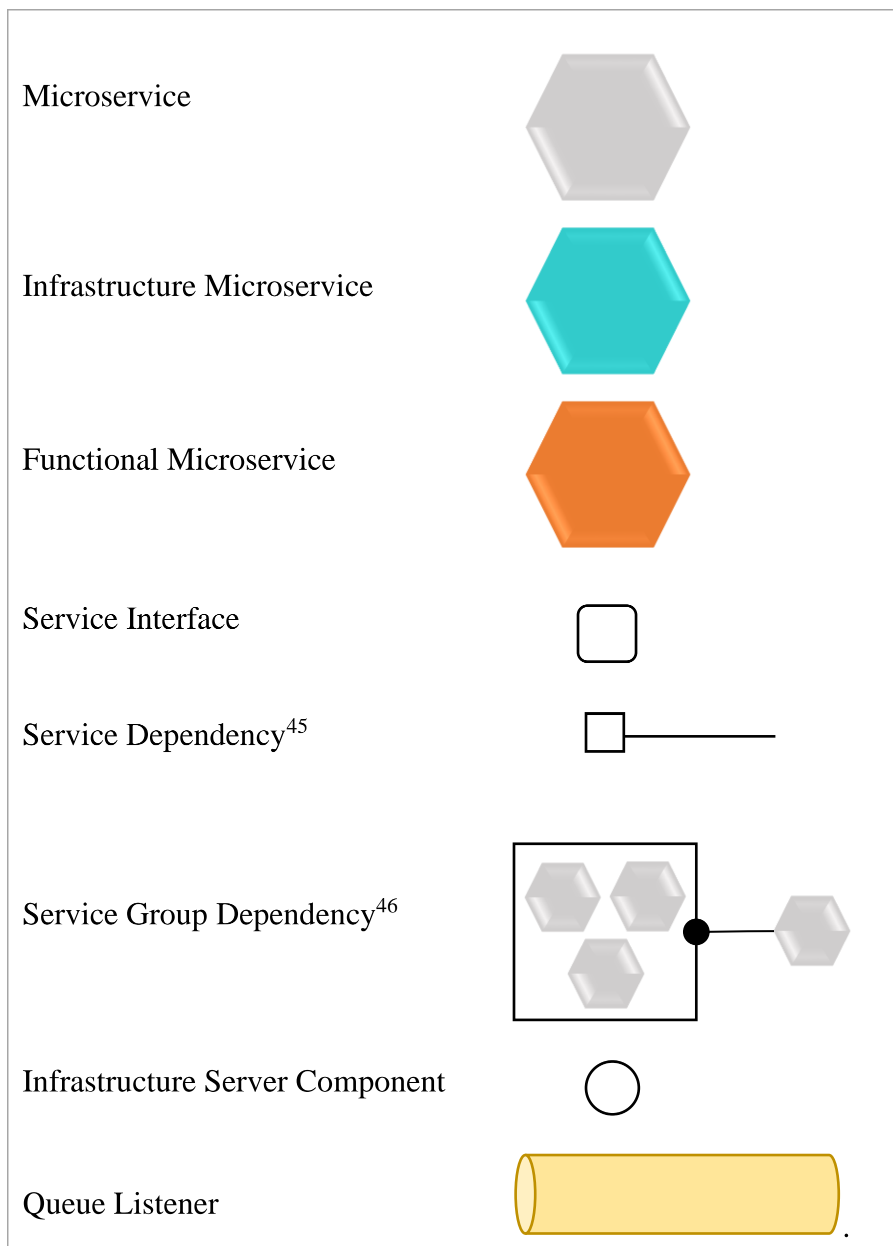
					'org.springframework.cloud:spring-cloud-netflix-sidecar' in the Build File of the microservice's project.
233	[7]	Build File	Registry and Discovery	Service Dependency	A Microservice provider to a Registry and Discovery is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-netflix-sidecar' in the Build File of the microservice's project.
234	[7]	Build File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-netflix-sidecar' in the Build File of the microservice's project.
235	[7]	Configurations File	Microservice	Service Dependency	A 'Netflix Eureka' Registry and Discovery provider to a Microservice is indicated by the property 'eureka.instance.healthCheckUrlPath:' in the Configurations File of the microservice's project.
236	[7]	Configurations File	Registry and Discovery	Service Dependency	A Microservice provider to a 'Netflix Eureka' Registry and Discovery is indicated by the property 'eureka.instance.healthCheckUrlPath:' in the Configurations File of the microservice's project.
237	[7]	Configurations File	Service Interface	Endpoint	The 'GET' path of Endpoint concept is indicated by the value of property 'eureka.instance.healthCheckUrlPath:' in the Configurations File of the microservice's project.
238	[7]	Configurations File	API Gateway	-	A 'Netflix Sidecar' API Gateway is indicated by the property 'sidecar.port:' and/or 'sidecar.healthUri:' in the Configurations File of the microservice's project.
239	[7]	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'sidecar.port:' and/or 'sidecar.healthUri:' in the Configurations File of the microservice's project.
240	[7]	Configurations File	Microservice	Service Dependency	A Microservice provider to a Registry and Discovery is indicated by the property 'sidecar.port:' and/or 'sidecar.healthUri:' in the Configurations File of the microservice's project.
241	[7]	Configurations File	Registry and Discovery	Service Dependency	A Microservice provider to a Registry and Discovery is indicated by the property 'sidecar.port:' and/or 'sidecar.healthUri:' in the Configurations File of the microservice's project.
242	[7]	Configurations File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by the property 'sidecar.port:' and/or 'sidecar.healthUri:' in the Configurations File of the microservice's project.

