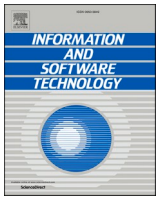
Information and Software Technology 186 (2025) 107808

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



A model-driven architecture approach for recovering microservice architectures: Defining and evaluating MiSAR

Nuha Alshuqayran [a,*], Nour Ali [b], Roger Evans [c]

[a] College of Computer and Information Sciences (CCIS), Imam Mohammad Ibn Saud Islamic University (IMSIU), Saudi Arabia
[b] Department of Computer Science, Brunel University of London, Uxbridge, UK
[c] Computing Engineering and Mathematics, University of Brighton, UK

ARTICLE INFO

Keywords:
Microservices
Model-driven engineering
MDA
Microservicearchitecture recovery
Reverse engineering

ABSTRACT

Context: Microservice architecture is an architectural style in modern software systems, characterized by small, independent services called microservices. This architecture is ideal to facilitate rapid feature deployment. However, it presents a challenge for software engineers, who often lack a comprehensive architectural view due to the distributed nature and complex interdependencies of microservices.
Objective: This paper presents a Model Driven Architecture approach for MicroService Architecture Recovery called MiSAR. Building on previous work that defined a Platform Independent Metamodel, this study seeks to extend this metamodel, introduce a Platform Specific Metamodel, and establish mapping rules. The goal is to enable the semi-automatic recovery of architectural models for microservice systems.
Methods: An empirical study was conducted on nine microservice systems to define MiSAR's artefacts and support semiautomatic recovery of architectural models. These artefacts are then implemented and used to semi-automatically recover the architectures of three systems. The effectiveness of MiSAR is evaluated based on metrics such as recall, precision, and F-measure, to assess the recovered models against actual architectures. We also compared the recovered architectural models with the ones documented by the developers.
Results: The study identified key requirements for the Platform Independent Metamodel to support comprehensive microservice architecture recovery, leading to an incremental extension of the MiSAR Platform Independent Metamodel. Mapping rules were established to effectively transform Platform Specific Models into Platform Independent ones. Furthermore, MiSAR was successfully implemented to recover architecture models. An evaluation using three systems demonstrated that MiSAR could recover architectural models with a high degree of completeness and correctness when compared with the actual architecture.
Conclusion: The MiSAR artefacts, including the extended Platform Independent Metamodel and mapping rules, effectively produce expressive architectural models of microservice systems. Systems confirmed MiSAR's ability to semi-automatically recover accurate architectural models, providing a holistic view often missing in current software engineering practices.

1. Introduction

As software applications evolve, their conceptual architectures often no longer represent their implementations. As software engineers lack an accurate and holistic understanding of their applications, it is hard to successfully refactor, migrate and upgrade them [1]. To overcome these issues, software architecture recovery (or reconstruction) [2], has recently received considerable attention [3,4] to obtain the actual (as-implemented) architectural structure and description from system artefacts such as source code. Architecture Recovery is an important asset for many software engineering activities, enabling software engineers to have control and understand improvements in software systems.

Software companies today emphasize continuous delivery to enhance customer value. MicroService Architecture (MSA) is a popular strategy for achieving this [5]. MSA is a type of service-oriented architectural style which is technology agnostic, and involves designing software as a set of independent services, each with a single business responsibility and independently running in isolation to other microservices [6]. While no precise definition pertaining to MSA exists, its key characteristics include: independent deployability of microservices, microservices should own their own state and they should communicate

* Corresponding author.

https://doi.org/10.1016/j.infsof.2025.107808
Received 30 July 2024; Received in revised form 31 May 2025; Accepted 2 June 2025
Available online 3 June 2025

via lightweight mechanisms and smart endpoints [7].

The MSA approach provides significant benefits including reliability, scalability, separation of concerns and ease of deployment [6,8,9]. However, the challenge of not fully understanding the implemented software architecture is emphasized in its nature: an evolutionary architecture composed of numerous, dynamic, small and distributed microservices with several inter- and intra-dependencies. Microservices architectures are inherently complex due to their reliance on other microservices and infrastructural components such as API gateways and monitoring components. This complexity increases with a high number of dependencies and the use of multiple technologies [7].

Architecture recovery [10] is a promising approach to aid in comprehending MSAs'complexity as it allows software engineers (developers/architects) to obtain an architectural model of the implemented system and its structure. Microservice architecture recovery supports software engineers in obtaining an up-to-date architecture of the implemented system and this has many benefits to software engineers in many cases which include: 1) comprehending the complexities of distributed microservice systems and identifying inter-service communication; 2) obtaining an up-to-date architecture for documentation; 3) inter-team communication and architecture awareness among different microservice teams because microservices of the same system are developed by different autonomous teams with different technologies; 4) identifying architecture inconsistencies between the implementation and an architecture and identifying architecture smells.

However, specific challenges for microservice architecture recovery include: microservices not being first-class software elements, microservice systems use different languages and technologies, and microservices are highly interdependent, complicating analysis and architecture abstraction. Therefore, there is a dearth of available architecture recovery approaches within the area of microservices [8]. Available approaches partially recover the architecture of microservice systems as they lack support for key architectural concepts related to asynchronous communication, e.g., [11] and microservice infrastructure, e.g., [12] and they haven't been evaluated for effectiveness. To address these issues, we propose the MicroService Architecture Recovery (MiSAR) approach, which follows the Model Driven Engineering (MDE) paradigm [13] to support the recovery of architectural models of microservice systems.

Our previous work conducted an empirical study where we manually extracted and clustered architectural concepts from microservice systems [10]. The latter, produced a Platform-Independent metamodel from analysing the source of the systems. However, an architecture recovery approach was not defined which is capable of semi-automatically recover architectural models. MiSAR was initially developed using 8 systems [10], and in this work, we expanded it by incorporating an additional 9 systems, leading to a total of 19 systems informing the design.

Therefore, in this paper, we define the holistic microservice architectural approach for MiSAR by defining the Model Driven Architecture (MDA) aretfacts: Platform Independent Metamodel, Platform Specific Metamodel and transformation rules that are able to generate architectural models of implemented microservice systems in a semi-automatic way. To achieve this, we have designed an empirical study where we identify a set of architectural requirements and apply them to extend the Platform-Independent Metamodel and define mapping rules, which automatise the architecture recovery process. Then, we implement the artefacts and use MiSAR to semi-automatically recover architectural models of 3 microservice systems. Finally, we evaluate the MiSAR recovered models and measure their completeness and correctness by comparing them to the actual architectures of the implemented systems. We also compare MiSAR models to the documentation written by the developers.

The main contribution of this paper is the introduction of a Model-Driven Architecture (MDA) approach for microservice architecture recovery, which semi-automatically generates expressive architectural models of microservice systems. The expressiveness of these architectural models encompasses elements such as support for asynchronous communication and infrastructure patterns, which are vital components of a microservice architecture [6]. This achievement is facilitated through the following sub-contributions: (i) Introduction of the MiSAR Platform-Specific Metamodel; (ii) Inclusion of mapping rules that are well defined and structured to support the automatic transformation from Platform Specific Models to Platform Independent Models; (iii) Implementation of MiSAR artefacts, including: 1) mapping rules in QVTo [14], 2) Platform-Specific and Platform-Independent metamodels in Ecore [15], and 3) parser to support the semi-automatic generation of architectural models; (v) Definition of MiSAR's recovery process; (vi) Application of MiSAR's recovery process to three systems and the evaluation of their semi-automatically generated architectural models.

This paper is organised as follows. Section 2 presents background information on model-driven engineering. Section 3 introduces MiSAR. Section 4 presents the study design used to define and formalize the MiSAR artefacts. Section 5 and Section 6 present our results leading to MiSAR artefacts and their implementation. Section 7 explains MiSAR's recovery process. Then, in Section 8, we evaluated MiSAR through three systems. Section 9 discusses our approach. Section 10 describes related work. Lastly, we conclude the paper and envision future research.

2. Model-driven engineering

This section provides a brief background on Model-Driven Engineering (MDE) [13]. MDE depends on three key characteristics: (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 of languages used to describe other models. A model is considered to" conform to" or "is an instance of' a metamodel. A metamodel identifies each concept that is used in defining a specific model, and the models use the concepts according to the relationships and rules specified by the metamodel [16, 13].

MDE has started to be recognised in the research community for addressing reverse engineering problems in the last few years [17]. 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 [18].

Another benefit is 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.

Model-Driven Architecture (MDA) is a set of guidelines for implementing MDE from the Object Management Group (OMG). In MDA, models can be Platform-Specific Models (PSMs) and Platform-Independent Models (PIMs) [19]. A PSM contains a set of technical concepts linked to technology-specific platforms, open or proprietary, such as Web Services, .NET, CORBA, J2EE and others. A PIM abstracts away technical details and is independent of platforms and technologies.

Previously, MDA has been used to generate code of service-oriented architecture [20]. In this work, we focus on the MDA's PIM and PSM abstraction levels in relation to the modeling of MSA. These models are critical for architecture recovery, where a PIM supports the architectural model recovered and a PSM supports the technology of implemented microservice systems.

3. Overview of the microservice architectural recovery (MiSAR) approach

This section presents an overview of MiSAR and how it supports the

architecture recovery of microservice systems. MiSAR follows MDE [13] to recover semi-automatically architectural models of existing microservice systems, by developing bottom-up, model-driven transformations for obtaining architectural models from the implementation level. MiSAR can unveil the architectural aspects and aims to abstract the complexities of MSA by allowing software engineers to understand an architecture's implemented structure.

MiSAR considers elements at three different abstraction levels (see Fig. 1). Level M0 includes the microservice software system as a set of physical artefacts. It currently considers source code, configuration, Docker, Docker compose, build files (POM) at the microservice level and project build files at system level, which contain information used by Maven to build a project. The M1 level, is the PSM which represents the software artefacts of M0 in different models, which conform to their Platform-Specific Metamodel, and supports the technology of the implemented microservice system. The M2 level represents the PIM, which abstracts the concepts of MSA in a technology-independent way.

Mapping rules are needed to map an implemented microservice-based system into an architectural model by instantiating the Platform-Independent Metamodel. The current mapping rules only support reverse engineering (as in Fig. 1) which involves transforming from lower abstraction levels to higher ones and not forward engineering. Future extensions of MiSAR could explore mechanisms for maintaining consistency/conformance between different abstraction levels.

In our previous work [10], we identified an initial Platform-Independent Metamodel by studying eight microservice systems. This paper extends that metamodel, introduces the Platform-Specific Metamodel, and defines the mapping rules between them by incorporating nine additional systems, bringing the total to 19 systems that inform the design of MiSAR. Furthermore, this work implements these rules to automatically generate architecture models for microservice systems.

3.1. Platform-Independent metamodel

The Platform Independent Metamodel defines the microservice architectural elements that describe a microservice architecture in a technology independent way. In the following, we explain the Platform-Independent Metamodel (as presented in [10]), which will be refined in Section 5.1. As shown in Fig. 2, Microservice is the central and main building block of our metamodel, and it is generally a software application that offers a completely independent service. Microservices are broadly classified into Functional Microservice types, which realize the system's business capabilities, and Infrastructure Microservice types, which implement an infrastructure pattern/component addressing a particular concern of a MSA, such as API Gateway, Configuration, Discovery and Registry and Tracing. The deployment concern of a MSA model is represented by the concept of Ambient as in [21] and Container elements. A container is an execution environment used to isolate each microservice, leveraging the host's hardware and operating system capabilities while enabling each microservice to appear as a completely stand-alone software artefact. The Service Dependency element describes the communication between one consumer microservice and one provider microservice. The Service Interface element aggregates all Service Operations as well as exposes Endpoints of a microservice. 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. A Service Operation reflects the main procedure/function that is directly executed by calling a corresponding endpoint.

3.2. Platform-specific metamodel

The MiSAR Platform-Specific Metamodel supports several platforms: the Java Language, Docker, Spring boot framework and technologies which include Consul, Eureka, MongoDB, MySQL, Neo4j Graph database, OAuth2 and RabbitMQ. Fig. 3 presents the parts of the MiSAR Platform-Specific Metamodel that include the Distributed Application Project as root element. It is described by application name and its root repository URI. Distributed Application Project captures the

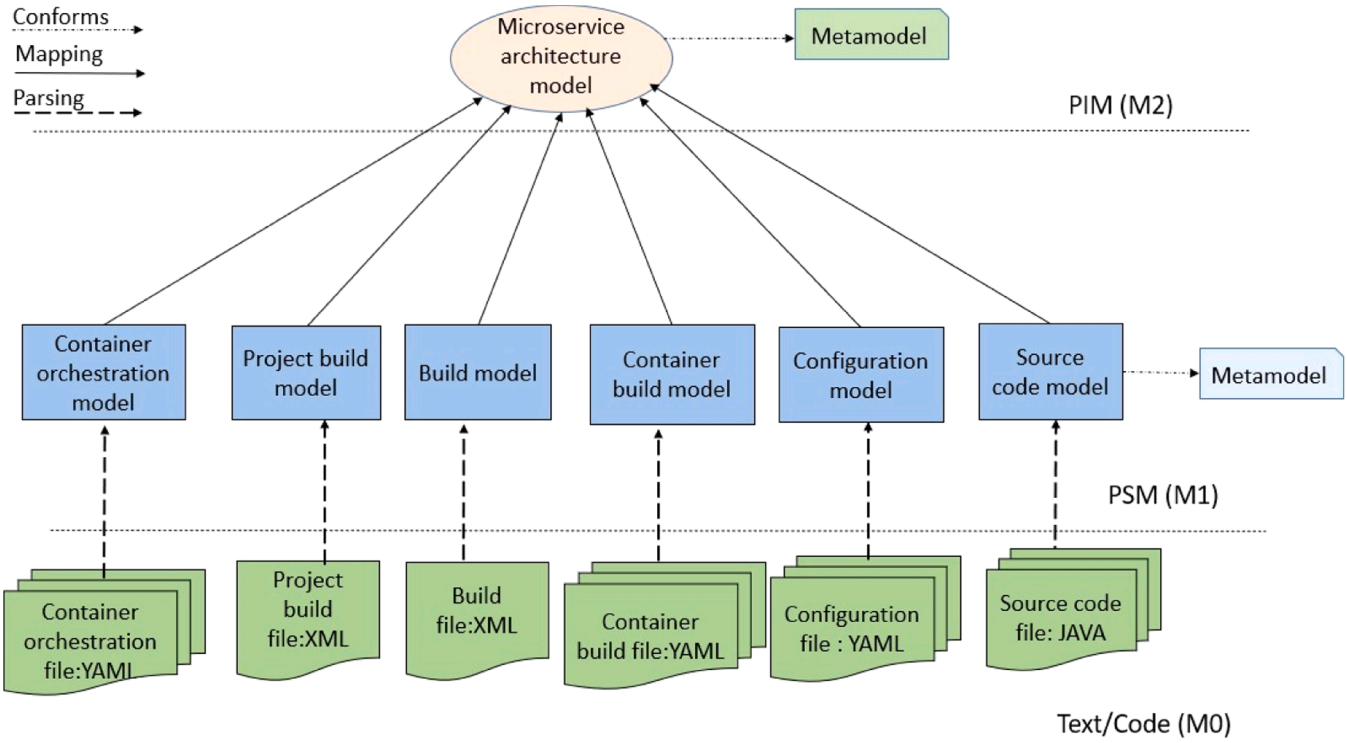


Fig. 1. MiSAR abstraction levels.

