# Towards Micro Service Architecture Recovery: An Empirical Study

Nuha Alshuqayran
*Computing Engineering and Mathematics*
*University of Brighton*
*Brighton, UK*
*Email: n.alshuqayran@brighton.ac.uk*

Nour Ali
*Computer Science Department*
*Brunel University London*
*Uxbridge, UK*
*Email: Nour.Ali@brunel.ac.uk*

Roger Evans
*Computing Engineering and Mathematics*
*University of Brighton*
*Brighton, UK*
*Email: r.p.evans@brighton.ac.uk*

***Abstract*—Micro service architectures are rapidly establishing themselves in the software industry as a more efficient and effective substitute for monolithic applications. In a micro service architecture, the application is broken down into many small elements called micro services. These are managed in a distributed way and typically involve several development teams. In such an environment, an architectural model can get lost along the way, making it difficult to perform many downstream software engineering tasks, such as migration, audit, integration or impact analysis. To address this problem, we are developing support for Micro Service Architecture Recovery (MiSAR) using a Model Driven Engineering approach. In this paper, we describe an empirical study which aims to identify the core elements of our approach, by undertaking manual analysis on 8 micro service-based open source projects. From this analysis, we define a metamodel for micro service-based architectures and a set of mapping rules which map between the software and the metamodel. The resulting metamodel and mapping rules provide a solid foundation for any micro service architecture recovery approach and hence are a key first step towards managing the architectural integrity of micro service-based applications.**

***Keywords*-micro service architecture, model driven engineering, software architecture, software architecture recovery, reverse engineering;**

## I. Introduction

In the dynamic environment of today's world, software applications need to keep up with the fast pace of development and be as agile as possible. Applications need to accommodate ever-changing business needs and a diverse range of clients, such as desktop and mobile browsers and native mobile applications, as well as third parties through APIs. It is very difficult to fulfill these requirements by using monolithic applications. This has led to an architectural shift from a monolithic to a "micro service" architectural style [1].

The Micro Service Architecture (MSA) style is emerging as a new way to structure applications. The key characteristics of MSA are that it is modular and distributed [2]. MSA has evolved from the traditional service oriented architectural style and it is now being deployed to provide cohesive business functions. In order to deploy such cohesiveness within a business organization, the system is divided into small services and as Fowler and Lewis comment [3], this allows fragmentation to become a dominant architectural style.

Generally, with the sophistication and complexities of such evolving and dynamic systems, the architectural model can get lost/drift along the way, as micro service identities and dependencies become less precise. The lack of a clear architectural model makes it difficult to perform many downstream software engineering tasks, such as migration, audit, integration or impact analysis. A possible solution to this problem is to allow software developers and maintainers to conduct architecture recovery, a technique which has been widely supported in object-oriented systems [4], [5], but to date has not been explored for micro service-based systems.

To address this, we are developing an approach called Micro Service Architecture Recovery (MiSAR) that uses a Model Driven Engineering (MDE) [6] approach to generate architectural models of micro service-based systems. Two key components of MiSAR are a metamodel, which abstracts the concepts of a micro service architecture in a technology independent way, and mapping rules, which map an implemented micro service-based system into an architectural model which instantiates the metamodel.

In this paper, we present a study which allows us to define MiSAR's metamodel and mapping rules based on empirical data. The study involves a systematic analysis of 8 micro service applications. The analysis is conducted in two stages: in the first stage we analysed one system, identified code elements and created architectural abstractions. This allowed us to have an initial version of the metamodel and mapping rules. In the second stage, we refined the metamodel and mapping rules by validating them on 7 additional systems.

This paper is organized as follows: section 2 motivates our approach; section 3 describes the design of our study; section 4 and 5 present and discuss the results; section 6 notes threats to the validity of our study; section 7 discusses related work; finally, section 8 highlights our conclusions.

## II. MiSAR Motivation and Overview

### A. Problem and motivation

There are a variety of benefits associated with the utilization of the MSA style. MSA has increasingly been considered more beneficial than traditional layered architectural software systems [3]. Several of its core benefits

are: agility, reliability, resilience, scalability, developer productivity, maintainability, separation of concerns and overall ease of deployment [7]. However, micro service architectures introduce a level of complexity into applications. MSA is very dynamic, each micro service is fine-grained and runs independently of others in its own container, resulting in a highly distributed system with many dependencies.

Keeping control of the overall architecture during MSA system development can be very difficult, especially when MSA is designed, developed and deployed by different stakeholders with different disciplines. Moreover, developers tend to follow evolutionary design strategies, making it very hard to manage architectural constraints that may be put in place by different architects at different times. The effect is that the architecture of an implemented system has typically diverged significantly from any documented model, and often fails to meet prescribed architectural constraints. A possible solution to this problem would be to recover the actual architectural structure of an implemented system, to help development teams comprehend the structure, connections and dependencies of their applications. However, previous support for architecture recovery processes has focused mainly on object-oriented systems with diverse language subsets [8]. Little research has been conducted regarding architecture recovery for MSA systems, as reported in a recent mapping study of this area [9].

The motivation of this study is to provide architectural recovery support for this emerging architectural style. This requires understanding and defining various concepts and elements that form an MSA model, and determining a repeatable method to map between the source code of a micro service implementation and an architectural model of that implementation. To achieve this, we have decided to take a MDE approach to formalize and implement the models and mappings.