243	[7]	Configurations File	API Gateway	Service Dependency	A Microservice provider to a 'Netflix Sidecar' API Gateway is indicated by the value of the property 'sidecar.port:' and/or 'sidecar.healthUri:' in the Configurations File of the microservice's project.
244	[7]	Configurations File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by the property 'ribbon.eureka.enabled: true' in the Configurations File of the microservice's project.
245	[7]	Source Code File	API Gateway	-	A 'Netflix Sidecar' API Gateway is indicated by a Java Class with '@EnableSidecar' annotation in the Source Code File of the microservice's project.
246	[7]	Source Code File	Microservice	Load Balancer	A 'Netflix Ribbon' Load Balancer to a Microservice is indicated by a Java Class with '@EnableSidecar' annotation in the Source Code File of the microservice's project.
247	[7]	Source Code File	Microservice	Service Dependency	A Microservice provider to a Registry and Discovery is indicated by a Java Class with '@EnableSidecar' annotation in the Source Code File of the microservice's project.
248	[7]	Source Code File	Registry and Discovery	Service Dependency	A Microservice provider to a Registry and Discovery is indicated by a Java Class with '@EnableSidecar' annotation in the Source Code File of the microservice's project.
249	[7]	Source Code File	Monitoring	Service Dependency	A Microservice provider to a 'Netflix Hystrix Dashboard' or 'Netflix Turbine' Monitoring is indicated by a Java Class with '@EnableSidecar' annotation in the Source Code File of the microservice's project.
250	[8]	Build File	Log Analysis	Service Dependency	A Microservice provider to a 'Logstash' Log Analysis is indicated by a 'compile' Gradle command with an argument 'net.logstash.logback:logstash-logback-encoder' in the Build File of the microservice's project.
251	[8]	Build File	Tracing	Service Dependency	A Microservice provider to a 'Zipkin' Tracing is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-sleuth' or 'org.springframework.cloud:spring-cloud-sleuth-stream' in the Build File of the microservice's project.
252	[8]	Build File	Microservice	Service Dependency	A 'RabbitMQ' Infrastructure Microservice provider to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-sleuth-stream' in the Build File of the microservice's project.
253	[8]	Build File	Monitoring	-	A 'Netflix Turbine' Monitoring is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-