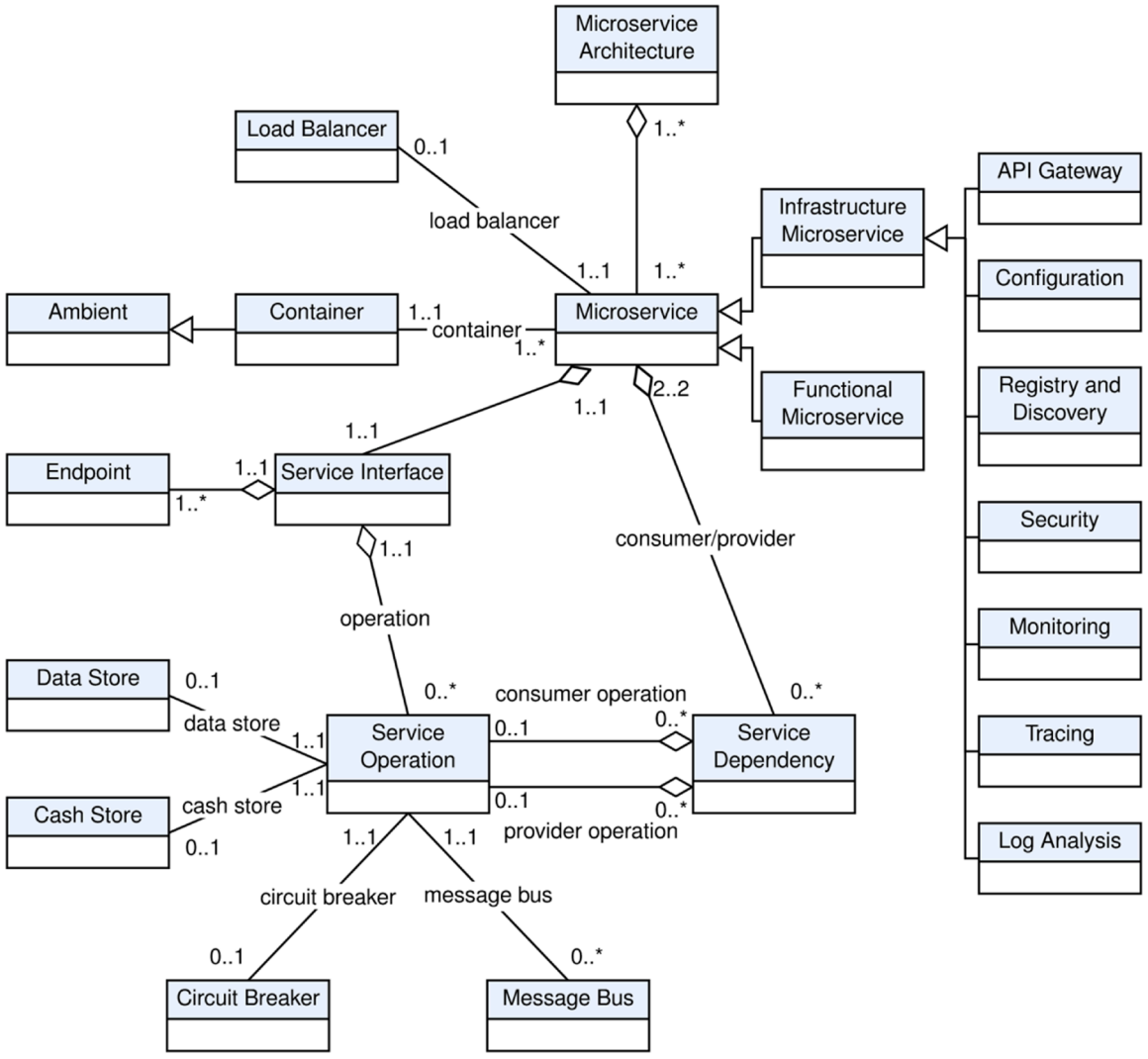


Fig. 2. Initial design of the MiSAR platform independent metamodel (version 1), as presented in [10].

architecture's development artefacts (multi-module project and module projects) as well as the runtime artefacts (Docker containers). The runtime artefacts are the collection of Docker Container Definition elements involved in the architecture and are defined in the Docker Compose as well as the Docker files. Each Docker Container Definition is described by a container name, build path, image name, and whether it generates logs 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 Docker Container Port and Docker Container Link runtime information is also captured for each Docker container.

The development artefacts are generally represented by the Application Project element, which is equivalent to a multi-module project along with its module projects, each represented by a MicroserviceProject element. The Microservice Project element generalises a wide range of project artefacts implemented in any framework or language, including Java Spring Boot/Cloud. Each Microservice Project defines a collection of Dependency Library elements that can be found in project build artefacts such as Maven POM.XML or Gradle BUILD. GRADLE. It also defines a collection of settings in YAML or PROPERTIES artefacts. These settings are represented with the Configuration Property element, which defines important functionality and execution information. The Java Spring Web Application Project element is a subtype of the Microservice Project element which reflects the specific characteristics of Spring Boot/Cloud framework applications.

4. Empirical study design

This section presents the aim and research questions of our study, followed by the research design and the protocol used to select systems for addressing two of our research questions.

4.1. Study aims and research questions

This study aims to incrementally formalize the Model Driven Architecture (MDA) artefacts of MiSAR using microservice-based systems. To achieve this, we defined three research questions:

RQ1: What are the architectural elements that are required for an existing microservice architectural model?
RQ2: What are the mapping rules between the Platform Specific Model and the Platform Independent model needed to automatize the model transformation of MiSAR?
RQ3: How does MiSAR perform in recovering semi-automatically microservice architectural models?

RQ1 focuses on including new architectural elements into the PIM metamodel [10] and RQ2 focuses on defining mapping rules between the PSM and the PIM by analyzing manually microservice systems. RQ3 focuses on applying all MiSAR implemented artefacts and obtaining semi-automatically architectural models at PIM level of microservice

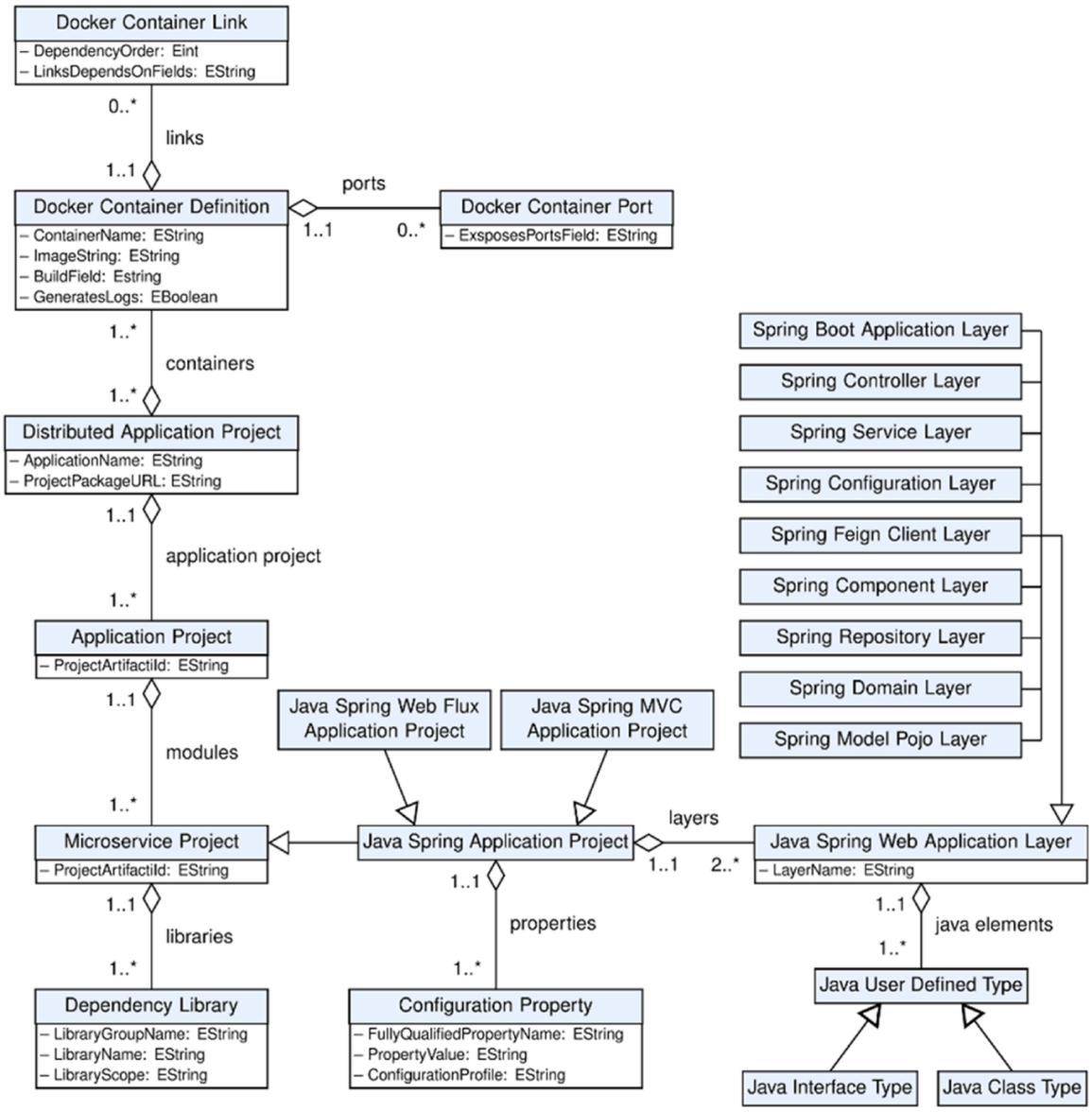


Fig. 3. Artefacts defined in the MiSAR platform-specific metamodel, supporting Java Spring and Docker containers.

systems. To evaluate the recovered architectural models, we will: 1) compare them with the actual architectures of the implemented systems, 2) we will compare them with the documented architectures and 3) we will evaluate the efficiency of the automatic recovery.

4.2. Research design

We follow the guidelines of [22]. To answer RQ1 and RQ2, our research design adopts an example-driven approach, where we have derived MiSAR from example microservice systems. This approach has been used in software engineering such as in metamodelling [23] and architecture reconstruction [24].

The example microservice systems used were searched on the github repository. We followed the guidelines in [25] to report the data source selection and threats to validity. We mainly analysed the source code, configuration files and documentation of the microservice systems.

In the following, we describe the design of our study, as depicted in Fig. 4:

Manual Architecture Recovery: This phase is divided into Activities A1 and A2. Both include manual recovery and are always performed in parallel to enhance and refine MiSAR in increments.

Activity A1. Application to metamodel: In this activity, we create increments to the PIM metamodel by applying the systems in Table 1 and identifying new architectural requirements. The objective of this activity is to validate the existing metamodel concepts against real systems and evolve the PIM metamodel accordingly. For each system, we analyze its source code and configuration files, abstract relevant elements, and identify corresponding architectural concepts. We then instantiate the metamodel using elements from its latest increment to create instances of the identified architectural concepts. Details of this process are provided in Section 5.1. If instantiation is successful, the current metamodel is considered sufficient for the given system. If not, we refine the metamodel incrementally by: 1) Defining new architectural concepts, 2) Updating or re-evaluating relationships (e.g., associations), and 3) Grouping related concepts for better integration. This iterative process results in new requirements that guide the metamodel's evolution, as discussed in Section 5.1.

Activity A2. Application to mapping rules: We manually analysed the implementation of the nine systems to define MiSAR's mapping rules between the PSM and the PIM repository. In the previous study [10], we defined mapping rules between source code and PIM, which are not applicable in the MiSAR approach. The objective of Activity 2 involves

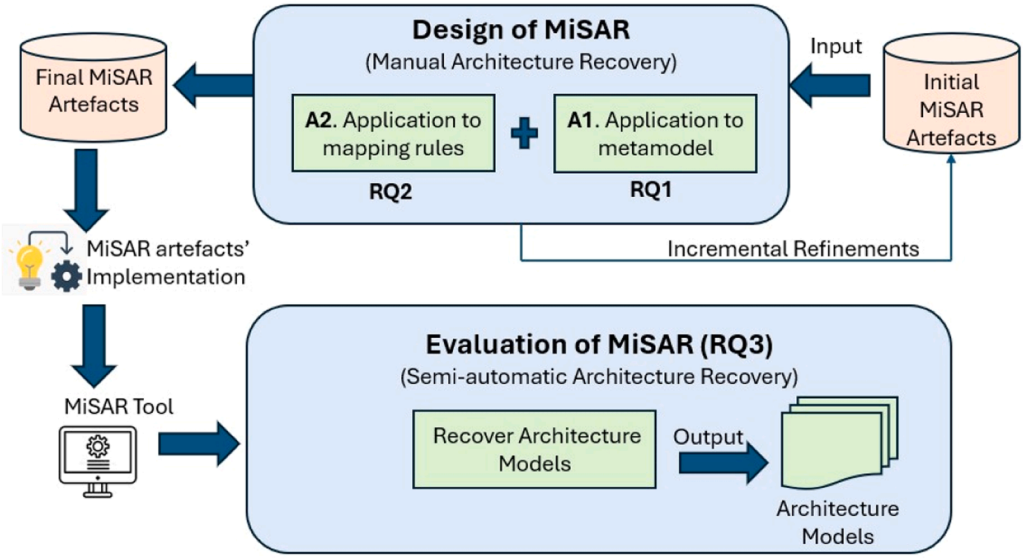


Fig. 4. Research design methodology.

Table 1
The selection criteria for the systems studied.

Criteria	
Inclusions	Step num
• Have architectural diagram and documentation.	2
• Implement business functionality.	3
• Implemented with Spring Boot/Spring Cloud framework in Java.	4
• Include one or more infrastructure Netflix OSS libraries (e.g., Zuul, Eureka, Hystrix, Sidecar).	
• Each module runs in a single process (Docker technology).	
• Implement lightweight synchronous/asynchronous interaction style and smart endpoints.	
• Consist of independent individual services.	
• Have Maven POM or Gradle build file(s).	
Exclusions	Stem num
• Do not use Spring Boot/Spring Cloud framework.	4
• Do not have Docker Compose file(s).	
• Do not revolve around an accumulation of independent individual services.	
• Do not have any data processing libraries (MongoDB, MySQL, or Graph).	
• Do not have Maven POM or Gradle build file(s).	
• Do not have Java source files (e.g., "java" resulted in JavaScript).	
• Do not have a clear description, documentation, paper, or tutorial pages.	2
• Use fewer than two functional microservices.	3
• Only include infrastructure microservices (e.g., development tools, operation frameworks).	

defining structured mapping rules between the PSM and PIM to enable the automatic model transformation for generating architectural models.