### *B. The Micro Service Architecture Recovery approach*

The complexity of the micro service architectural style makes the task of understanding its many artifacts very difficult, as applications consist of many small components, interfaces and dependencies. The ideal way to comprehend the complexities is to model the artifacts themselves as accurately as possible. MDE achieves this by adopting a bottom-up model-driven transformation process for recovery of the architecture. We believe that this approach is particularly suited to the distributed, fine-grained nature of MSA systems. In addition, one of its competitive advantages is that modeling occurs at multiple abstraction levels, which could help elucidate a model-driven transformation for a more holistic approach to architecture.

This study focuses on the MDE's Platform Independent Model (PIM) alongside the Platform Specific Model (PSM) abstraction level in relation to the modeling of micro service architecture platforms. These models are critical in order to better understand the core of reverse engineering, where the PIM supports the architectural model recovered and the PSM supports the technology of the implemented micro service system. The MiSAR process is based on the transformation from PSM to PIM as shown in Figure 1. This implementation pathway has a process that includes code, XML files, schema, run-times, etc. that are converted into PIM. This is achieved by providing mapping rules to derive these models.

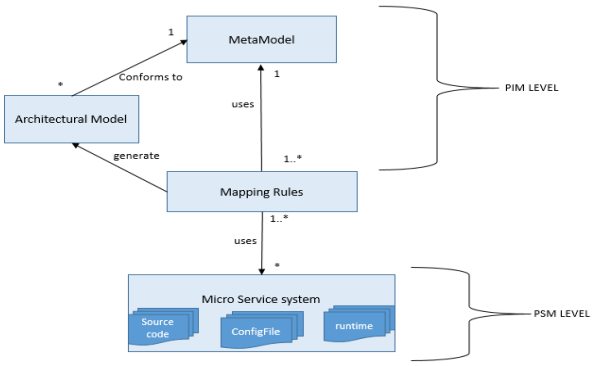


Figure 1. The MiSAR approach

The PIM level describes micro service models which are independent of the execution platform and of the technology being used. The PSM level describes micro service platform models as executable artifacts, combining the PIM with additional features of a specific platform. Platform specific models in micro services can be conceptualized into two areas: the runtime platform and the development technology. The runtime model provides information to understand the connectivity and orchestration. Micro services can be packaged into runnable images which can be Docker based containers or an open VM format. The other area is the development technology or frameworks which are used to accelerate the development of micro services. There are configurable frameworks such as Spring Boot which provide many design pattern implementations to make micro service development rapid for a developer. For instance, a Docker file, a pom.xml and a bootstap.yml at PSM Level are used to represent a micro service concept at PIM level as in Figure 2.

Our approach aims to recover the architecture of micro service based systems. In order to do this, it has to include a metamodel and mapping rules that support the recovery process. By following the MDE approach, we aim to make these artifacts reusable, so that even though our own focus is on reverse engineering, these artifacts can also be used in other forward engineering architectural approaches, such as code generation.

## III. Study Design

### *A. Study aim and research questions*

The aim of this study is to develop MiSAR from empirical data. To achieve this, we have defined three research ques-

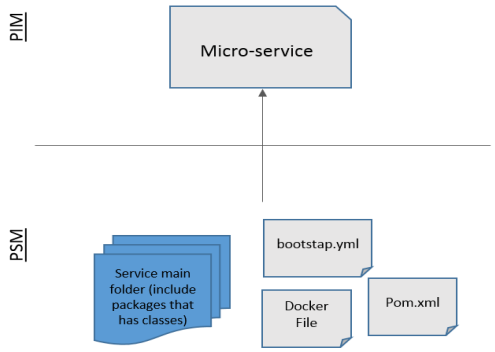


Figure 2. Representation of micro service concept at PSM and PIM layers

| | Project Name | Number of micro services | Size LOC (line of code) | Number of developers | Timeline for project development | Documentation | Architecture Diagram |
|---|---|---|---|---|---|---|---|
| 1 | PiggyMetrics | 13 | 3309 | 5 | Mar 29, 2015 – Aug 17, 2017 | available | available |
| 2 | 3PillarGlobal | 7 | 474 | 1 | Aug 30, 2015 – Jan 4, 2018 | Not available | Not available |
| 3 | ThoughtMechanix | 7 | 2261 | 1 | Jun 30, 2017 – May 13, 2017 | Not available | Not available |
| 4 | Microservice Consul Sample | 11 | 2434 | 3 | Jun 19, 2016 – Jan 4, 2018 | available | Not available |
| 5 | spring-cloud-consul-example | 7 | 286 | 1 | Jun 5, 2016 – Jan 4, 2018 | available | available |
| 6 | spring-cloud-netflix-example | 9 | 328 | 1 | Jun 5, 2016 – Jan 10, 2018 | available | available |
| 7 | Spring Cloud microservices and integrating sidecar applications | 5 | 2434 | 1 | Dec 20, 2015 – Jan 4, 2018 | Not available | Not available |
| 8 | blog-microservices | 14 | 2093 | 1 | Mar 1, 2015 – Jan 4, 2018 | Not available | Not available |

Figure 3. Selected systems for analysis

tions which the study needs to address (see Table I). This can be successfully achieved by defining the metamodel and mapping rules, which correspond to the artifacts of the MDE approach (RQ1, RQ2). As we are reverse engineering from a software system, we need to classify the kind of software analysis to be conducted for extracting the micro service architecture as either static or dynamic analysis (RQ3).

### B. Selecting the case studies to study

We selected open source projects from the Github repository[1] that followed the micro service architecture. We started by performing a search on the repository facility using the terms "micro service", "micro service", "micro-service" and "micro service architecture". Furthermore, we applied specific criteria to support project relevance as stated in Table II. In this study, we limited our study to 8 systems (case study 1 [10], case study 2[11], case study 3 [12], case study 4 [13], case study 5 [14] , case study 6 [15], case study 7 [16] and case study 8 [17], as described in Figure 3.