					cloud-starter-turbine-amqp' in the Build File of the microservice's project.
254	[8]	Build File	Monitoring	Service Dependency	A 'Netflix Turbine' Monitoring provider to a 'Netflix Hystrix Dashboard' Monitoring is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-turbine-amqp' in the Build File of the microservice's project.
255	[8]	Build File	Microservice	Service Dependency	A 'RabbitMQ' Infrastructure Microservice provider to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-starter-turbine-amqp' in the Build File of the microservice's project.
256	[8]	Build File	Tracing	-	A 'Zipkin' Tracing is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-sleuth-zipkin-stream' in the Build File of the microservice's project.
257	[8]	Build File	Tracing	-	A 'Zipkin' Tracing is indicated by a 'runtime' Gradle command with an argument 'io.zipkin.java:zipkin-autoconfigure-ui' in the Build File of the microservice's project.
258	[8]	Build File	Microservice	Service Dependency	A 'RabbitMQ' Infrastructure Microservice provider to a Microservice is indicated by a 'compile' Gradle command with an argument 'org.springframework.cloud:spring-cloud-stream-binder-rabbit' in the Build File of the microservice's project.
259	[8]	Configurations File	Service Interface	Endpoint	The 'POST /shutdown' path of Endpoint concept is indicated by the property 'endpoints.shutdown.enabled: true' in the Configurations File of the microservice's project.
260	[8]	Configurations File	Service Interface	Endpoint	The 'POST /restart' path of Endpoint concept is indicated by the property 'endpoints.restart.enabled: true' in the Configurations File of the microservice's project.
261	[8]	Configurations File	Monitoring	-	A 'Netflix Turbine' Monitoring is indicated by the property 'turbine.amqp.port:' in the Configurations File of the microservice's project.
262	[8]	Configurations File	Monitoring	Service Dependency	A 'Netflix Turbine' Monitoring provider to a 'Netflix Hystrix Dashboard' Monitoring is indicated by the property 'turbine.amqp.port:' in the Configurations File of the microservice's project.
263	[8]	Configurations File	Monitoring	Service Dependency	A Microservice provider to 'Netflix Turbine' Monitoring provider is indicated by the property 'turbine.amqp.port:' in the Configurations File of the microservice's project.