Evaluation of MiSAR: To answer RQ3, we implemented the MiSAR artefacts and used them to semi-automatically recover the architectures of systems. We have evaluated the MiSAR approach through systems. The details of the evaluation are described in Section 8.

4.3. Selecting the systems for RQ1 and RQ2

Based on the guidelines in [25], we designed a selection protocol, consisting of the steps as illustrated in Fig. 5, which also shows the number of systems filtered at each step. The steps for the selection and filtering of the 9 best systems can be found on GitHub.[1] We also defined, following the guidelines of [22], a list of inclusion and exclusion criteria to select the systems. We implemented the criteria in 4 steps as presented in Table 1. The list of selected systems in this empirical study is shown in Table 2.

Step 1. Basic search: We designed our search string to be a conjunction of two corresponding populations: Microservice population AND Frameworks technology population. Concerning the Microservice population, we have considered that Microservice itself should be a recurrent keyword, just to make sure, we have widened the search by including a more open keyword with the prefix service:

Microservice population = (microservice* OR "micro-service"* OR "micro service"* OR "microservice architecture"*).

As for the Frameworks technology population, we included the frameworks and technologies which currently MiSAR is designed to support. As a result, the string is:

Frameworks technology population = (spring* AND java* AND (docker-compose OR docker) * OR netflix* OR asynchronous* OR reactive*).

We applied the search string over the GitHub repository and the outcome resulted with 121 items.

Step 2. Initial scan for system documentation: The title, description, architectural diagram and documentation of the projects were reviewed by one of the authors. Projects were discarded for not having a clear description (or written in other languages than English, e.g., Chinese and Spanish), any documentation, paper or tutorial pages. The outcome result set of this step turned out to have a length of 78 items.

Step 3. System functionality considered: Two authors were appointed to every remaining project to examine the functionality of the systems. We included projects which implement at least two business functionalities (e.g.,

"Stock Price Viewer" system was excluded since it implements only one business service named "stock-service") and excluded projects that only implement infrastructure services, (e.g., "Microservice Monitoring" system was excluded since it only implements monitoring infrastructure services). The outcome result set of this step turned out to have a length of 55 items.

Step 4. Source artefact analysis: All three authors were involved in an

[1] https://github.com/MicroServiceArchitectureRecovery/misar/tree/main/EmpiricalStudyReplication/SelectedSyst emsUsedToDefineMiSARartefacts

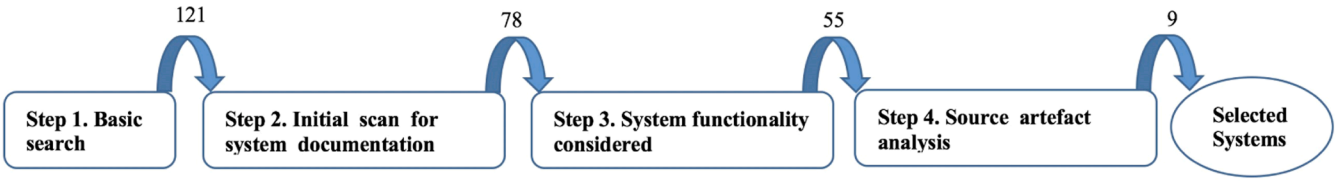


Fig. 5. Step by Step filtering process for selecting the systems in our study.

Table 2
Selected microservice systems from GitHub for analysis.

		Microservice		
ID	Project Name	URL	Count	Req
1	Spring-Netflix-OSS-microservices	[26]	9	2.1,2.2
2	Spring-RabbitMQ-microservices	[27]	7	3.1
3	Cloud-enabled-microservice	[28]	7	1.3
4	Event-sourcing-microservices	[29]	10	1.3
5	Spring-cloud-sidecar-polygot	[30]	7	1.1,1.2,1.3
6	Microservices-basics-spring-boot	[31]	10	2.1,2.2
7	Spring-cloud-event-sourcing	[32]	15	1.1, 1.3
8	Spring-Boot-Graph-Processing	[33]	9	1.3, 3.1
9	BookStoreApp-Distributed-Application	[34]	14	1.2, 3.1

analysis which checks the files of the artefacts in full and searches through source files to classify the systems as included or excluded based on existence of essential framework artefacts as pointed in the inclusion/exclusion criteria in Table 1. Each researcher is responsible for analyzing the artifacts of their assigned systems to identify specific patterns or technologies. The division of the responsibilities for each researcher and how the 9 systems were selected can be found on GitHub.[2]

5. Results of RQ1 and RQ2

In this section, we present our analysis and results according to our research questions: RQ1 and RQ2. As stated in Section 4.2, activities 1 and 2 are performed in parallel, but we have separated them for presentation purposes.

5.1. MiSAR platform-independent metamodel (RQ1)

In this section, we present our analysis of the architectural concepts empirically derived from the nine systems in a number of increments. Each increment identified/ elicited a set of requirements that enhance architectural support and expressiveness within a microservice architectural metamodel. The Req column in Table 2 indicates which requirements emerged from the analysis of each system. For each elicited requirement, we explain how we have fulfilled it by modifying the Platform Independent Metamodel to create an updated Platform Independent Metamodel version (see Fig. 6), enabling the recovery of more expressive architectural models for microservice systems. We represent these modifications in labelled boxes, where Requirement- denotes the elicited requirement, and Application of requirement- describes how the requirement was fulfilled within the metamodel.

5.1.1. Increment–1: supporting components of microservice patterns

Context-1: Many microservice patterns are supported in microservice-based implementations by the usage of frameworks. For example, service operations are not defined in the source. For illustration, the edge-service microservice in system 7 (mentioned in Table 2) uses API Gateway and Circuit Breaker patterns via Spring Boot/Cloud framework, without explicitly implementing any Service Operations in its source artefacts. This discussion leads to the following requirements:

Context-2: An infrastructure microservice can have multiple infrastructure patterns. For illustration, the bookstoreconsul-discovery microservice from system 9 provides multiple infrastructure patterns simultaneously, including Configuration and Registry and Discovery.

We drew inspiration for the microservice pattern categories from the groupings presented in [35] and [36]. For each system we analysed, we checked the presence of these groupings and patterns. Therefore, we added a new enumeration type, Infrastructure Pattern Category, to the metamodel (see Fig. 6).

Context-3: An Infrastructure microservice can use a pattern component as a client or provider. To illustrate, the edge-service microservice in system 7 uses both infrastructure patterns: Registry and Discovery, and Configuration. The discovery-service provides Registry and Discovery to edge-service, since the edge-service uses this pattern to register its address. The config-server is considered a Configuration provider since edge-service needs this pattern to pull its centralized configuration properties.

5.1.2. Increment–2: supporting synchronous communication through endpoints

Context-1: Request-response synchronous inter-service communication is usually represented by Service Dependencies. However, it is also important to know which Service Operations invoke remote provider operations. This is often unclear in the source code due to frameworks abstracting Service Operation implementations.

In addition, information about the format of the request and response data messages at the provider's endpoint is important to be represented. Also, the response/output message should be specified by the data type of the object returned, if any.

In response to the requirements of Increment 2, Fig. 6 illustrates how the Endpoint, Service Message and ServiceOperations are defined.

5.1.3. 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.

To illustrate, we examine the message-driven inter-service communication implemented in System 2. In this system, there are three queues- weathersimple:queue, weatherbackend:queue and weatherservice:queue, which are bound to the message exchanges. We observe that when a microservice's outbound queue corresponds to another microservice's inbound queue, a Dependency exists between them, as the former sends messages to the latter. In this context, inbound queues act as asynchronous alternatives to traditional service endpoints.

The metamodel now has now a QueueListener and Message Destination concepts as shown in Fig. 6.

[2] https://github.com/MicroServiceArchitectureRecovery/misar/blob/main/EmpiricalStudyReplication/SelectedSyst emsUsedToDefineMiSARartefacts/StepsforSelectionOfSystems.pdf

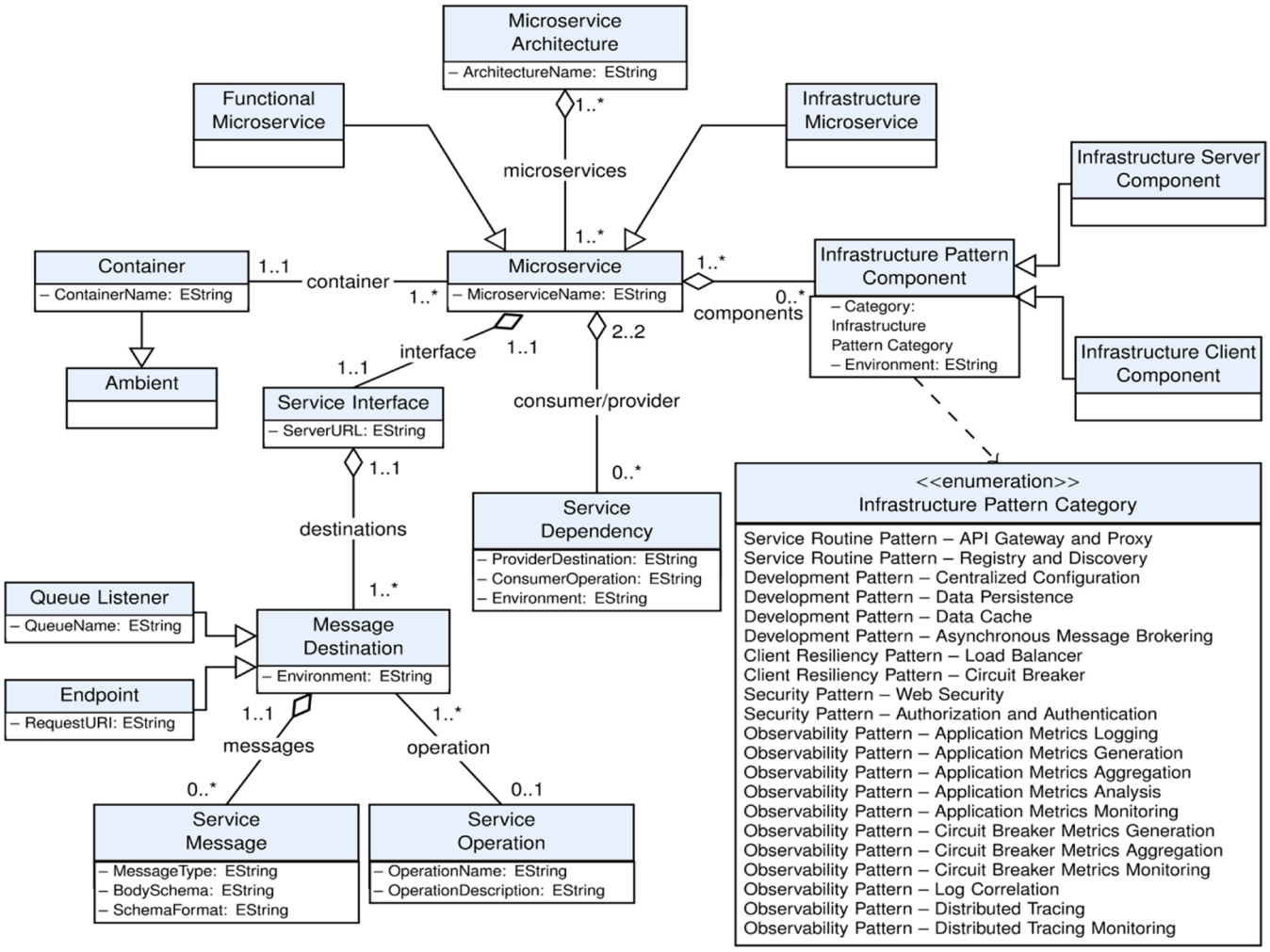


Fig. 6. Final PIM metamodel (version 4).

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.

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

Requirement-1.2 → One Infrastructure Microservice can have multiple-infrastructure patterns.

Application of requirement-1.2 → A new Infrastructure Pattern Component concept is introduced. A microservice can aggregate zero to many Infrastructure Pattern Components. Each component represents an architectural element supporting a pattern's functionality and is classified into a specific Infrastructure Pattern Category. All subtypes of Infrastructure Microservice from metamodel version 1 are now instances of this enumeration.

Requirement-1.3 → The metamodel should distinguish infrastructure pattern components that microservices use. Infrastructure pattern components should be divided into two types: those that provide services to microservices and those that request them.

Application of requirement-1.3 → Infrastructure pattern component has 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.

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

Application of requirement-2.1 → The association of Service Operation is re-positioned from Service Interface to Endpoint. This association is an optional association that goes from Endpoint to Service Operation.

Requirement-2.2 → An Endpoint of a microservice should define the format and type of its data messages, if any.

Application of requirement-2.2 → A Service Message concept is introduced. Service Message is associated with Endpoint and it is defined by MessageType, i.e., request/response/error, Schema and Schema Format, i.e.,

XML/JSON.

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

Application of requirement-3.1 → The concept Queue Listener is introduced, defined by its Name and, Endpoint is associated with Service Interface. 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.

5.2. MiSAR mapping rules (RQ2)

We analysed the nine systems using Activity 2. A sample of the defined mapping rules are shown in Table 3. MiSAR's mapping rules are in a structured tree that maps PSM element(s) into target PIM element (s). Mapping rules are represented with a Left-Hand Side (L-H-S) and a Right-Hand Side (R-H-S). At the L-H-S, PSM elements are specified by their attributes' values and the references between them. Mapping rules check that the L-H-S or PSM elements exist. If they exist, then PSM elements are transformed into a group of target PIM elements specified at R-H-S, with specific attribute values and references between them. L-H-S elements' are identified before the word 'indicates' and R-H-S elements' are identified after 'indicates'. The aim of this structure is to formalize the transformation, facilitate the implementation and eventually make the recovery process more 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), and then group them by each source (PSM element).

Table 3
Specification of 6 out of 275 MiSAR mapping rules.