[1]https://github.com/

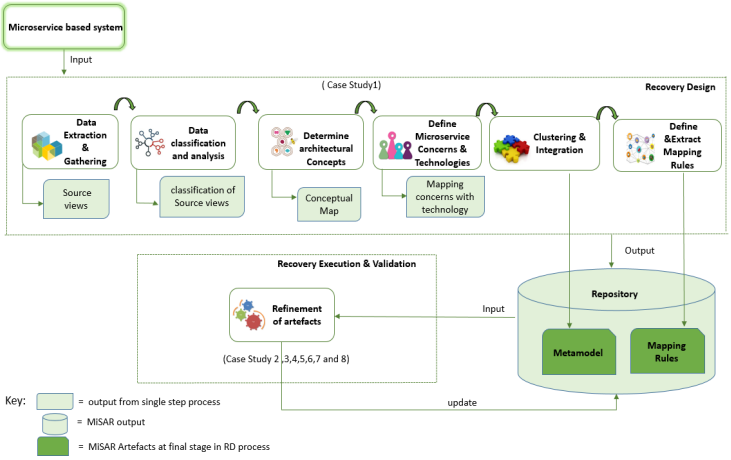


Figure 4. Micro service architecture recovery steps

### C. Research design

As the objective of our study is architecture recovery, our study is designed as a manual architecture recovery process. We have customized the process presented in [18], which includes two main phases: Recovery Design (RD) and Recovery Execution (RE) as depicted in Figure 4. Typically, the two phases are iterative and incremental. The first phase attempts to plan the recovery by defining the architectural concepts along with the mapping rules. In the second phase, we execute the plan for validation purposes, apply the metamodel and mapping rules defined in the first phase to create architectural models. The outcome of the validation may lead us to repeat the steps, by refining the metamodel and mapping rules, and re-validating.

Within the RD phase, our study analysed case study 1. This case study was chosen due to the availability of its architecture documentation and supporting diagrams with illustrations, which can be used to compare the results of this phase with the documentation. Case studies 2 to 8 were used in the second phase. The steps taken in each phase are described in the following sections.

*1) **Phase 1: Recovery design**:* During recovery design we determine the micro service architectural concepts that build the system, and identify mapping rules from the code to the concepts. These steps are separated into the following:

***Step 1 – Data extraction and gathering**:* It is important to gather the required data from artifacts of the software system for recovery of the micro service architecture. Data from artifacts includes the source code, configuration files, descriptive files etc. which are then collected and stored within a data repository.

**Technique.** We have extracted data from the following source files:

- Docker files: text documents that contain command lines to assemble an image in order to run a container and/or a service.

Table I
THE RESEARCH QUESTIONS AND THEIR MOTIVATIONS

| Number | Research Question | Motivation |
|---|---|---|
| 1 | What are the micro service architectural elements/concepts that are identified from the source code? | The aim is to identify the concepts and elements needed to build a metamodel and specific-purpose abstraction of the micro service based system. |
| 2 | What are the mapping rules between the source code of micro service implementations and the architectural model? | The aim is to develop mapping rules that derive a target model from the source model |
| 3 | What kind of software analysis is needed to capture the micro service architecture ? | The aim is to evaluate and assess the need of static and dynamic analysis in the process of system recovery within the micro service framework. |

Table II
THE SELECTION CRITERIA

| | Criteria |
|---|---|
| Inclusion | • Projects that are only developed in Java.<br>• Projects which demonstrate the usage of the Micro Service Architectural Style. (by asking developers and reviewing documentation)<br>• Projects that have Spring Boot and/or Spring Cloud OSS framework<br>• Projects that have Docker Technology. |
| Exclusion | • Projects that use two or less Spring Cloud components. A project that uses few Spring Cloud libraries decreases the probability of the project being a micro service<br>• Project/repositories under the Spring Cloud GitHub repository because they are framework projects not micro service applications. |

- Docker Compose files: for defining and running multi-container Docker applications.
- Java source code.
- Maven POM.XML file: An XML file that contains information used by Maven to build the project.
- YAML files used for configuration files.
- Documentation and Tools Support.

**Output.** The outcome of this stage is a repository which contains data of the source files.

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

**Technique.** Static analysis is performed by observing only the artifacts of the system. To extract a static view of the system, we used a reverse engineering tool: Enterprise architect[2]. This tool is applied to the Java source code to generate UML class diagrams. Dynamic analysis is performed by observing the system during execution and aims to extract information from running code. We have implemented and enabled the Zipkin[3] server with our open source projects[4], then used the Zipkin tool to trace communication between micro services, so that we were able to build a call graph from one micro service to another. Zipkin captured all the calls and dependencies between different micro services as shown in Figure 5 for Case Study 1. TCPDump provided us with low level TCP protocol connectivity and communication which provided information about the latencies between different components of the system. Since the system was deployed using Docker container, the Sysdig tool was used to obtain performance diagnostics at the container level which provided information about the ports, IP addresses of containers, and connectivity between different containers.