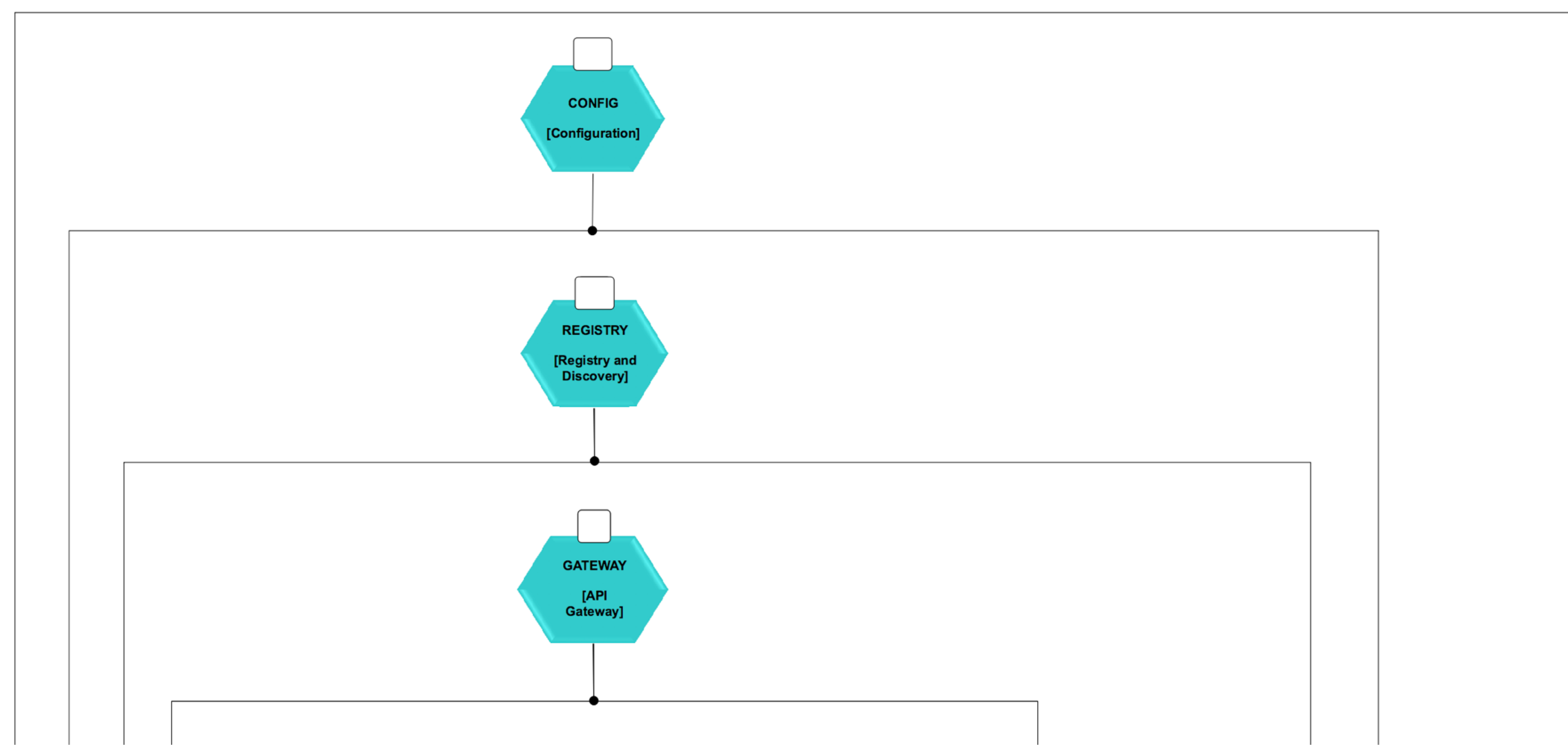
264	[8]	Configurations File	Microservice	Service Dependency	A 'RabbitMQ' Infrastructure Microservice provider to a Microservice is indicated by the property 'turbine.amqp.port:' in the Configurations File of the microservice's project.
265	[8]	Source Code File	Tracing	-	A 'Zipkin' Tracing is indicated by a Java Class with '@EnableZipkinStreamServer' annotation in the Source Code File of the microservice's project.
266	[8]	Source Code File	Microservice	Service Dependency	A 'RabbitMQ' Infrastructure Microservice provider to a Microservice is indicated by a Java Class with '@EnableZipkinStreamServer' annotation in the Source Code File of the microservice's project.
267	[8]	Source Code File	Microservice	Service Dependency	A Microservice provider to a Microservice is indicated by the URL in the first argument of a Java Method 'getForEntity()' of a Java Interface 'RestOperations' of package 'org.springframework.web.client' in the Source Code File of the microservice's project.
268	[8]	Container Orchestration File	Log Analysis	-	A 'Kibana' Log Analysis concept is indicated by an 'image:' key with value that starts wth 'docker.elastic.co/elasticsearch/elasticsearch:' or 'docker.elastic.co/logstash/logstash:' in the Container Orchestration File of the application's project.