	Systems from Table 2where the rule applies								
	1	2	3	4	5	6	7	8	9
R1: Mapping Docker Container Definition to Infrastructure Server Component: [L-H-S] One *Docker Hub Image Container* with *Image Field* value which contains: "consul", indicates [R-H-S] 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"*.									×
R2: Mapping Annotation of a Java Method in a Java Class to Service Operation: [L-H-S] 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 [R-H-S] a *Service Operation* with *Operation Name* value: "[operation-name]" and *Operation Description* value: "An operation with name [operation-name] that responds with object [datatype-name]".	×	×	×	×	×	×	×	×	×
R3: Variation of R2 by using reactive web application (WebFlux): [L-H-S]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" or "Flux" indicates [R-H-S] a *Service Operation* with *Operation Name* value: "[operation-name]" and *Operation Description* value: "An operation with name [operation-name] that responds with object [datatype-name]".				×					
R4: Mapping A Java Annotation and its Parameter to Queue Listener: [L-H-S] A *Java Annotation* with *Annotation Name* value: "RabbitListener" which has a *Java Annotation Parameter* with *Parameter Name* value: "value" or "queues" and *Parameter* value: "[queue-name]" and belongs to a *Java Method* with *Element Profile* value: "[destination-environment]" indicates [R-H-S] a *Queue Listener* with *Queue Name* value: "[queue-name]" and *Environment value*: "[destination-environment]".		×						×	
R5: Mapping A Java Method to Service Dependency (asynchronous communication): [L-H-S] A *Java Method* with *Element Identifier* value: "convertAndSend" whose parent is a *Java User Defined Type* with *Element Identifier* value: "RabbitTemplate" or "AmqpTemplate", which has one *Java Method Parameter* with *Parameter Order* value: "2″ and *Field* 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 [R-H-S] a *Service Dependency* with *Provider Destination* value: "QueueListener[QueueName:[Queue Name]]".		×						×	
R6: Mapping A Dependency Library to Endpoint and Service Message: [L-H-S] A *Dependency Library* with *Library Name* value:"spring-boot-starter-actuator" and *Library Scope* value: "[destination-environment]" indicates [R-H-S] 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"]]] ".	×		×	×	×		×	×	×

For example, the R1 mapping rule in Table 3, which resulted from Increment 1 (see Section 5.1.1) of introducing the Infrastructure Pattern Component, transforms a PSM Element: Docker Hub Image Container with Image Field: [Image Field =‘ consul’] into three target PIM elements: an Infrastructure Server Component with Category value: [Service Routing Pattern - Registry and Discovery], an Infrastructure Server Component with Category value: [Development Pattern-Centralized Configuration] and an Infrastructure Server Component with Category value: [Development Pattern - Asynchronous Messaging].

An example of rule R2 is its ability to recover the Service Operation concept, as demonstrated in Table 3. The variation in mapping rules involves adding new mapping rules that recover existing PIM concepts, implemented using technologies not previously encountered [10]. For instance, rule R3 also recovers the Service Operation concept, producing the same output as R2. However, the input in R3 represents a reactive, non-blocking microservice architecture. This reactive architecture was introduced following the analysis of system 4.

Rules R4 and R5 are added as a result of Increment 3 (see Section 5.1.3): R4 recovers the Queue Listener concept which is a Message Destination based on message-driven inter-service communication and R5 recovers service dependency for asynchronous communication. These concepts were added after analysis of systems 2 and 8.

The addition of mapping rules with hard-coded values to recover production endpoints that are not implemented explicitly in source code and are activated only at runtime, along with their message types and the formats of outsourced famous infrastructure technologies, an example is R6.

6. Implementation of MiSAR artefacts

To support the automatic recovery of the MiSAR architectural models, the metamodels and mapping rules have been implemented. Metamodels have been implemented as Ecore models using the Eclipse Modeling Framework (EMF) [15]. Fig. 7 presents the MiSAR PSM Ecore metamodel, which can be found on GitHub.[3] To develop and automate the mapping rules, we have used the Eclipse Model-to-Model Transformation (M2M) project, by incorporating the operational QVT transformation language (QVTo) [14]. Fig. 8 shows the implementation of the R4 mapping rule of Table 3. The mapping body which includes the init-section (lines 5 to 7) used for initialising parameters and variables, and the population section (lines 9 to 18), which specifies the actual mapping are illustrated. In line 14 the DependencyLibrary2Endpoint() invokes the mapping DependencyLibrary2EndpointServiceMessage(). In an Eclipse QVTo implementation, mapping rules are organised in a top-bottom order i.e., a rule that maps a top PSM element such as ApplicationProject should invoke all mapping rules to recover a top PIM element such as Microservice Architecture and its subsequent elements.

Instances of a PSM metamodel are instantiated using the MiSARParser. The MiSARParser is a python application that incorporates PyEcore, JavaLang, Yaml, XMLtoDict and other python libraries in order to parse a microservice-based application into a MiSAR PSM model that will be fed as input to the QVTo transformations for the final generation of a MiSAR PIM model. Currently, the parser analyses the following files of a microservice system: Docker Compose Files (.yml—.yaml), Maven POM (Project Object Model) files, Configuration Files (.yml—.yaml—.properties) and JavaSourceFiles which specifically include either the org.springframework.boot or the org.springframework.cloud.

7. Application of misar

Fig. 9 illustrates the different parts of the architecture recovery process; the thick arrows represent the steps of the recovery process; the boxes represent the artefacts and the thin arrows indicate the inputs and outputs of the transformation engine. To apply the process of MiSAR, download the MiSAR project and follow the manual described in ”MiSAR Recovery Steps.pdf’ found on GitHub.[4] In the following, we explain the MiSAR architecture recovery process:

Step 1- Artefact collection (semi-automatic): The files from GitHub are first downloaded locally. Then, the required artefacts are uploaded to the existing MiSAR parser, as illustrated in Fig. 10. The user has to input the Project name. Users can choose between an automatic uploader or a manual selection process. If users select a manual selection process, they have to specify: Build directory of each microservice (single-module) project, Path of the build file (POM) for the entire system (multi-module) project, Path of the build file (POM) for each microservice (single-module) project. Additionally, users can delete or add uploaded files, allowing them to control which parts of the system they want to recover. Some may prefer to recover the entire architecture, while others may focus on specific microservices. Because of this user control, the process is classified as semi-automatic rather than fully automated.
Step 2- Automatic Instantiation of the PSM: The MiSAR parser will process the provided artefacts and then generate the PSM model in XMI format at the same path as the PSM Ecore file. This XMI file is instantly readable and viewable by the Eclipse QVTo project.
Step 3- Automatic Recovery of the PIM: The PIM architectural model 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 generated in step 2.

8. Evaluating misar in semi-automatically recovering microservice architecture (RQ3)

This evaluation follows protocols formulated by Brereton et al. [37] and [38] to increase validity and reliability. The design, selection of the systems, procedure, data collection and data analysis, are presented in the next sections.

8.1. Design

The objective is to evaluate the MiSAR approach in terms of recovering architectural models of microservice systems. In this regard, we applied the recovery process and made use of the implemented MiSAR artefacts: Ecore metamodels, QVTo Model Transformations and Parser. The first evaluation is designed to obtain the architectural model completeness and correctness. To do this, we compared the semi-automatically recovered models with the architectural elements of the systems by analysing the source artefacts (e.g. source code, build POM, configuration

YAML, Docker Compose and Docker files). To perform this check, we first manually create an expected architectural model which is an architecture of the implementation of the system by using the MiSAR Platform Independent Metamodel. Then, we manually compare the MiSAR (semi-automatically) recovered architectural model with the one we manually represented. To perform this comparison, we listed all the elements, attributes and their Values of the MiSAR representation of the actual architecture and checked them against the ones in the recovered architectural model. The systems were adopted to answer the following research questions:

CRQ1: What degree of completeness do the recovered microservice architectural models have?

[3] https://github.com/MicroServiceArchitectureRecovery/misar

[4] https://github.com/MicroServiceArchitectureRecovery/MiSAR-Parser-and-Model-Transformation

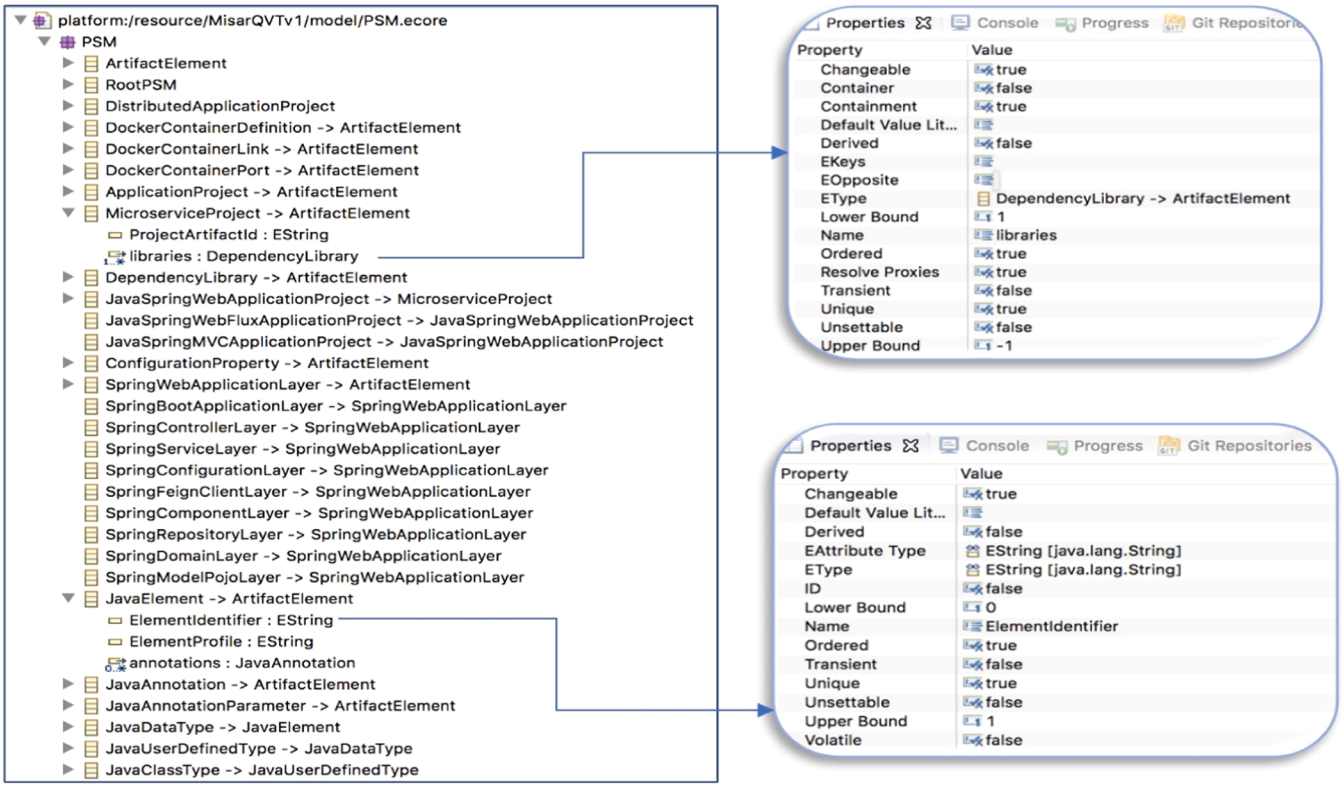


Fig. 7. Ecore implementation (XMI tree view) of MiSAR PSM.

```
// declaration section mapping DependencyLibrary::DependencyLibrary2Endpoint(uri: String):
Endpoint {
// body section // init section init { var endpoint := Endpoint.allInstances()-
   >selectOne(e |
         e.container().oclAsType(ServiceInterface).container().oclAsType(Microservice).
      MicroserviceName
         = self.ParentProjectName and e.RequestURI = uri); if endpoint
      <> null then { result := endpoint;} endif;
   }

   // 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;
}
// end section
```

Fig. 8. The QVTo code for mapping rule R6 to recover the Endpoint.

CRQ2: What degree of correctness do the recovered microservice architectural models have?

To answer CRQ1 and CRQ2, the total number of architectural elements recovered in the architectural model are compared to the total number of architectural elements of the actual (or expected) architecture. Recall is used to measure the completeness of the recovered architectural model and precision is used to measure the correctness of the recovered architectural model. The F-measure measures the overall accuracy of the recovered model [39] as follows:

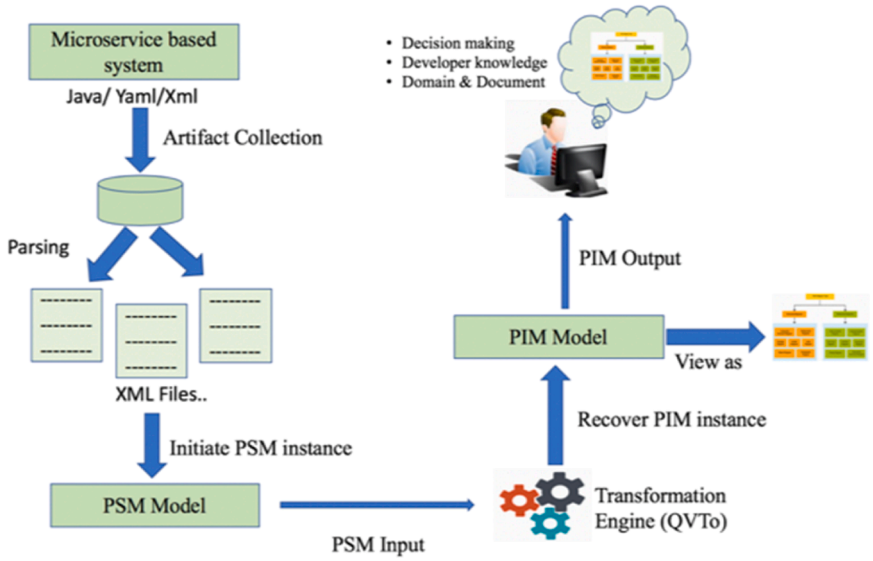


Fig. 9. Steps of the MiSAR architecture recovery process.

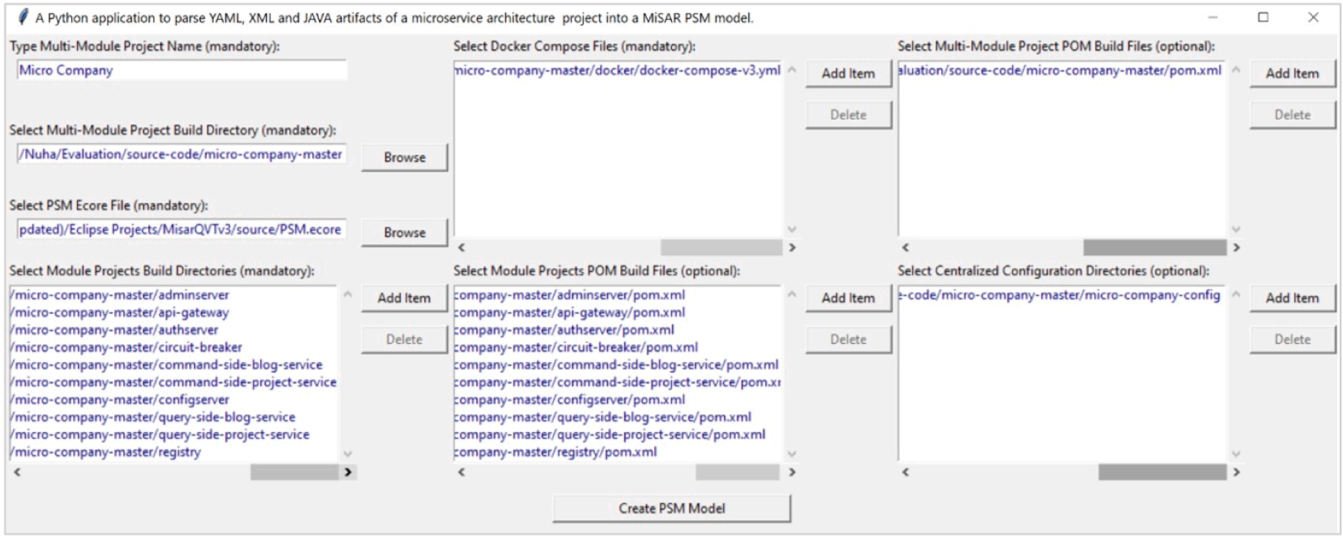


Fig. 10. Parser user interface to collect artefacts for MicroCompany application.

$$\text{Recall} = \frac{(TP)}{(TP + FN)} \tag{1}$$

$$\text{Precision} = \frac{(TP)}{(TP + FP)} \tag{2}$$

$$F_{Measure} = 2 \times \frac{\text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \tag{3}$$

Where TP is the number of True Positives which are the Correctly Recovered elements checking the source artefacts, FP is the number of False Positives which are Incorrectly Recovered elements or partially recovered elements and FN is the number of False Negatives which are the number of Missed Elements, i.e., the elements which are not recovered by MiSAR even though they existed in source artefacts or actual architecture.