**Output.**The outcome of this stage is a fusion of extracted information from both static and dynamic analysis.

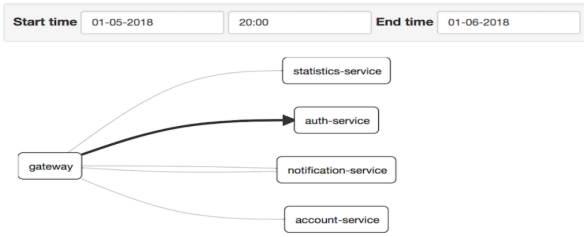


Figure 5. Dynamic analysis by using Zipkin

***Step 3 – Determine architectural concepts:*** We have used both bottom-up (code→model) and top-down (model→code) techniques to understand and determine the micro service architectural concepts. The mechanism used for the bottom-up technique is to initially analyze the source code and configuration files. The concepts are discovered after abstracting and evaluating their relevance to architectural elements. The top-down technique focused on using micro service patterns [19] which allowed us to identify several concepts and supported the definition of the underlying features and behaviour of micro service-based systems.

**Output.** The outcome of this stage is a conceptual map which contains the identified concepts from both techniques that are relevant to our analysis, as depicted in Figure 6.

***Step 4 – Define micro service architectural concerns:*** Concerns are common characteristics of micro services in an

[2]http://www.sparxsystems.com.au/products/ea/
[3]https://zipkin.io/
[4]https://github.com/nuha77/piggymetric-with-Zipkin

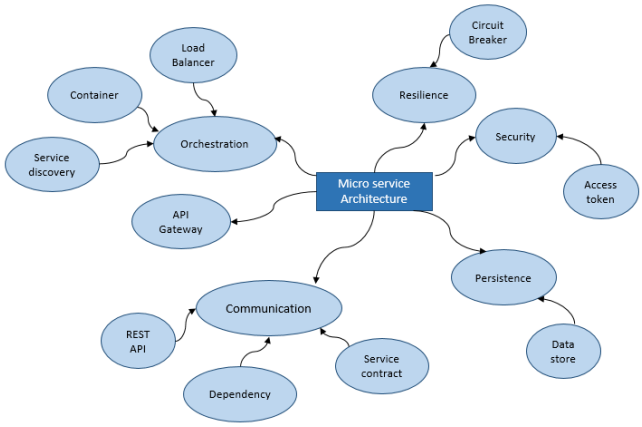


Figure 6. Micro service conceptual map

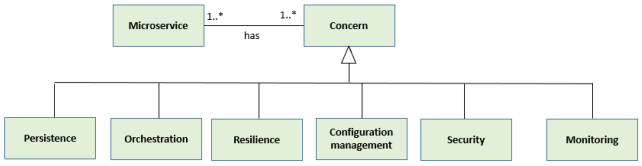


Figure 7. Micro-service architecture concerns

architecture, which can be commonly implemented across multiple micro services. These can be related to making micro service fault tolerant, ease their deployment and discovery. The focus of the study will be the most common technical concerns which are presented in Figure 7, whereas non-technical concerns such as organization structure, culture and so on are excluded.

**Technique.** Several concerns appeared in Step 3 as shown in our conceptual map. As concerns are difficult to identify from the code, we used the literature as in [20] to review the most common ones that have to be considered in micro service systems. We then identify the technologies that are commonly used to implement these concerns in a Spring Cloud OSS-based micro services environment. Recovering these technologies can help determine whether a given micro service implements specific concerns. This will help in identifying and building the relations between various platform services and the functional or business services in the platform independent model.

**Output.** The outcome of this stage is a list of concerns to be taken into account in the micro service architecture.

***Step 5 – Clustering and integration:*** After identifying the various common architectural concepts in micro services, we clustered them together based on high level related concerns that we have identified.

**Technique.** Our technique represents the concepts as meta classes by grouping related architectural concepts together in one cluster based on their micro service concerns. An association in the metamodel is added for Meta classes that are related. Finally, we integrate and abstract the concepts and their relationships.

**Output.** The outcome of this stage is a metamodel for micro service architectures.

***Step 6 – Define mapping rules:***

**Technique.** To define the mapping rules we manually inspected and examined the system by analyzing source files available in the project directory. Then, for each concept in the metamodel, we used the extracted files of that concept and analysed them to define the mapping rules which map architecture concepts with implementation artifacts. The mapping rule extraction process was performed at two analysis levels: micro service system level and micro service level. The micro service system level involves analyzing docker-compose.yml file, and project build files generated either by the Maven build tool (e.g. pom.xml), or by Gradle (e.g. settings.gradle). The micro service level involves analyzing the Maven build file, the source code and Docker file. Information collected for each mapping rule included the input artifact being studied (e.g. container orchestration file, application build file, source code file etc.), and then mapping architecture concepts (e.g. micro service, dependency, service discovery, API gateway etc.) onto implementation artifacts. Mapping rules were then classified and grouped based on the output architectural element they mapped to.

**Output.** The outcome of this stage is a set of mapping rules for each concept in the metamodel.

*2) **Phase 2: Recovery execution and validation:*** we conduct a validation for the results obtained from the RD phase, using case studies 2 to 8.

***Refinement of artifacts:*** The metamodel and mapping rules obtained in the RD phase are applied and validated against the seven case studies for enhancement and validation purposes.

**Technique.** We analyzed the 7 system implementations manually and applied the mapping rules and metamodel. Based on the success of this analysis, we amended and enhanced the mapping rules and architectural elements.