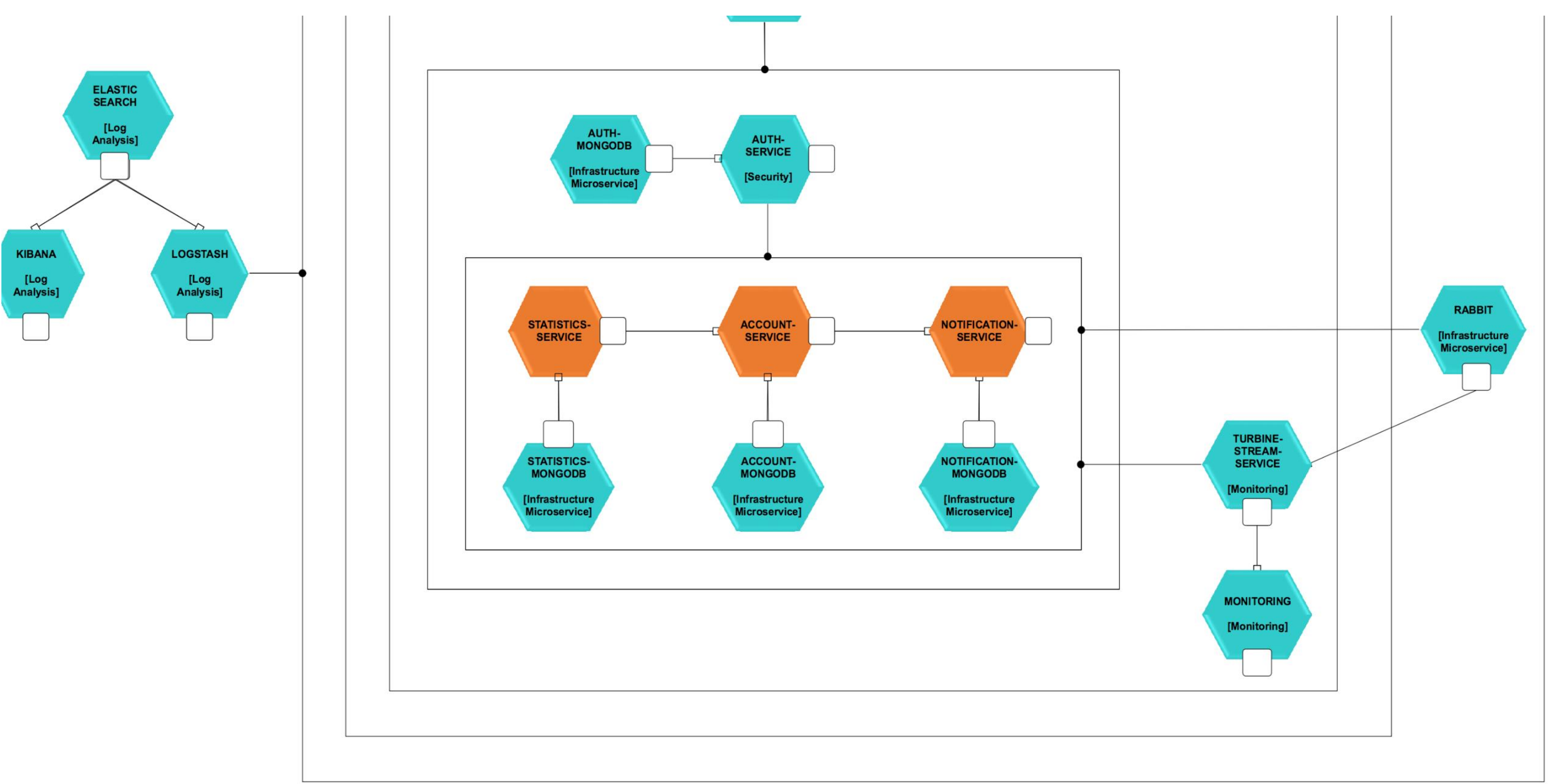
Appendix-B: 1- Graphical Notations of Architecture Diagram and their mappings to PIM metaclasses



[45] The arrow square head represents the consumer.
[46] It represents the interaction between a group of microservices and one microservice.