The second evaluation is designed to compare the recovered architectural models with the available documentation, diagrams, textual description of each system provided by the developers. We call this comparison consistency check between the MiSAR Recovered Models and the Documentation. The objective of this consistency check is to evaluate MiSAR recovered models in comparison with the manual documentation of developers. To perform this check, we listed all elements and attributes of the MiSAR representation of the documentation and compared them against the ones in the recovered architectural model, which are presented in Section 8.5.

The third evaluation consists of measuring the efficiency of MiSAR in recovering semi-automatically architectural models in comparison with human manual recovery. To do this, we took the execution time in seconds for obtaining an architectural model with MiSAR.

The fourth evaluation involves comparing a MiSAR recovered architectural model with one recovered from another architecture recovery approach, Prophet [40]. The design and results are presented in Section 8.7.

8.2. Case selection

We carefully selected three microservice-based applications from [41] (MicroCompany, TrainTicket, and MusicStore) to evaluate MiSAR, ensuring diversity in programming languages, frameworks, and

architectural patterns. This selection allows us to assess MiSAR's capabilities and limitations across a variety of real-world microservice architectures. The selection was guided by the following well-defined criteria:

- System scale and complexity
  - TrainTicket: A large-scale benchmark system with 69 microservices (41 of which are business-oriented), making it significantly larger than most existing benchmarks.
  - MicroCompany (11 microservices) and MusicStore (9 microservices): Provide medium- and small-scale case systems, ensuring that MiSAR is evaluated across different system sizes.
- Diversity in microservice design and technology stack
  - The selected systems employ multiple programming languages, including Java, Node.js, Python, Go, and C#, ensuring MiSAR's ability to handle heterogeneous environments.
  - The systems are built using various microservice frameworks such as Spring Cloud, Express, Django, Webgo, and Steeltoe, allowing us to test MiSAR's adaptability to different microservice architectures.

Table 4 summarises the characteristics of the systems. i. MicroCompany application [42]: A Java Spring Boot/Spring Cloud microservice-based application with 11 microservices, 4 of which are business-oriented. It features both synchronous and asynchronous inter-service communication, making it a representative example of enterprise-level Java microservices. ii. TrainTicket application [43]: A large-scale microservice-based system where most microservices are developed using Java Spring Boot/Spring Cloud, but also includes 5 non-JVM-based microservices implemented in Node.js, Python, and Go. This selection allows us to evaluate MiSAR's handling of polyglot architectures. iii. MusicStore application [44]: A microservice-based system entirely implemented in C# using the Steeltoe framework. It consists of 9 microservices, 4 of which are business-oriented. This system provides insights into MiSAR's applicability beyond Java-based ecosystems.

Table 4
The characteristics of the selected systems.

Characteristic	System		
	MicroCompany	TrainTicket	MusicStore
Lines of code	127.1K	507.2K	116.6K
Date of the version used	October 2, 2022	Jan 19, 2020	October 2, 2022
Number of microservices	11	69	9
Functional	4	41	4
Infrastructural	7	28	5
Programming Languages	Java	Java, Node.js., Python, Go	C#
Frameworks	Spring Cloud	Spring Cloud, Express, Django, Webgo	Steeltoe
Variety technologies and implementations used in projects			
Supported by MiSAR:			
Containerization	Docker	Docker	Docker
DB	Spring Framework Mongo, MySQL, HSQLDB	Spring Framework Mongo, MySQL	MySQL
Discovery			Eureka
Circuit Breaker			Hystrix
Tracing			Zipkin
Communication	Synchronous and Asynchronous	Synchronous	Synchronous
	Spring Admin Sever CQRS & Event Sourcing – Spring Axon Spring Websocket	Unsupported by MiSAR: Node.js – Express Python – Django Go – Webgo Tracing – Jaeger	C# – ASP.Net Steeltoe Framework

8.3. The recovered architectural models of the systems

All the PIM instances for the 3 systems can be found on GitHub.[5] The generated (recovered) MicroCompany architecture consists of 11 microservices: 5 instances of Functional Microservices and 6 instances of Infrastructure Microservices. The generated (recovered) TrainTicket architecture consists of 69 microservices: 37 instances of Functional Microservices, 27 instances of Infrastructure Microservices and 5 instances of the supertype Microservice. The generated (recovered) MusicStore architecture consists of 9 microservices: zero instances of Functional Microservices, 2 instances of Infrastructure Microservices and 7 instances of the supertype Microservice.

It can be noticed that some microservices recovered were not classified as either functional or infrastructure e.g., the 7 instances of MusicStore. This indicates that MiSAR has managed to capture the existence of a certain microservice, but for some reason was not able to precisely recognize (classify) its type mainly because the implementation language and technologies used in these microservices are unsupported in MiSAR artefacts, i.e., MiSAR has no equivalent mapping rules for them.

MiSAR recovered models have different views, one at an architectural level and at a microservice level. The architectural level reflects the recovered PIM instance, as shown in Fig. 11 for MicroCompany, which includes the high-level view of all microservices of the architecture and their types (Infrastructure or Functional). At a microservice level, a more detailed view of a microservice is provided, which includes its Service Interface, Messages Destinations (e.g. Endpoint and/or Queue Listener), Service Messages, Service Operations, the Infrastructure Pattern Components of an individual microservice and their dependencies such as asynchronous/synchronous interactions between microservices. Fig. 12 shows the microservice level view for a functional microservice for MicroCompany "query-side-blog". As it can be noticed, the view includes its microservice container, 9 Infrastructure Pattern Components, 6 Infrastructure Client Components, its service interface with its endpoints and one Queue Listener and 10 service dependencies. In addition, the microservice view has the attributes for the microservices. Fig. 13 shows attributes for some elements of the microservice. For instance, (a) the "query-side-blog" microservice exposes an endpoint with request URI "GET /blogposts/search/findByDraftTrue" which is handled by (b) the service operation "findByDraftTrue()" and (c) returns a response service message of model "Page(BlogPost)". As it can be noticed, one of the attributes is "Generating PSM" which indicates which element from the PSM this element was generated from and provides traceability and backtracking support for the recovery.

The PIM recovered models are in XMI format and can be opened as tree views with Sample Reflective Ecore Model editor provided by the Eclipse Modeling Framework (EMF).

8.4. MiSAR's architectural model completeness and correctness

After we recovered the MiSAR PIM instances, i.e., the recovered MiSAR architectural models of MicroCompany, TrainTicket and MusicStore, we manually compared their MiSAR generated architectural models with their actual architectural elements as explained in Section 8.1. We created an excel sheet called Metric Analysis which includes a tab for each system on GitHub.[6]

To answer research questions CRQ1 and CRQ2 set in Section 8.1, we calculated recall, precision and F1-score metrics to measure,

[5] https://github.com/MicroServiceArchitectureRecovery/misar/tree/main/EmpiricalStudyReplication/EvaluationOf MiSAR

[6] https://github.com/MicroServiceArchitectureRecovery/misar/tree/main/EmpiricalStudyReplication/EvaluationOf MiSAR

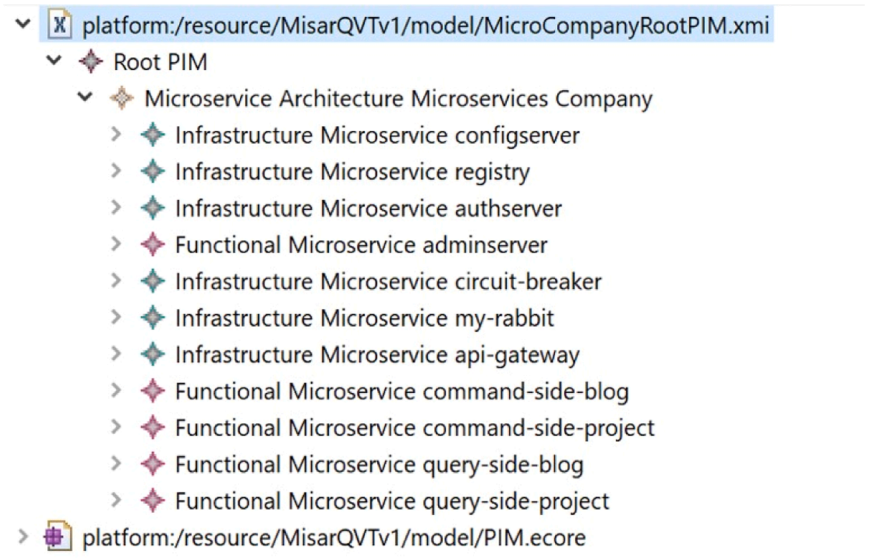


Fig. 11. PIM model for MicroCompany architectural view recovered by MiSAR.

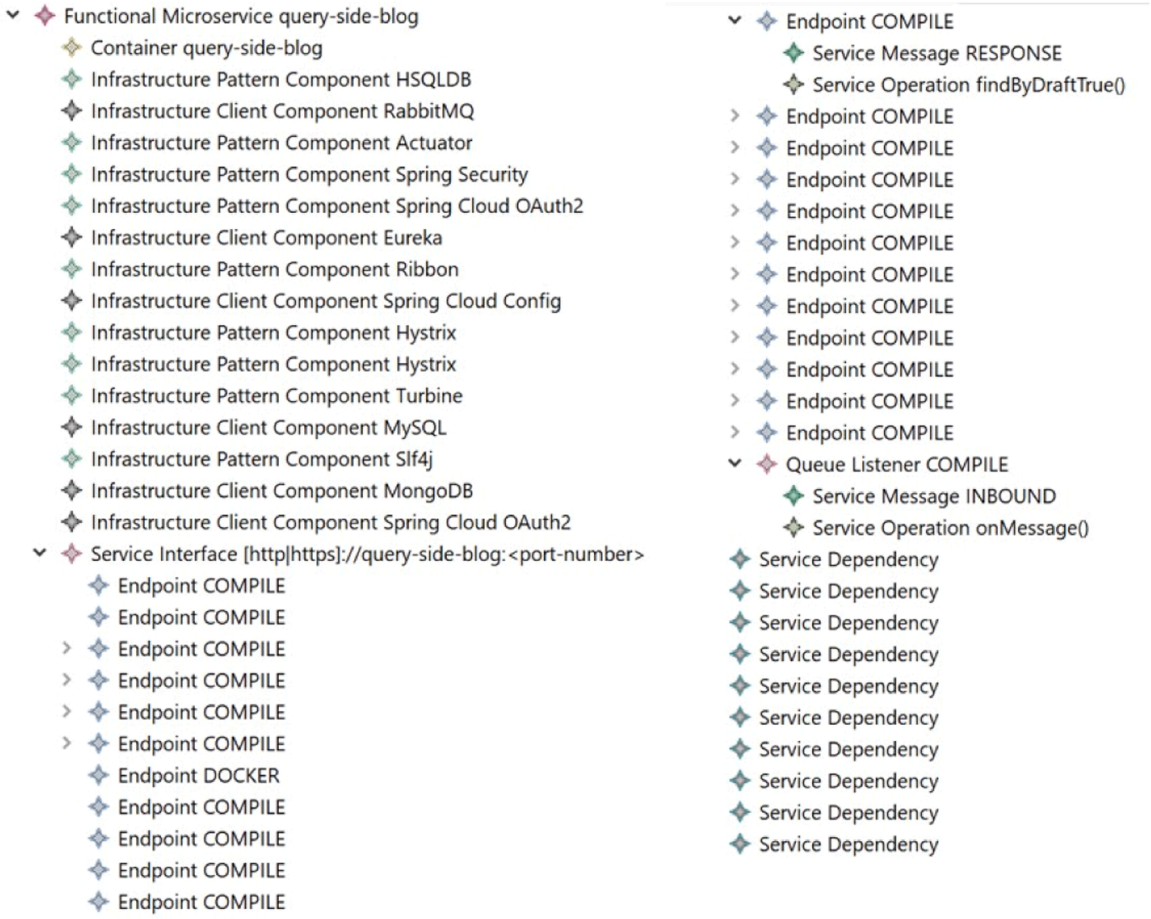


Fig. 12. Example of the recovered "query-side-blog" functional microservice instance in MicroCompany.

respectively, the completeness, correctness and overall accuracy of the recovered models, as shown in Table 5. These metrics were calculated for every recovered PIM element and the expected elements as shown in Table 5.

The highest overall effectiveness of MiSAR based on the F-measure is 94 % for the MicroCompany system. It also achieved a high precision score of 100 % of correct elements and a recall score which indicates that MiSAR has recovered 88 % of architectural elements. The recall for the MicroCompany was not 100 % due to MiSAR missed to recover the elements of "adminserver" and "api-gateway" as follows:

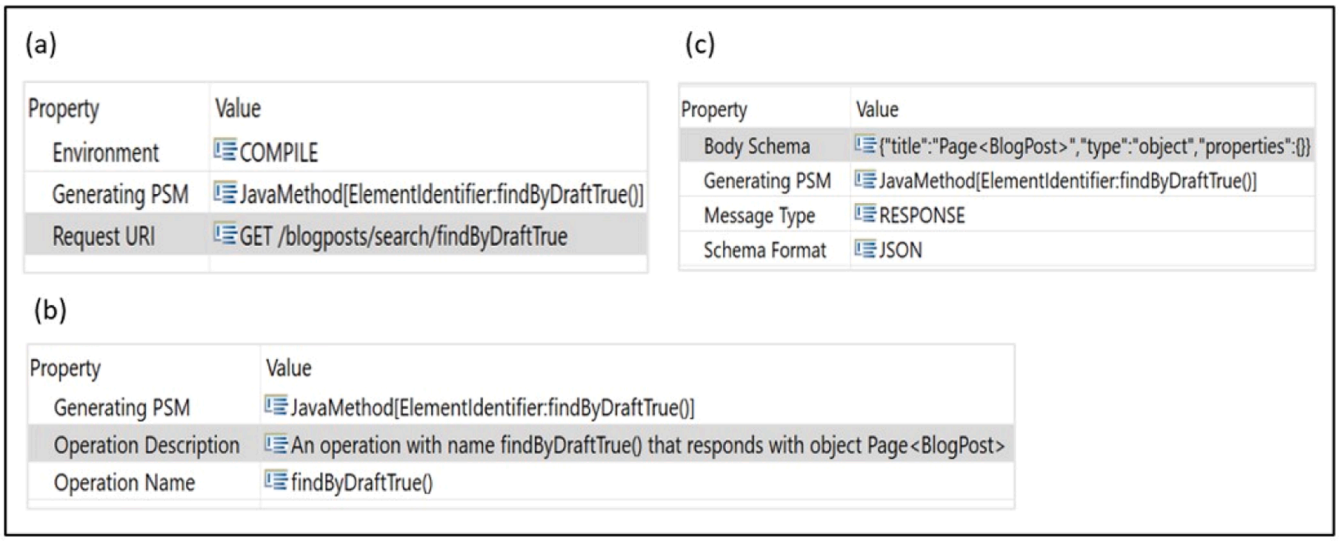


Fig. 13. Example of the attributes recovered for the "query-side-blog" microservice: a) one Endpoint, b) one Service Operation associated with a, c) one Service Message associated with b.