**Output.** The outcome of this stage is an updated MiSAR repository with mapping rules and metamodel.

## IV. Results

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

**RQ1: Micro service metamodel**. Regarding RQ1, there are various architectural elements which are fundamental to any system. Therefore, they appear across all the selected case studies. Figure 8 shows the architectural concepts we have discovered, and the case studies (indicated in numbers) where we have identified them from. We can also observe that various architectural elements appear only in few cases due to various contextual demands of the projects. We have

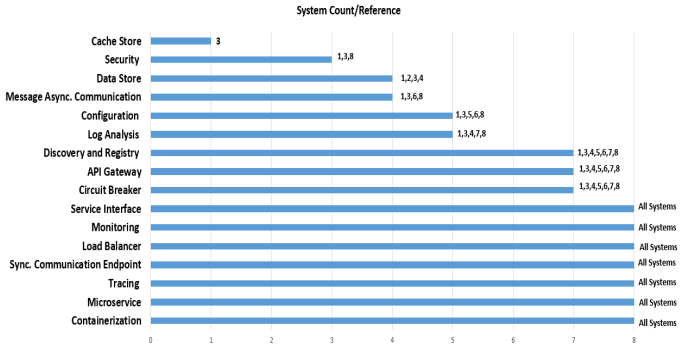


Figure 8. Architectural concepts, counts and system references

analyzed the context of these cases and determined the need for these elements to be used by the designer of the system.

We can observe that containerization appears across all the projects. This is due to the fact that our initial selection criteria for the case studies included the usage of Docker. Docker is fundamentally a containerization technology, hence all the case studies in Figure 3 use containerization as an architectural element. Docker is the most commonly used containerization technology hence most micro services use Docker as the container image format of choice.

Configuration is also a fundamental architectural element which happened to have been used across all case studies except case study 2 and 4. This is probably due to the size of the project. It contains few micro services and the project's objective is a proof-of-concept for micro service development

API gateway is present in most projects, suggesting that the use of API gateways is very common in micro services. This is due to the fact that API gateway allows architects to configure cross functional elements such as security, logging and authorization. This relieves individual services to handle these architectural elements within their code.

Registry and discovery were discovered in most projects. However, each project uses different technologies to implement this concept. For example, 5 case studies used Netflix Eureka, while Consul was used in 2 studies(4 and 5). Again, as with the configuration element, due to the small size of system 2, registry and discovery are not used.

As shown in Figure 8, we find that some concepts are essential in micro service architecture, and are found in all systems, while others are not. Based on these counts, we have defined the metamodel shown in Figure 9. For instance, Containerization, Microservice, Service Interface and Endpoint are found in all analyzed systems so when defining our metamodel we would represent this with mandatory associations: one-to-one or one-to-many multiplicities. For example, one micro service should run independently in one container and have at least one communication endpoint.

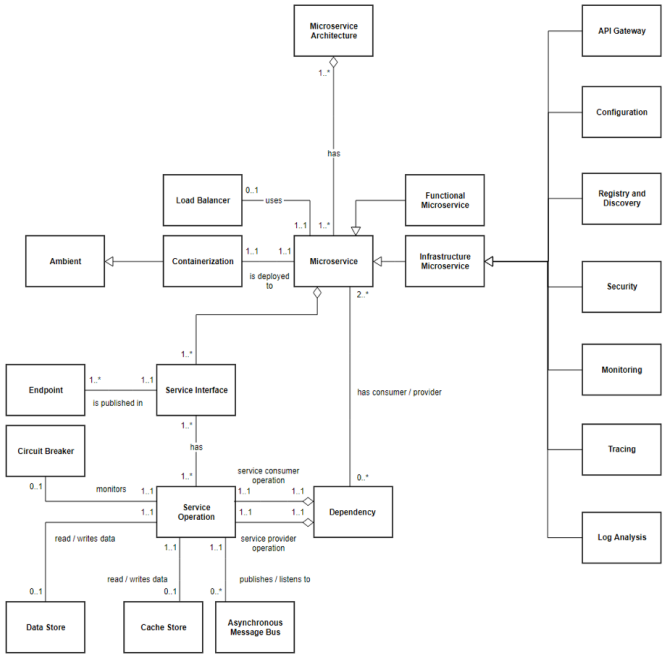


Figure 9. Micro service architecture metamodel at PIM-level

However, Security was implemented in only three systems, even though it is an essential concern of a micro service system, and so its association is not mandatory.

We can also notice from Figure 8 that most metamodel elements were discovered in Phase 1, as case study 1 was used in this phase. The metamodel was refined in Phase 2 with a new element cache store, which was discovered in case study 3.

In the following, we provide a description of the concepts of our metamodel:

**Micro service architecture** is the logical repository of micro services. It contains one or many micro service instances along with the components implementing them. **Micro service** is the central and main building block of our metamodel and it is generally a software application that offers a complete independent service. In a micro service architecture, there might be multiple instances of the same micro service type as well as different types depending on the domain of a micro service system. Micro services are broadly classified into: **Functional micro service** type, which realizes the system's business capabilities as well as a set of **Infrastructure micro service** types, which provide various back-end support to the operation of micro services. Infrastructure micro service types include **API Gateway**, **Configuration**, **Discovery and Registry**, **Security**, **Log Analysis**, **Monitoring** and **Tracing**. As stated in Figure 9, every micro service in a micro service architecture has to be configurable, discoverable and able to communicate its health.

Although implementation of micro services differs, every micro service instance is defined by at least one **Service Interface**. The Service Interface identifies what **Service Operations** can be called by remote services and how. Unless the micro service instance is stateless, one Service Operation can preserve the micro service's local state in one **Data Store**. A **Cache Store** element, on the other hand, preserves response data of remote micro services that were requested previously in order to decrease number of future requests. This element contributes to improve the response time of the micro service especially if the data at the remote micro service does not change often.

The deployment concern of the micro service architecture model is represented by **Ambient** and **Containerization** elements. They describe in which architectural context the micro services are to be deployed. Ambient is the boundary of a micro service [21]. A container is a kind of ambient. Each micro service instance will be running in exactly one container. A container is an execution environment used to isolate each micro service within one virtual machine leveraging the host's hardware and operating system capabilities while enabling each micro service to appear as a completely stand-alone software artifact that is running externally [22].