Appendix-B: 2- Architecture diagram recovered of case study 1 (PiggyMetrics)





ELASTIC SEARCH
[Log Analysis]
KIBANA
[Log Analysis]
LOGSTASH
[Log Analysis]
AUTH-MONGODB
[Infrastructure Microservice]
AUTH-SERVICE
[Security]
STATISTICS-SERVICE
ACCOUNT-SERVICE
NOTIFICATION-SERVICE
STATISTICS-MONGODB
[Infrastructure Microservice]
ACCOUNT-MONGODB
[Infrastructure Microservice]
NOTIFICATION-MONGODB
[Infrastructure Microservice]
RABBIT
[Infrastructure Microservice]
TURBINE-STREAM-SERVICE
[Monitoring]
MONITORING
[Monitoring]

Appendix-B: 3- Elaboration on manual recovery and visualization steps for PiggyMetrics

Step-1 Artefact collection

The artefacts collected from the project are pulled form its public GitHub repository. PiggyMetrics has two distributed GitHub repositories one for the main application[47] and anotherr for ELK stack[48].

Step-2 Sort All Mapping Rules by the following Artefact Type order:

- *GitHub Repository*
- *Container Orchestration File*
- *Container Build File*
- *Build File*
- *Configurations File*
- *Source Code File*

Step-3 Check All Mapping Rules of GitHub Repository

Usually it is one repository but more than one is also possible.

Step-4 Check All Mapping Rules of Build File of Application's Project

Usually it is one file.

One POM.XML for Maven application.

One SETTINGS.GRADLE for Gradle application.

Step-5 Check All Mapping Rules of Container Orchestration File of Application's Project

Check all DOCKER COMPOSE files.

For Each Container in Docker Compose File:

Step-6 Check All Mapping Rules of Build File of Microservice's Project

Usually it is one file.

One POM.XML for Maven application.

One BUILD.GRADLE for Gradle application.

[47] https://github.com/sqshq/piggymetrics
[48] https://github.com/sqshq/ELK-docker

Step-7 Check All Mapping Rules of Configurations File of Microservice's Project

Check all YAML and PROPERTIES files in the microservice's project.
bootstrap.yml, bootstrap.properties, application.yml, application.properties
Check all YAML and PROPERTIES files in the configuration server's project related to the microservice.

Step-8 Check All Mapping Rules of Source Code File of Microservice's Project

Check all JAVA files in the microservice's project.
Start with @SpringBootApplication class then @Controller class then a class with @Scheduled method.

Step-9 Filter All Mapping Rules with 1 Value

Step-10 Sort All Mapping Rules by the following PIM Concept (Source) order:

- *Microservice Architecture*
- *Container*
- *Microservice*
- *Functional Microservice*
- *Infrastructure Microservice*
- *API Gateway*
- *Configuration*
- *Registry and Discovery*
- *Security*
- *Monitoring*
- *Tracing*
- *Log Analysis*
- *Service Interface*
- *Service Operation*

Step-11 Sort All Mapping Rules by the following PIM Concept (Destination) order:

- *(Blank)*
- *Microservice*
- *Container*
- *Load Balancer*
- *Service Interface*
- *Endpoint*

- *Service Operation*
- *Data Store*
- *Cash Store*
- *Circuit Breaker*
- *Message Bus*
- *Service Dependency*

Step-12 Draw Architecture Diagram: draw concepts and update them as they appear in PIM Concept (Source) except for [Microservice][49] -> Service Dependency