Table 5
Evaluation metrics for MiSAR recovery of systems.

		Correctly	Incorrectly				
	Total	Recovered	Recovered	Unrecovered			
PIM Element	Expected	TP	FP	FN	Recall	Precision	F-Measure
System: TrainTicket							
Container	69	69	0	0	100 %	100 %	100 %
InfrastructureMicroservice	30	27	3	0	100 %	90 %	94 %
FunctionalMicroservice	39	36	3	0	100 %	92 %	96 %
InfrastructureServerComponent	32	27	0	5	84 %	100 %	92 %
InfrastructureClientComponent	208	131	0	77	63 %	100 %	77 %
Endpoint	550	456	0	94	83 %	100 %	91 %
QueueListener	0	0	0	0	0.0 %	0.0 %	0.0 %
ServiceDependency	792	589	0	203	74 %	100 %	85 %
Total	1720	1335	6	379	78 %	100 %	87 %
System: MicroCompany							
Container	11	11	0	0	100 %	100 %	100 %
InfrastructureMicroservice	7	6	1	0	100 %	86 %	92 %
FunctionalMicroservice	4	4	0	0	100 %	100 %	100 %
InfrastructureServerComponent	7	6	0	1	86 %	100 %	92 %
InfrastructureClientComponent	90	89	0	1	99 %	100 %	99 %
Endpoint	148	144	0	4	97 %	100 %	99 %
QueueListener	7	3	0	4	44 %	100 %	60 %
ServiceDependency	282	226	0	56	80 %	100 %	89 %
Total	556	489	1	66	88 %	100 %	94 %
System: MusicStore							
Container	9	9	0	0	100 %	100 %	100 %
InfrastructureMicroservice	5	2	3	0	100 %	40 %	57 %
FunctionalMicroservice	4	0	4	0	0.0 %	0.0 %	0.0 %
InfrastructureServerComponent	5	2	0	3	40 %	100 %	57 %
InfrastructureClientComponent	35	26	0	9	74 %	100 %	85 %
Endpoint	80	27	0	53	34 %	100 %	51 %
QueueListener	0	0	0	0	0.0 %	0.0 %	0.0 %
ServiceDependency	104	34	0	70	33 %	100 %	49 %
Total	242	100	7	135	43 %	94 %	59 %

"adminserver" was recovered as a FunctionalMicroservice instead of a InfrastructureMicroservice and missed the recovery of the monitoring InfrastructureServerComponent along with all its 42 related ServiceDependencies. The "adminserver" is a monitoring infrastructure microservice that requests the health and metrics endpoints of all other microservices in the architecture. MiSAR was not able to analyze the Spring Admin Server used to implement the monitoring infrastructure of "adminserver", therefore these results.

"api-gateway" microservice was recovered with 3 missed QueueListeners along with their related ServiceOperations, 4 missed Endpoints and 1 missed message broker InfrastructurePatternComponent. The "api-gateway" is a gateway infrastructure microservice that listens to four events published onto the message broker by "command-sideblog" and "command-sideproject" functional microservices, then, it sends corresponding notifications to an internal WebSocket component. MicroCompany uses Spring Axon to implement event-driven CQRS pattern which

decorates the queue listener methods with @EventHandler annotation. MiSAR has no previous knowledge of this pattern and annotation yet. However, the high-level Axon's queue listeners invoke one low-level AMQP queue listener method decorated with a @RabbitListener annotation, which MiSAR already knows. Therefore, MiSAR was able to correctly recover that low-level QueueListener and its related ServiceOperation instead of the four higher-level Axon's listeners and their related ServiceOperations.

MusicStore system has the lowest overall effectiveness, based on its F-measure of 59 %. MiSAR also achieved a precision score of 94 % of correct elements. The recall score indicates that MiSAR has recovered 43 % of architectural elements. This low recall is due to the large number of missed elements, which in turn, is due to having all the microservices implemented with Steeltoe framework and C# language which MiSAR does not analyze yet. This caused MiSAR to incorrectly recover four FunctionalMicroservices and three InfrastructureMicroservices as abstract Microservices and miss most of their internal Endpoints and infrastructure components which rely on the transformation of methods written in C# instead of Java. On the other hand, Docker Compose, Eureka, Zipkin and Spring Config infrastructure artifacts utilized by Steeltoe assisted to recover the 43 % of the non-JVM MusicStore's architecture.

Finally, as for the TrainTicket system, the overall effectiveness of MiSAR based on the F-measure is 87 %. MiSAR also achieved a precision score of 100 % of correct elements. The recall score indicates that MiSAR has recovered 78 % 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.

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.

8.5. Consistency checking between the MiSAR recovered models and their documentation

In this section, we present the results for the consistency checks we performed for the MiSAR recovered architectural models and their documentation. The results are as follows (see Table 6 for the results of MicroCompany):

Consistent elements: They are the MiSAR recovered elements which are consistent with the documentation.

Discrepant elements: They are MiSAR recovered elements which are not in agreement with how they are represented in the documentation. For example, in the MicroCompany system, MiSAR recovered Spring Admin Server incorrectly as a FunctionalMicroservice instead of an InfrastructureMicroservice.

In addition, all FunctionalMicroservices in MusicStore system were incorrectly recovered as Microservices without identifying their type since MiSAR doesn't learn yet the C# language. In the TrainTicket system, MiSAR was able to recover the correct paths for "tsorder-service" endpoints even though they were incorrectly documented. These were checked manually in the source artefacts and it was discovered that the MiSAR recovered model had the correct representation.

Table 6
Results of the consistency check between the MiSAR's recovered architectural model of MicroCompany and its documentation.

PIM Element	(1)	(2)	(3)	(4)
Container	11	0	0	0
InfrastructureMicroservice	6	0	0	0
FunctionalMicroservice	4	1	0	0
InfrastructureServerComponent	6	0	1	0
InfrastructureClientComponent	29	0	1	59
Endpoint	5	0	0	139
QueueListener	3	0	0	0
ServiceOperation	0	0	0	19
ServiceMessage	0	0	0	43
ServiceDependency	36	0	5	77
Total	100	1	7	337

(1) Consistent. (2) Discrepant. (3) Absent. (4) Additional.

Absent elements: They are elements that exist in the documentation but are absent in MiSAR's recovered models. As explained in Section 8.4 this is due to that MiSAR still does not support several programming languages and technologies. For illustration, in MicroCompany system, MiSAR did not recover many of the elements of "adminserver" microservice. In the TrainTicket system, an example of an absent component is an InfrastructureClientComponent of category 'Service Routing Pattern Registry and Discovery', which according to the documentation implements Kubernetes (k8s), a technology that MiSAR currently does not support. This also applies to the MusicStore system since it is developed completely in C# language which is not yet MiSAR supported.

Additional elements: MiSAR recovered more architectural elements compared to the documentation. In particular, MiSAR was able to recover service operations and service messages of the documented endpoints, in addition to several infrastructure pattern components, infrastructure client components and service dependency elements that are not documented. For illustration, in MicroCompany system, MiSAR recovered 19 service operations and 43 service messages that do not exist at all in the documentation (but exist in the source artifacts) along with 139 endpoints more than the 5 endpoints that exist in the documentation as shown in Table 6.

8.6. The efficiency of MiSAR's semi-automation

Automated approaches for architecture recovery are valuable because they can handle large and complex systems more efficiently than manual methods, especially when they include diverse dependencies and components. Automated tools save time and provide a more accurate understanding of the architecture, especially when software engineers do not have previous knowledge and experience of a system and when dealing with large systems or systems with minimal documentation.

Table 7 shows the time it takes for each MiSAR toolset component, on an Intel Processor Core(TM) i5–7200 U CPU @ 2.50 GHz, 2701 Mhz, 2 Core(s), 4 Logical Processor(s) to create the PSM and PIM for the three open-source projects. It can be noticed that for a large project, such as

Table 7
Time of MiSAR toolset to obtain as-implemented architecture models.

	LOC	Parser to generate PSM (*sec*)	Model Engine to Transform PSM to PIM (*sec*)	Approx. Man Days
MicroCompany	127.1K	9	3.89	6
TrainTicket	507.2K	446	63.15	16
MusicStore	116.6K	1	1.07	4

TrainTicket, the parser takes most of the time of the recovery process. The last column shows the total man days it took for the manual architecture recovery of the projects. These man days reflect the time of two of the authors, who were not involved in the development of the projects. As it can be noticed the as-implemented architecture can be achieved through MiSAR in several seconds and several minutes (for large projects), whereas it can take several days in a manual process. However, the time spent on the manual recovery should be taken carefully considering that the effort is subjective and can depend on many factors which can include the knowledge and expertise of the team.

8.7. The comparative evaluation of MiSAR and Prophet

This section presents a comparative evaluation of MiSAR and Prophet [40], a software architecture recovery approach, designed for Java Spring microservice-based projects. The objective of this comparison is to assess the effectiveness and completeness of MiSAR in recovering architectural elements of a microservice system. The evaluation uses the MicroCompany system [42]. To measure completeness, correctness and overall accuracy of the recovered architectural models, we apply the metrics of recall, precision and F-measure.

8.7.1. Design

To enable a precise and structured comparison of the architecture recovery capabilities of MiSAR and Prophet, we developed a unified evaluation framework based on a curated set of Merged Architectural Elements. Table 8 illustrates the mapping process, showcasing how MiSAR and Prophet's elements contribute to the new merged model. This model bridges conceptual differences between the architectural elements identified by MiSAR and Prophet, ensuring a consistent and fair evaluation process. The merged elements were created by analyzing the key features of both approaches, identifying commonalities, and integrating complementary attributes. The process involved the following steps:

- Merging equivalent concepts under MiSAR's naming conventions: Elements that represent the same architectural concept in both approaches but differ in naming are merged using MiSAR's terminology. For example, MiSAR's MicroserviceArchitecture and Prophet's SystemContext were merged into the MicroserviceArchitecture element in the unified model.
- Preserving MiSAR's and Prophet's exclusive elements: Elements unique to MiSAR that do not have an equivalent in Prophet were retained using MiSAR's terminology and elements unique in Prophet were retained using Prophet's terminology. This ensures compatibility with the existing expected model. For example, MiSAR's QueueListener and InfrastructurePatternComponent were included directly, as Prophet lacks corresponding elements.
- Introducing new composite elements for complementary views: When MiSAR and Prophet provided distinct yet relevant perspectives, new merged elements were created to capture both. For example, MiSAR's ServiceMessage and Prophet's Entity and Field were merged to create ServiceEntity and ServiceEntityField, capturing both entity definitions and their field types.

8.7.2. Results

In terms of overall performance against the merged expected model, MiSAR demonstrated superior results, achieving an average recall of 86 %, precision of 99 %, and F-measure of 92 %. In contrast, Prophet yielded a significantly lower recall of 7 %, precision of 96 %, and F-measure of 13 %, as shown in Table 9. These differences highlight MiSAR's effectiveness in recovering a broad range of architectural elements, while Prophet's performance was limited to specific categories.

For structural elements (e.g., Containers, Microservices), MiSAR successfully recovered all Container, InfrastructureMicroservice, and InfrastructureClientComponent elements with 100 % recall. Prophet, however, failed to recover any of these deployment and infrastructure components (0 % recall). It was only able to recover 50 % of FunctionalMicroservice elements, particularly those whose Java source files contain classes annotated with @Entity. MiSAR's ability to analyse beyond java source files enabled the comprehensive identification of both functional and infrastructure elements, providing greater coverage of deployment architectural elements.

Regarding Endpoints, MiSAR achieved a recall of 97 %, correctly identifying 144 out of 148 expected entries. Prophet failed to recover any endpoints, resulting in 0 % recall for this category. This limitation is due to Prophet's reliance on detecting JAX-RS annotations, which were absent in the evaluated system. MiSAR, however, demonstrated the ability to detect endpoints defined using higher-level annotations, such as @RepositoryRestResource, which were prevalent in the MicroCompany system. Furthermore, Prophet does not support the recovery of asynchronous endpoints or queues, further limiting its endpoint detection capabilities. MiSAR's recovery of both synchronous and asynchronous endpoints highlights its strength in reconstructing API interactions.

As for Data Model Recovery, Prophet outperformed MiSAR, by achieving 100 % recall and precision in detecting both Entity and EntityField elements. MiSAR, by comparison, achieved 50 % recall for Entity and 60 % recall for EntityField. This disparity stems from MiSAR's focus on data objects exchanged through endpoints and queues, rather than standalone domain/entity classes. In the MicroCompany system, MiSAR successfully identified the BlogPost entity as a bodyschema: ServiceMessage in the query-side-blog microservice, as its usage was

Table 8
Merged architectural element model for comparative analysis.

Merged element	MiSAR's elements	Prophet's elements
MicroserviceArchitecture	MicroserviceArchitecture	SystemContext
Container	Container	–
InfrastructureMicroservice	InfrastructureMicroservice	–
FunctionalMicroservice	FunctionalMicroservice	Module
ServiceInterface	ServiceInterface	–
InfrastructureServerComponent	InfrastructureServerComponent	–
InfrastructureClientComponent	InfrastructureClientComponent	–
InfrastructurePatternComponent	InfrastructurePatternComponent	–
Endpoint	Endpoint	EndpointContext:(httpMethod,arguments)
ServiceOperation	ServiceOperation	EndpointContext:(method)
ServiceMessage	ServiceMessage	EndpointContext:(returnType)
QueueListener	QueueListener	–
ServiceDependency	ServiceDependency	MsEdge, MsLabel
Entity	Element's Attribute: BodySchema of ServiceMessage	Entity
EntityField	Element's Attribute: BodySchema of ServiceMessage	Field
EntityAssociation	–	MermaidEdge

Table 9
Evaluation metrics for MiSAR and Prophet recovery of MicroCompany.