The **Dependency** element describes the communication between one consumer micro service and one provider micro service by which the consumer service leverages the information and functionality of the provider service. One micro service (whether it is consumer or provider) can have Dependency instances. This communication takes place as one consumer's Service Operation invokes one provider's Service Operation per one Dependency instance. It occurs either in synchronous request-based manner or in asynchronous message-based manner. The asynchronous fashion of service-to-service communication occurs when the provider service publishes messages at one or many channels in an **Asynchronous Message Bus** and the consumer service listens to provider's incoming messages on the same channels. A Dependency can occur between two different instances of Functional micro services, two different instances of Infrastructural micro service or between an instance of Infrastructural micro service and another instance of Infrastructural micro service.

In such an environment that is rich in communication taking place among multiple instances of micro services, resilience and load balancing requirements are necessary to maintain a healthy execution environment for micro services. Resilience is represented by the **Circuit Breaker** element which fails fast requests to misbehaving micro services. Each **Service Operation** can be monitored by one **Circuit Breaker**. The **Load Balancer** element distributes all requests of a Micro service instance over available instances of another Micro service type. The aim of load balancing is to achieve parallel execution and hence a faster response time for accomplishing a service. Like **Circuit Breaker**, **Load Balancer** is optional for any Micro service instance such that one Micro service instance may use at most one **Load Balancer**.

**RQ2: Mapping Rules**. Regarding RQ2, Figure 10 shows the number of related mapping rules per architectural concept. Several architectural concepts have several mapping rules because several technologies are used to implement the same concept. The Data Store concept, for instance, was implemented differently as MongoDB in case study 1 and 2, as PostgreSQL in case study 3, and as HSQLDB in case study 4. On the other hand the Container concept was always implemented as Docker and the micro service concept as a Spring Boot application. The effect of this was noticeable in the recovery process where concepts with a standard implementation were faster to recover.

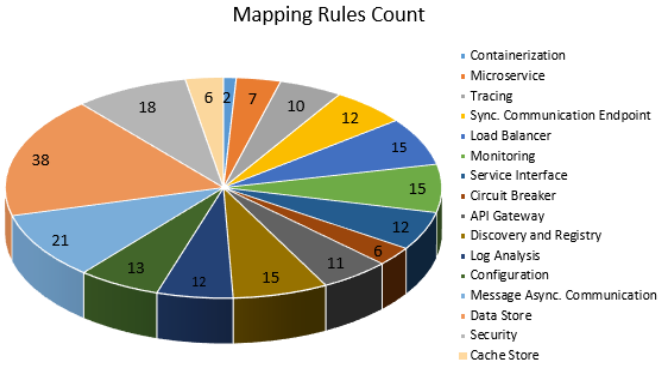


Figure 10. Number of mapping rules per concept

211 mapping rules were identified from the source files. In the RD stage where Case Study 1 was used, we identified 82 rules. In the RE stage, we identified 129 new rules and we refined 10 rules, which were identified in the RD. For example, a total of 11 rules are defined for the API Gateway concept as shown in Table III. Table IV shows all mapping rules for Containerization concept.

As we can see from Table III and Table IV, the mapping rules map between source files and architectural elements (or PIM concepts). There are two types of mapping rules, one type is the PIM Concept Identification Rule which identifies the implementation of corresponding architecture element type, i.e. at PIM Concept (Source). The other type is the PIM Dependency Identification Rule which indicates the association between two PIM concepts, source and destination. For example in Table III all mapping rules but the fourth are considered as PIM Concept Identification Rule since they map to a PIM Concept (source). On the other hand, the fourth mapping rule identifies a dependency from one Micro service PIM concept, i.e. source, to another API Gateway PIM concept, i.e. destination.

**RQ3: Kinds of system analysis**. Mapping rules related

Table III
MAPPING RULES TO IDENTIFY API GATEWAY

| Source File (Artifact Type) | PIM Concept (Source) | PIM Concept (Destination) | Mapping rules |
|---|---|---|---|
| Container Build File | API Gateway | | 1.implementation of Apache HTTP gateway server is indicated by command: FROM ubuntu:16.04 and RUN apt-get install -y -qq apache2. |
| Build File | API Gateway | | 2.implementation of Spring Cloud Netflix Zuul gateway server is indicated by node: <artifactId>spring-cloud-starter-zuul</artifactId>. |
| Build File | API Gateway | | 3.implementation of Spring Cloud Netflix Sidecar API gateway server is indicated by <artifactId>spring-cloud-netflix-sidecar</artifactId> entry |
| Configurations File | Microservice | API Gateway | 4.microservice that is routed by Spring Cloud Zuul API gateway is indicated by zuul.routes.[microservice-name].path entry. |
| Configurations File | API Gateway | | 5.implementation of Spring Cloud Netflix Zuul gateway server is indicated by zuul.routes. entry. |
| Configurations File | API Gateway | | 6.implementation of Spring Cloud Sidecar API gateway server is indicated by sidecar.port:[port-number] entry. |
| Configurations File | API Gateway | | 7.implementation of Spring Cloud Netflix Zuul gateway server is indicated by zuul.host.entry. |
| Configurations File | API Gateway | | 8.implementation of Spring Cloud Netflix Zuul gateway server is indicated by zuul.prefix. entry. |
| Configurations File | API Gateway | | 9.implementation of Spring Cloud Netflix Zuul gateway server is indicated by zuul.ignoredServices. entry. |
| Configurations File | API Gateway | | 10.implementation of Spring Cloud Netflix Zuul gateway server is indicated by @EnableZuulProxy or @EnableSidecar class-level annotation in its Spring Boot Application class. |
| Configurations File | API Gateway | | 11.implementation of Spring Cloud Netflix Sidecar gateway server is indicated by @EnableSidecar class-level annotation in its Spring Boot Application class. |