1. **Recover Microservice Architecture:**
   If exists, take architecture name from build file of application's project.
   Else, take architecture name from GitHub repository.
   Otherwise, take architecture name from build file of the first microservice's project.
2. **Recover Microservice Architecture -> Microservice:**
   If exists, take microservice name from configurations' file of microservice's project.
   Else, take microservice name from build file of microservice's project.
   Otherwise, take microservice name from container orchestration file.
3. **Recover Microservice -> Container:**
   If exists, take container name from container orchestration file.
   Else, take container name from build file of microservice's project.
   Otherwise, take container name from configurations' file of microservice's project.
4. **Recover Microservice -> Service Interface:**
   If exists, take server path from configurations' file of microservice's project.
   Else, take server path from build file of microservice's project.
   Otherwise, take server path from container orchestration file.
   Finally, add path prefix.
5. **Recover Service Interface -> Endpoint:**
   Finally, add path prefix to all recovered endpoints.
6. **Recover Service Interface -> Service Operation:**
   Add prefixes at the end to all recovered endpoints.
7. **Recover Service Operation -> Data Store:**

[49] Microservice, Functional Microservice, Infrastructure Microservice, API Gateway, Configuration, Registry and Discovery, Security, Monitoring, Tracing, Log Analysis.

If possible, trace back the service operation that invoked functions mentioned in the mapping rule.

8. **Recover Service Operation -> Circuit Breaker:**
   If possible, trace back the service operation that invoked functions mentioned in the mapping rule.
9. **Recover Service Operation -> Message Bus:**
   If possible, trace back the service operation that invoked functions mentioned in the mapping rule.
10. **If exists, recover [Infrastructure Microservice][50]:**
   Otherwise, set Microservice to Functional Microservice.

After all Microservice Concepts are Recovered:

Step-13 Refine Architecture Diagram: Recover [Microservice][51] -> Service Dependency

1. **Recover Microservice -> Service Dependency:**
   If provider name is not available, filter mapping rule list by [Infrastructure Microservice][52] type of the provider type mentioned in the mapping rule.
   If possible, trace back the service operation that fired the mapping rule of provider operation.

2. **Recover [Infrastructure Microservice][16]-> Service Dependency:**
   Filter mapping rule list by the [Infrastructure Microservice][16] type of the currently recovered Microservice.
   If possible, trace back the service operation that fired the mapping rule of provider operation.

---

[50] API Gateway, Configuration, Registry and Discovery, Security, Monitoring, Tracing, Log Analysis.
[51] Microservice, Functional Microservice, Infrastructure Microservice, API Gateway, Configuration, Registry and Discovery, Security, Monitoring, Tracing, Log Analysis.

Appendix-B: 4- A PIM notation at microservice level

PIM Concept	Microservice Level Diagram Box	Draw Box Inside
FunctionalMicroservice - MicroserviceName: EString	{Microservice-Name} {Functional Microservice}	None
InfrastructureMicroservice - MicroserviceName: EString	{Microservice-Name} {Infrastructure Microservice}	None
ServiceInterface - ServerURL: EString	{Service-URL} {Service Interface}	[Functional Microservice] [Infrastructure Microservice]
Endpoint - RequestURI: EString - Environment: EString	{Request-URI} {Endpoint} {Environment}	[Service Interface]
QueueListener - QueueName: EString - Environment: EString	{Queue-Name} {Queue Listener} {Environment}	[Service Interface]
ServiceMessage - MessageType: EString - BodySchema: EString - SchemaFormat: EString	{Message-Type} {Service Message} {Environment}	[Endpoint] [Queue Listener]
ServiceOperation - OperationName: EString - OperationDescription: EString	{Operation-Name} {Operation-Description} {Service Operation}	[Endpoint] [Queue Listener]
InfrastructurePatternComponent - Category: InfrastructurePatternCategory - Environment: EString	{Category} {Infrastructure pattern component}	[Functional Microservice] [Infrastructure Microservice]

InfrastructureClientComponent - Category: InfrastructurePatternCategory - Environment: EString	{Category} {Infrastructure client component} {Environment}	[Functional Microservice]
		[Infrastructure Microservice]
InfrastructureServerComponent - Category: InfrastructurePatternCategory - Environment: EString	{Category} {Infrastructure server component} {Environment}	[Infrastructure Microservice]