	Expected	Correctly Recovered TP		Ncorrectly Recovered FP		Unrecovered FN		Recall		Precision		F-Measure	
Architecture Element	Merged	MiSAR	Prophet	MiSAR	Prophet	MiSAR	Prophet	MiSAR	Prophet	MiSAR	Prophet	MiSAR	Prophet
System: MicroCompany Container	11	11	0	0	0	0	11	100 %	0 %	100 %	0 %	100 %	0 %
InfrastructureMicroservice	7	6	0	1	0	0	7	100 %	0 %	86 %	0 %	92 %	0 %
FunctionalMicroservice	4	4	2	0	2	0	2	100 %	50 %	100 %	50 %	100 %	50 %
ServiceInterface	11	11	0	0	0	0	11	100 %	0 %	100 %	0 %	100 %	0 %
InfrastructureServerComponent	7	6	0	0	0	1	7	86 %	0 %	100 %	0 %	92 %	0 %
InfrastructureClientComponent	90	89	0	0	0	1	90	99 %	0 %	100 %	0 %	99 %	0 %
Endpoint	148	144	0	0	0	4	148	97 %	0 %	100 %	0 %	99 %	0 %
QueueListener	7	3	0	0	0	4	7	44 %	0 %	100 %	0 %	60 %	0 %
ServiceDependency	282	226	0	0	0	56	282	80 %	0 %	100 %	0 %	89 %	0 %
ServiceOperation	23	16	0	3	0	4	23	80 %	0 %	84 %	0 %	82 %	0 %
ServiceMessage	0	0	0	0	0	0	0	0 %	0 %	0 %	0 %	0 %	0 %
Entity	2	1	2	0	0	1	0	50 %	100 %	100 %	100 %	67 %	100 %
EntityField	40	24	40	0	0	16	0	60 %	100 %	100 %	100 %	75 %	100 %
EntityAssociation	0	0	0	0	0	0	0	0 %	0 %	0 %	0 %	0 %	0 %
Total	632	541	44	4	2	87	588	86 %	7 %	99 %	96 %	92 %	13 %

explicitly defined in the source code. However, MiSAR failed to recover the Project entity because its service operation was not explicitly defined in the source code, limiting MiSAR's ability to recover it.

9. Discussion

In this section, we discuss our findings. The discussion is organized as follows:

Architectural Expressiveness: In the three systems, MiSAR's recovered architectural model proved more expressive than the developers' documentation. Moreover, the comparative evaluation between Prophet and MiSAR demonstrated that MiSAR is capable of representing architectural elements absent in Prophet. This enhanced expressiveness results from MiSAR's PIM metamodel, which includes explicitly first class citizens for infrastructure pattern components, synchronous/asynchronous service interfaces and dependencies. For illustration, MiSAR identified additional architectural elements not documented by the MicroCompany team, such as information about data persistence, security client-side load balancer, circuit-breaker, metrics generation and logging patterns used in the microservice. It also provided information about the service operation names and the schema of request/response data message(s) for each endpoint as shown in Fig. 13. MiSAR was also able to recover the correct paths for "ts-order-service" endpoints even though they were incorrectly documented in TrainTicket. This advantage, along with its high precision, suggests MiSAR can offer comprehensive microservice architectural models of the implementation.

MiSAR reliability: Currently, MiSAR uses static analysis of source artefacts, which has proven been sufficient to recover the architectural concepts of the current metamodel. Unlike dynamic recovery approaches, such as using Zipkin, static analysis doesn't require the application to be running, avoiding issues with resource demands or bugs. Dynamic recovery also necessitates designing trace requests to capture the application's behavior, an effort eliminated by the static method, which recovers all architecture specifications as long as mapping rules exist. It can be noticed from Table 5 that MiSAR achieved high correctness (greater than 90 %) in three systems including the non-JVM MusicStore, supporting the reliability of MiSAR's static approach. The recall score is also high but lower than precision, as MiSAR recovered 88.1 %, 77.9 % and 42.6 % of existing architectural elements in MicroCompany, TrainTicket and MusicStore, respectively. These results are limited by the currently supported languages and technologies, which can be expanded in the future. Although most recovery approaches use dynamic analysis, MiSAR's static method successfully recovers service interactions and data structure. It has been observed that the mapping of the DockerContainerLink, ConfigurationProperty and JavaMethod PSM concepts into ServiceDependency PIM concepts each attributed by ProviderName and ProviderDestination are responsible for the successful recovery of the Service Dependency, synchronous Endpoints, asynchronous Queue Listeners and ServiceMessage PIM concepts.

The Ability of MiSAR to Discover the Existence of Non-JVM Applications: Although MiSAR was initially designed for Java applications using the Spring Boot/Spring Cloud frameworks, it demonstrated the capability to identify non-JVM microservices. For example, in the MusicStore system all functional microservices are developed with Steeltoe framework and C#. Similarly, in Trainticket, "ts-voucher-service" is developed with Python language, "ts-news-service" with Go language, "ts-ticket-office-service" with Node.js and "ts-ui-dashboard" with JavaScript as NGINX proxy. MiSAR managed to capture the existence of those non-JVM microservices by recovering the elements of Container, Microservice and ServiceInterface from the Docker Compose files 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 contrast, Prophet's capabilities are limited to Java based systems that use JAX-RS annotations for identifying architectural

components. As a result, Prophet failed to recover any information for endpoints and infrastructural elements lacking support for cross-language recovery or the analysis of Docker- or build-based configurations. Furthermore, even in JVM-based services, Prophet's reliance on specific Java annotations restricted its coverage to a narrow set of structural elements. In addition, more elements were recovered for non-JVM microservices in MusicStore, such as infrastructure pattern components and endpoints thanks to MiSAR mapping rules and PSM configuration artifacts that were analyzed based on Spring Boot/Spring Cloud framework. These findings underscore MiSAR's greater flexibility and broader applicability in heterogeneous microservice environments compared to Prophet.

MiSAR support for Traceability and Backtracking: MiSAR mapping rules are implemented with Eclipse QVTo, which accomplishes model traceability by means of the resolve() function. This traceabilitys can allow software engineers to analyze the orders in which mapping rules have been invoked and, at any point of the recovery process, retrieve the elements previously recovered in order to make updates on their values and/relations without re-executing the rules. In order to check the validity of the recovered elements, especially in the case of generating 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. 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.

MiSAR Extendability: Microservice-based architectures are implemented with a variety of emerging technologies and patterns. The MiSAR Platform Independent Metamodel is independent of any technology and abstracts microservice elements without implementation details. It also includes abstract concepts like the Message Destination which abstracts communication types: the Endpoint concept representing synchronous communication and the Queue representing asynchronous communication. If a new communication type emerges in the microservice area, they can be added as subtypes, allowing easy future extensions. In terms of patterns, MiSAR currently supports five categories of microservice patterns. For supporting new emerging patterns, these can be added into the Infrastructure Pattern Category enumeration list in Fig. 6). To illustrate, in MicroCompany system, "api-gateway" implements WebSocket pattern and WebSocket endpoints which MiSAR does not currently support its technologies. The WebSocket pattern can be represented using the existing InfrastructurePatternComponent PIM concept with InfrastructurePatternCategory as "Development Pattern Asynchronous Message Brokering" or by appending a new category in the enumeration type InfrastructurePatternCategory. To support the recovery, only the Platform-Specific Metamodel, the related mapping rules and parser should be extended for new physical and implementation technologies. For example, the parser will need to incorporate new libraries for parsing the new technologies and abstract them to PSM elements.

Empirically deriving a MDA Recovery Approach: In the literature, MDA and architecture recovery approaches are often empirically evaluated, but few are derived empirically. The benefits for deriving empirically our approach include: 1) Researchers and practitioners can trace the rationale for our artefacts. This can aid researchers in extending and reusing MiSAR artefacts and practitioners in making decisions in the adoption of our approach and understanding the architectural models produced. 2) Researchers and MDA and architecture recovery providers can repeat our steps and learn how to derive new approaches. The main pitfalls for deriving empirically such an approach are that it is 1) Time consuming. However, the artefacts are expressive and reflect real systems. 2) Empirical Derivation can be influenced by an existing PIM metamodel: The research design we followed chose to enhance an existing PIM metamodel. To make our results more generic and applicable to other PIM metamodels, we created the "Requirements" which researchers and providers can apply to enhance other microservice PIM metamodels.

Microservice Architecture Recovery Benefits to Software Engineers: Our evaluation demonstrates that MiSAR's semi-automatic recovery generates architectural models with more architectural elements than the developer documentation, identifying inconsistencies and missing elements such as Service Dependencies in the documentation. MiSAR can help software engineers in obtaining an up-to-date microservice architecture documentation either while developing or after system development. In addition, MiSAR automatically recovers models in seconds. The only step which is semi-automatic is the selection of the files of a project to be parsed. This allows users to customize and control the scope of the recovery process. This is crucial for teams using different technologies and standards, as they may lack the expertise to analyze unfamiliar microservices. It is important to highlight a current limitation of MiSAR: the absence of automatic change propagation. Our approach does not detect changes of source level artefacts such as Docker Compose files or configuration files (at M0) which could provoke changes to architectural models. As a result, users must manually re-run the MiSAR toolset and initiate the recovery process whenever source-level changes occur in order to update the architectural model. This manual intervention increases the overall effort required from users, particularly as the system evolves over time.

9.1. Threats to validity

Internal threats of validity concern factors that impact the integrity of the study results. There are four major threats to the internal validity of the empirical study.

The absence of applicable concepts: The first one lies in the absence of applicable concepts. Activities 1 and 2 were conducted manually (extracted, compiled and analyzed), which implies that there could be unintentionally a few concepts that are missed out. To mitigate this threat, we semi-automatically recovered the architectural model of 3 systems and compared the recovered models with their actual architecture and their documented ones.

The divergence of documentation from the actual implementation: In the evaluation using the three systems, we had to identify the expected architectural elements to be able to measure precision, recall and accuracy. We initially wanted to use the architectural elements of the documentation. However, usually, the documentation published by the developers diverges from the actual implementation, to mitigate this threat: 1) we manually created the actual architecture of the system from the source implementation of the systems. Therefore, we were aware of cases of divergence and used the source artefacts as the expected elements; 2) for every architectural element recovered by MiSAR we also checked that it existed in the source artefacts.

Bias in the data extraction during the manual recovery: During the extraction of the elements from the source during Activities A1 and A2, and in the evaluation we had to perform manual recovery. This led to encountering different interpretations in the analysis due to authors' bias. To mitigate this threat, the data extraction process was conducted by two authors. The first author acted as the data extractor and the second author as the data checker. Any disagreements among all the authors were resolved by the third author through discussions.

The errors in the source code: Bugs were found in the source code of the system. Even though the static recovery process can still run if there are bugs in the source code, the resultant model might represent some incorrect information. To mitigate this threat, we checked that these bugs are not affecting the architecture.

Threats to external validity are related to the generalizability of findings: (a) Mapping rules and architectural concepts were derived from a limited number of systems, which may limit their applicability to a broader range of architectures. If the selected systems do not fully represent the diversity of real-world microservice implementations, the

proposed approach may not generalize well. To address this threat, the initial artefacts were developed using eight systems, and in this study, they were incrementally refined with the analysis of an additional nine systems. The nine systems were carefully selected to represent fundamental and best practices in microservice-based architectures. Furthermore, these systems were chosen based on the availability of rich architectural documentation and illustrative diagrams, ensuring sufficient detail for meaningful analysis. Conceptual saturation was achieved by identifying recurring microservice types, service dependencies, and widely adopted architectural patterns. Additionally, the MiSAR Metamodel was designed with extensibility, enabling future adaptation to emerging architectural styles and technologies. The three systems used for evaluation were carefully chosen to cover different programming languages, frameworks, and architectural patterns, reducing bias towards a specific technology stack. (b) The current scope of MiSAR effectively recovers the architecture of microservice-based applications that are developed using Java and Spring Boot/Spring Cloud frameworks. However, since MiSAR currently only supports static analysis, projects developed with unconventional implementations or developer-specific logic may be missed during the recovery process. Additional evaluations are required to address cases with such logic and further improve MiSAR's applicability.

10. Related work

Although significant software architecture recovery methods exist [3,4,45,46], few of the current methods have mainly focused on a system that specifically addresses MSA.

One of the few existing works related to ours is MicroART [11] as shown in Table 10. MicroART is a MSA recovery approach similar to MiSAR, as it uses model-driven engineering principles. There are several differences between MiSAR and MicroART as follows:1) The PIM metamodel of MiSAR is more expressive as it includes explicitly more microservice concepts. MicroART only has 8 concepts, which includes Developer, Team, Product, and lightweight communication whereas MiSAR has 17 concepts. Having an expressive PIM metamodel allows for the recovery of more aspects of the microservice system such as asynchronous communication and infrastructure patterns. 2) MiSAR was

Table 10
Comparison of microservice architecture recovery approaches.

	MiSAR	MicroArt	MicroTOM	Prophet	Microlyze
Year	2020	2017	2023	2022	2018
Tool Support	Yes	Yes	Yes	Yes	Yes
Analysis/ Transformation Software Analysis	Static	Hybrid (Static & Dynamic)	Hybrid (Static & Dynamic)	Static	Dynamic
Method(S)	– Text-To-Model (output: PSM) – Model-To-Model, (output: PIM)	Text-To-Model (output: MicroArt-DSL) Model-To-Model (output: Refined Microart-DSL)	Text-To-Model (output: MicroTOSCA2) Model-To-Model (output: Refined MicroTOSCA)	Text-To-Model (output: Context Map) Text-To-Model (output: Communication Diagram)	– Text-To-Model (output: Adjacency Matrix)
Automation	Semi-Automated	Semi-Automated	Automated	Automated	Semi-Automated
Input Technology / Framework Support	– Docker / Docker Compose, Maven/Gradle, Spring Cloud/ Netflix OSS, Spring Boot/Spring Cloud	– Docker, TCPdump	Kubernetes, Istio/Kiali	– Java Spring	– Eureka / Consul, Zipkin
Artefact(S)	– Docker Compose File (YAML) / Dockerfile, POM File (XML) / Gradle Files,Configuration Files (YAML), Source Files (Java)	Github Repository Docker Compose File (YAML) / Dockerfile Tracing Logs	– Kubernetes Manifest Files (YAML) – Kiali Graph File (JSON)	– Source Files (Java)	– Source Files (Java)
Outputs/Architecture Ele Artefact(S)	Ments Platform Independent Model (PIM) File (XMI)	Refined MicroArt-DSL	MicroTOSCA Refined Topology Graph (YAML)	Context Map (JSON) Communication Diagram (JSON)	Refined Adjacency Matrix
Functional Microservices	Yes	Yes	Yes	Yes	Yes
Infrastructure Microservices	Yes	Yes	Yes	No	Yes
Endpoint (Synchronous Service Interface)	Yes	Yes	No	Yes	Yes
Queue Listener (Asynchronous Service Interface)	Yes	No	No	No	No
Service Operation	Yes	No	No	Yes	Yes
Service dependency (Synchronous (HTTP) Communication)	Yes (Endpoint-Level)	Yes (Endpoint-Level)	Yes (microservice-level)	Yes	Yes
Service dependency (Asynchronous Communication)	Yes (Queue-Level)	No	Yes (microservice-level)	No	No
Container	Yes	No	No	No	Yes
Infrastructure Server Component	Yes	No	No	No	No
Infrastructure Client Component	Yes	No	No	No	No
Infrastructure (API Gateway / Proxy)	Yes	Yes	Yes	No	Yes
Infrastructure (Message Brokering)	Yes	Yes	Yes	No	No
Infrastructure (Circuit Breaker)	Yes	No	Yes	No	No
Infrastructure (Security)	Yes	Yes	No	No	No
Infrastructure (Tracing / Monitoring)	Yes	Yes	No	No	No

developed empirically using an example-based approach which incrementally developed its artefacts (PIM metamodel, mapping rules) by analysing existing microservice systems. MicroART was not developed in a systematic approach. It is based on the authors interpretation of the needs and characteristics in [7]. 3) MiSAR followed an MDA approach and has used the OMG standards such as automatic Model Transformations implemented in QVTo and includes a PSM to represent the specifics of technologies, whereas MicroART uses JAVA to implement mapping rules. 4) MiSAR purely uses static analysis to analyse existing systems. MicroART uses static analysis to obtain knowledge of the system and developers, and uses dynamic analysis to obtain knowledge to create the architecture. 5) The architectural model is recovered automatically from the PSM in MiSAR, i.e., there is no human intervention whereas in MicroART a software architect needs to identify service discovery services. The latter can be due to the dynamic analysis nature or to the fact that the MicroART PIM metamodel does not have an explicit concept for asynchronous communication and associated mapping rules.