Table IV
MAPPING RULES TO IDENTIFY CONTAINERIZATION

| Source File (Artifact Type) | PIM Concept (Source) | PIM Concept (Destination) | Mapping rules |
|---|---|---|---|
| Container Build File | Microservice | Container | 1.Each microservice application has one Dockerfile container image build file under its root directory. |
| Container Orchestration File | Microservice | Container | 2.microservice container is indicated by either one node under services node or by first-level node with image:[microservice-folder-name] or build:[microservice-folder-name] properties. |

to all architecture concepts in our proposed metamodel were extracted using static analysis, but that was mainly possible due to the presence of a container orchestration file (docker-compose.yml). Without it, dynamic analysis would have been required in order to inspect the execution context of the micro service architecture including integration of non-JVM applications and external backing services needed at runtime. These aspects cannot be checked statically as they are sometimes wrapped in Spring annotations and default configurations. Several mapping rules could be identified by using both static and dynamic analysis. For example, we can identify the port of a micro service using the docker-compose.yml and/or the Dockerfile and at the same time, we can confirm this by running software like TCPDump or trace the requests that the service sends/receives at runtime. Figure 11 shows extracted information using static analysis and dynamic analysis.

| Analysis type | Extracted information | Extracted information source | Type of extraction required (Manual/Tool support ) |
|---|---|---|---|
| Static Analysis | Classes/packages | Java Source code | Enterprise architecture tool |
| | Docker file defines how the micro service is packaged including any operating system level dependencies. | Docker File | Manual |
| | Docker compose describes how multiple micro services are run together. It may also describe what ports that the micro services will be listening to accept external requests. | Docker compose file | Manual |
| | YML file describe system runtime configuration values. | YML File | Manual |
| | Developer ,LOC, timeline | Java Source code | Manual -GitHub metadata |
| | POM file describes the micro service build dependencies. | POM.XML | Manual |
| Dynamic Analysis | Traces provided us information about how micro services are communicating with another. | Traces | Zipkin tool |
| | Information about how system is behaving | logs | Docker logs command |
| | Image name , IP addresses | Containers | Docker inspect command |
| | IP addresses of containers , ports and connectivity between different containers | Commands in terminal | Sysdig tool |
| | low level TCP connection and communication information between various micro services | Network trace/logs | Tcpdump |

Figure 11. Static/Dynamic system analysis

## V. DISCUSSION

The study we conducted followed a manual recovery process as described in subsection III-C. Initially, we wanted to follow the same steps as the one described in [18]. This kind of process starts by defining the architectural concepts, abstractions and concerns that could be recovered. Usually, these are known beforehand, and architects extract and classify system data to map them to these architectural concepts. However, we noticed that when we started this process that there is no standard metamodel for the micro

service architecture. Therefore, we opted to extract the data of the system and analyse it first and then abstract the result into architectural concepts. For the architectural concerns of micro services, there are standard ones.

From the results of our study, we can notice that from the RD stage where Case Study 1 was used, we identified all architectural concepts but One and for the mapping rules we identified 82 rules. In the RE stage, we identified 129 new rules and we refined 10 rules, which were identified in the RD. We did notice that the refinement of the mapping rules became less needed, as we validated them with new case studies.

It is clear that the mappings between many architectural elements (PIM) and software elements (PSM) are not 1-1, that is many mapping rules have to be applied to map one concept. For example, 38 mapping rules have to be used to generate the data store architectural concept as shown in Figure 10. Many of these new mapping rules were not related to new concepts but to the fact that for the same architectural concepts, different technologies can be used. This makes the implementation of these mappings more complicated, and any future MiSAR tool should be able to identify and analyse a range of different technologies.

Most of our mappings use static code analysis. We also observed that the model recovered from these rules can be validated if a dynamic analysis is performed at system runtime. This is an important finding, as it can demonstrate that many parts of the micro service architectures can be recovered statitically, which is much easier than using dynamic analysis.

## VI. Threats to validity

Internal threats of validity concern factors that impact the integrity of the study results. As a new alternative superceding the monolithic application, one of the drawbacks of the micro service architecture approach is that it is still developing. Hence there is a lack of consensus within the industry on what this architecture is and how it can be implemented. The present study was based on systems considered to represent basic and best practices, but micro service patterns are still evolving [19]. In addition, although the study results have been validated by eight case studies, the mapping rules are not yet complete and more mapping rules could be identified by analyzing more projects. Furthermore, empirical reliability, which refers to the consistency of data capture and interpretation, is relevant to this type of study as data extracted and analysed is qualitative and can be interpreted differently. To minimize this, the analysis was all conducted by the first author researcher and samples were reviewed by second author. When disagreements happened, the third co-author was available to help resolve them.

Threats to external validity concerning this study are related to the generalization of results. Our metamodel is independent of technology and can be applied in different technologies. However, our study was only limited to analyse Java systems using the Spring Cloud framework. Further evaluation on various projects utilizing different programming languages and frameworks will be important to asses the generality of our model.

## VII. Related work

The popularity and success of architecture recovery solutions in extracting architectural information is commendably strong with a very rich debate nurturing strong academic research [23]. Nonetheless, there is a dearth of available research analysing the architecture recovery within the micro services area. This awareness became apparent from two recent literature surveys, [24], which reported that “in the literature area only little work on reverse engineering and architecture recovery in micro service architecture had been described”, and [9] which concluded that “little published work is available on reverse engineering”.