Another microservices architecture recovery approach is Microlyze, by Kleehaus et al. [12]. Unlike MiSAR, MicroLyze does not adopt a model-driven approach. Instead, it utilizes a distributed tracing component that dynamically monitors simulated user requests. In terms of synchronous service interface and synchronous service dependency, MicroLyze is similar to MiSAR, where MicroLyze illustrates intra-relationships among microservices via the adjacency matrix it produces as output. However, MicroLyze does not recover information about the queue-based service dependency nor the queue-based service interface of each microservice, as shown in Table 10. In MiSAR, this information is recovered as Queue Listener and Service Dependency concepts. In addition, compared to MiSAR, MicroLyze lacks recovery of various infrastructure features, as shown in Table 10.

The microTOM approach [47] automatically analyses microservice-based applications by transforming deployment artefacts into a model following the microTOSCA metamodel. Differently to MiSAR, microTOM uses both dynamic and static analysis but only is able to recover synchronous and asynchronous communication at microservicelevel. MiSAR is able to recover both synchronous and asynchronous communication at the endpoint and queue level. It also does not fully support deployment and infrastructure features like security and monitoring as shown in Table 10.

Prophet [40] mainly focuses on the architecture extraction by using automatic static analysis and transforming Java source and bytecode artefacts into two separate models: one for a communication view and another for a domain view. Like MiSAR, Prophet can analyze Java Spring microservice-based projects. However, unlike MiSAR, Prophet does not analyze other source artefacts such as deployment, build, and configuration files. In contrast, MiSAR emphasizes the recovery of high-level architectural elements, including both synchronous and asynchronous communication patterns and infrastructure components. While Prophet's code-centric approach allows it to accurately identify domain entities, MiSAR offers a broader perspective by employing a Model-Driven Architecture (MDA) approach. This approach integrates various artefacts—such as Java classes, Spring annotations, Docker Compose files, and configuration descriptors—and transforms them into platform-independent models (PIMs). Through rule-based model-to-model transformations guided by metamodels, MiSAR abstracts low-level implementation details into rich architectural concepts. As a result, MiSAR provides significantly more comprehensive architectural coverage than Prophet, particularly in heterogeneous microservice systems where code alone may not reveal the full architectural context.

As presented in Table 10, MiSAR is the microservice architecture recovery approach that currently supports more technologies. In addition, it is the approach that has more architecture expressiveness as it recovers more architecture elements. This means that the as-implemented architectures it recovers are more complete than the other approaches. To the best of our knowledge, there is a lack of empirical based approaches in MDE and architecture recovery. MiSAR is the only microservice-oriented static SAR tool that can reveal the service interaction of an architecture as well as the structure of service data, the Body Schema attribute of the Service Message. Furthermore, in architecture recovery, Misar makes a clear distinction between functional and infrastructure microservices. Specifically, for infrastructure patterns, MiSAR also categorizes them based on whether they act as servers or clients. MiSAR considers them as first-class elements in the architecture PIM, a feature absent in other approaches. This permits infrastructure microservices to have many patterns.

11. Conclusion & future directions

In this work, we present an in-depth empirical investigation into microservice-based systems for defining requirements to include in a MSA recovery approach. Through this study, we formalized MiSAR with the aim of recovering microservice architectures. MiSAR provides semi-automatic architecture recovery by implementing MDA artefacts (QVTo transformations and ECore metamodels). At the moment, MiSAR recovers architectures of microservice based systems implemented in Java by the Spring Boot/Spring Cloud framework. The main benefit in adopting MiSAR is allowing software engineers to obtain an up-to-date architectural model which provides a view of the underlying structure of their microservice systems, and thus models can be later used for many purposes, such as documentation, obtaining system's knowledge, architectural analysis, maintenance and impact analysis between the implemented architecture and the designed one. For future research, we plan to use MiSAR in an industrial setting. We have planned an empirical study that would allow us to obtain feedback from practitioners on the usefulness of the architectural models recovered and the user-friendliness of the tool. In addition, the Python Software Foundation has recently funded MiSAR to extend its parser and platform specific model to support python based microservice projects. We also plan to extend MiSAR to support architecture conformance checking and forward engineering. Ensuring consistency between the Platform-Independent Model (M2) and the Platform-Specific Model (M1) is a significant concern in model-driven engineering. Future extensions of MiSAR could investigate mechanisms to maintain consistency across abstraction levels—such as automated validation techniques or bidirectional model transformations in order to support system evolution. This could be achieved by implementing the mapping rules using QVT-R (QVT-Relations), which supports bidirectional transformations, instead of QVTo, enabling synchronization between system artefacts and architectural models.

CRediT authorship contribution statement

Nuha Alshuqayran: Writing – review & editing, Writing – original draft, Software, Project administration, Formal analysis, Data curation. **Nour Ali:** Writing – review & editing, Writing – original draft, Supervision, Formal analysis, Data curation. **Roger Evans:** Writing – review & editing, Writing – original draft, Supervision, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data and code available on github

References

[1] D. Garlan, Software architecture: a roadmap, in: Proceedings of the Conference on the Future of Software Engineering, 2000, pp. 91–101.
[2] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley Professional, 2003.
[3] N. Ali, S. Baker, R. O'Crowley, S. Herold, J. Buckley, Architecture consistency: state of the practice, challenges and requirements, Empir. Softw. Eng. 23 (1) (2018) 224–258.
[4] S. Ducasse, D. Pollet, Software architecture reconstruction: a process-oriented taxonomy, IEEE Trans. Softw. Eng. 35 (4) (2009) 573–591.
[5] C. Pahl, P. Jamshidi, O. Zimmermann, Architectural principles for cloud software, ACM Trans. Internet Technol. (TOIT) 18 (2) (2018) 1–23.
[6] S. Newman, Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, Inc, 2021.
[7] J. Lewis, M. Fowler, "Microservices, Available: http://martinfowler.com/articles/microservices.html, [Accessed: 10-Aug2019] (2014).
[8] Di Francesco, P., Malavolta, I. and Lago, Research on architecting microservices: trends, focus, and potential for industrial adoption, In 2017 IEEE International Conference on Software Architecture (ICSA), 2017, pp. 21–30.
[9] N. Alshuqayran, N. Ali, R. Evans, A systematic mapping study in microservice architecture, in: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), IEEE, 2016, pp. 44–51.
[10] N. Alshuqayran, N. Ali, R. Evans, Towards Micro Service Architecture Recovery: An Empirical Study, IEEE, 2018, pp. 47–4709.
[11] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, A. Di Salle, Microart: a software architecture recovery tool for maintaining microservice-based systems, IEEE International Conference on Software Architecture Workshops (ICSAW), 2017, pp. 298–302.
[12] M. Kleehaus, O. Uluda ̈g, P. Sch ̈afer, F. Matthes, MICROLYZE: a framework for recovering the software architecture in microservice-based ̈environments. Information Systems in the Big Data Era: CAiSE Forum 2018, Springer, Tallinn, Estonia, 2018, pp. 148–162. June 11-15, 2018, Proceedings 30.
[13] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, Morgan & Claypool, 2017.
[14] P.J. Barendrecht, Modeling Transformations Using QVT Operational Mappings, Eindhoven University of Technology Department of Mechanical Engineering Systems Engineering Group, Research project report, Eindhoven.
[15] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: eclipse modeling framework (2008).
[16] B. Combemale, R.B. France, J.-M. Jez ́equel, B. Rumpe, J. Steel, D. Vojtisek, Engineering Modeling Languages: Turning Domain Knowledge' Into Tools, 1st Ed., CRC Press, 2016.
[17] C. Raibulet, F.A. Fontana, M. Zanoni, Model-driven reverse engineering approaches: a systematic literature review, IEEE Access 5 (2017) 14516–14542.
[18] D. Akehurst, S. Kent, A relational approach to defining transformations in a metamodel 5 (2002) 243–258.
[19] O.M. Group, MDA Guide revision 2.0, Available: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01, [Accessed: 01-Jan-2019] (2014).
[20] N. Ali, R. Nellipaiappan, R. Chandran, M.A. Babar, Model driven support for the service oriented architecture modeling language, in: Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems, Association for Computing Machinery, 2010, pp. 8–14.
[21] N. Ali, M.A. Babar, Modeling service oriented architectures of mobile applications by extending soaml with ambients, in: 2009 35th Euromicro Conference on Software Engineering and Advanced Applications, IEEE, 2009, pp. 442–449.
[22] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, J. Rosenberg, Preliminary guidelines for empirical research in software engineering, IEEE Trans. Softw. Eng. 28 (8) (2002) 721–734, https://doi.org/10.1109/TSE.2002.1027796.
[23] J. Lopez-Fern andez, J. Cuadrado, E. Guerra, J. De Lara, Example-Driven Meta-Model Development, Springer, 2015, pp. 1323–1347.
[24] O. Nierstrasz, M. Kobel, T. Girba, M. Lanza, Example-driven reconstruction of software models, in: European Conference on Software Maintenance and Reengineering (CSMR'07), IEEE., 2007, pp. 275–286.
[25] M. Vidoni, A systematic process for mining software repositories: results from a systematic literature review, Inf. Softw. Technol. 144 (2022) 106791, https://doi.org/10.1016/j.infsof.2021.106791.
[26] F.A. Barbeiro Campos, Spring-netflix-oss-microservices-master, Available: https://github.com/fernandoabcampos/spring-netflix-oss-microservices, [Accessed: 05-Sep-2019] (2019).
[27] J. Hecht, Spring-rabbitmq-messaging-microservices, Available: https://github.com/jonashackt/spring-rabbitmq-messaging-microservices, [Accessed: 05-Sep-2019] (2019).
[28] Sergeikh, Cloud-enabled-microservice, Available: https://github.com/sergeikh/cloud-enabled-microservice, [Accessed: 05-Sep-2019] (2019).
[29] K. Bastani, Event-sourcing-microservices, Available: https://github.com/kbastani/event-sourcing-microservices-example, [Accessed: 05-Sep-2019] (2019).
[30] B. Arivazhagan, Spring-cloud-sidecar-polygot, Available: https://github.com/BarathArivazhagan/spring-cloud-sidecar-polygot, [Accessed: 05-Sep-2019] (2019).
[31] A. Allewar, Microservices-basics-spring-boot, Available: https://github.com/anilallewar/microservices-basics-spring-boot, [Accessed: 05-Sep-2019] (2019).
[32] K. Bastani, Spring-cloud-event-sourcing, Available: https://github.com/kbastani/spring-cloud-event-sourcing-example, [Accessed: 05-Sep-2019] (2019).
[33] K. Bastani, Spring-boot-graph-processing-example, Available: https://github.com/kbastani/spring-boot-graph-processing-example, [Accessed: 09-Dec-2019] (2019).
[34] D. Reddy, Bookstore-consul-discovery, Available: https://github.com/devdcores/BookStoreApp-Distributed-Application, [Accessed: 14-Jan-2020] (2019).
[35] J. Carnell, I. Sanchez, Spring microservices in action. Simon and Schuster, 2021, 2nd Ed., Microservices patterns: with examples in Java, Simon and Schuster, Richardson, 2019.
[36] C. Richardson. Microservices patterns: with examples in Java, Simon and Schuster, 2019.
[37] P. Brereton, B. Kitchenham, D. Budgen, Z. Li, Using a protocol template for case study planning, in: In 12th International Conference on Evaluation and Assessment in Software Engineering (EASE), 2008, pp. 1–8.
[38] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, A. Wessl ́en, Experimentation in software engineering, Springer Science & Business ́Media, 2012.C. Manning, P. Raghavan, H. Schutze, Introduction to Information Retrieval, Cambridge University Press, Cambridge, 2008.
[39] C. Manning, P. Raghavan, H. Schutze. Introduction to information retrieval, Cambridge University Press, Cambridge, 2008.
[40] V. Bushong, D. Das, T. Cerny, Reconstructing the holistic architecture of microservice systems using static analysis, in: Proceedings of the 12th International Conference on Cloud Computing and Services Science-CLOSER, 2022.
[41] D. Taibi, Microservices Project List, Available: https://github.com/davidetaibi/Microservices_Project_List, [Accessed: 02-August-2022] (2021).
[42] I. Dugalic, MicroCompany, Available: https://github.com/idugalic/micro-company, [Accessed: 02-September-2022] (2022).
[43] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, W. Zhao, Benchmarking microservice systems for software engineering research, in: Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings - ICSE (2018) 323–324.
[44] S. OSS, MusicStore, Available: https://github.com/SteeltoeOSS/Samples/tree/main/MusicStore, [Accessed: 02-September2022] (2022).
[45] J. Buckley, N. Ali, J. English, M.and Rosik, S. Herold, Real-Time Reflexion Modelling in architecture reconciliation: A multi case study, Information and Software Technology 61 (2015) 107–123.
[46] I. Pashov, M. Riebisch, Using feature modeling for program comprehension and software architecture recovery. in: Proceedings. 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, IEEE, 2004, pp. 406–417.
[47] J. Soldani, J. Khalili, A. Brogi, Offline Mining of Microservice-based Architectures, SN Computer Science (2023) 63–73, https://doi.org/10.1007/S42979-023-017214.