To the best of our knowledge the only study found in the literature that tackles architecture recovery in micro services is MicroART [25]. MicroART is the first prototype of an architecture recovery tool for micro service-based systems. It recovers the architecture at two distinct phases: physical and logical architecture recovery. The Logical architecture recovery is performed by allowing the user to interact and refine the physical architecture automatically recovered. Similarly to our approach, it uses Model Driven Engineering principles [26]. Our approach as presented in this paper has been developed based on a manual recovery of a set of micro service systems. The metamodel of MicroART presented in [26] is simpler than MiSAR, covering fewer architectural concepts and concerns. Also, the MicroART metamodel is concerned with the development teams, whereas ours focuses on the micro service system.

Another related work is the one presented in [27]. Even though this work focuses on the service oriented architecture rather than the micro service architecture, there are many similarities in the MDE approach undertaken. Their approach is similar to MiSAR as it focuses on service level components rather than class level of software design. However, their reverse engineering method focuses on understanding the problem domain rather than focusing specifically on the implementation pathway.

## VIII. Conclusions

This paper presents the metamodel and mapping rules of the MiSAR approach, which are artifacts that are used to recover architectures of micro service systems. To be able to define these MDE artifacts, we designed and conducted a study which included a manual and iterative recovery process. We believe that by conducting our study, our approach has considered the key architectural concepts encountered in micro service systems and their mapping rules.

Our empirical study will feed into further research to support and provide practitioners and researchers with unified approaches, terminologies, mechanisms, methodologies and processes of performing architecture recovery in micro service architecture. Moreover, we have generated a solid conceptual framework to help compare case studies, as well as specific concepts and mapping rules which can act as a catalyst for tool developers to create reverse engineering tools.

## REFERENCES


[1] M. Rahman and J. Gao, "A reusable automated acceptance testing architecture for microservices in behavior-driven development," in *Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on*. IEEE, 2015, pp. 321–325.

[2] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[3] M. Fowler and J. Lewis, "Microservices," *ThoughtWorks. http://martinfowler.com/articles/microservices.html [last accessed on February 17, 2017]*, 2014.

[4] N. Ali, J. Rosik, and J. Buckley, "Characterizing real-time reflexion-based architecture recovery: an in-vivo multi-case study," in *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*. ACM, 2012, pp. 23–32.

[5] J. Buckley, N. Ali, M. English, J. Rosik, and S. Herold, "Real-time reflexion modelling in architecture reconciliation: A multi case study," *Information and Software Technology*, vol. 61, pp. 107–123, 2015.

[6] A. Sadovykh, C. Hahn, D. Panfilenko, O. Shafiq, and A. Limyr, "Soa and sha tools developed in shape project," in *Fifth European Conference on Model-Driven Architecture Foundations and Applications*, 2009, p. 58.

[7] S. Daya, N. Van Duy, K. Eati, C. M. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins *et al.*, *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM Redbooks, 2016.

[8] T. Richner and S. Ducasse, "Recovering high-level views of object-oriented applications from static and dynamic information," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 13–22.

[9] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*. IEEE, 2016, pp. 44–51.

[10] A. Lukyanchikov, "Microservice architecture with spring boot, spring cloud and docker," https://github.com/sqshq/PiggyMetrics, March 2015 – 2017.

[11] I. 3Pillar Global, "3pillar global," https://github.com/3PillarGlobal/microservice-blog/tree/part4/step3, March 2018.

[12] J. Carnell, "Pillarglobal," https://github.com/carnellj/spmia-chapter10, May 2017.

[13] E. Wolff, "Microservice consul sampler," https://github.com/ewolff/microservice-consul, January 2018.

[14] M. Zhang, "spring-cloud-consul-example," https://github.com/yidongnan/spring-cloud-consul-example, January 2018.

[15] ——, "spring-cloud-netflix-example," https://github.com/yidongnan/spring-cloud-netflix-example, January 2018.

[16] D. Steiman, "Spring cloud microservices and integrating sidecar applications," https://github.com/xetys/microservices-sidecar-example, January 2018.

[17] C. Enterprise, "blog-microservices," https://github.com/callistaenterprise/blog-microservices, January 2018.

[18] A. Van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva, "Symphony: View-driven software architecture reconstruction," in *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*. IEEE, 2004, pp. 122–132.

[19] C. Richardson, "Microservice architecture," http://microservices.io/patterns/microservices.html, 2017.

[20] B. Ibryam, "Spring cloud for microservices," https://developers.redhat.com/blog/2016/12/09/spring-cloud-for-microservices-compared-to-kubernetes/, December 2016.

[21] S. Hassan, N. Ali, and R. Bahsoon, "Microservice ambients: An architectural meta-modelling approach for microservice granularity," in *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1–10.

[22] S. J. Vaughan-Nichols, "What is docker and why is it so darn popular?" http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/, May 2017.

[23] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cimpan, and H. Verjus, "Towards a process-oriented software architecture reconstruction taxonomy," in *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*. IEEE, 2007, pp. 137–148.

[24] P. Di Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 21–30.

[25] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Microart: A software architecture recovery tool for maintaining microservice-based systems," in *IEEE International Conference on Software Architecture (ICSA)*, 2017.

[26] ——, "Towards recovering the software architecture of microservice-based systems," in *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*. IEEE, 2017, pp. 46–53.

[27] R. Akkiraju, T. Mitra, and U. Thulasiram, "Reverse engineering platform independent models from business software applications," in *Reverse Engineering-Recent Advances and Applications*. InTech, 2012.