The present work was submitted to
the Research Group
Software Construction

of the Faculty of Mathematics,
Computer Science, and
Natural Sciences

Master Thesis

Mocking Microservice Architectures through Message Sequence Models

presented by

Pavan Nadkarni

Aachen, May 25, 2023

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Alex Sabau, M.Sc.

Acknowledgment

Firstly, I express my profound gratitude towards my supervisor and mentor, Alex Sabau. This endeavour would have been unimaginable without his constant support and invaluable feedback that helped in this journey's phases.

Next, I would like to express my immense appreciation towards Prof. Dr. rer. nat. Horst Lichter and Prof. Dr. rer. nat. Bernhard Rumpe for examining my thesis and providing indispensable feedback that helped me refine and polish my ideas and vision for this thesis. Additionally, a special thanks to the Software Construction Research Group and RWTH Aachen, which helped streamline the entire process.

Finally, I sincerely thank my family for their perpetual support and presence during my most challenging times and my friends and work colleagues whose constant support has kept me focused and high-spirited. Thank you all for making this journey a very memorable part of my life.

Pavan Nadkarni

Abstract

Microservice Architecture (MSA) is a modern architectural style of building software applications where a single application gets broken down into a suite of granular services called a microservice. Individually microservice is designed to have a single responsibility within a defined bounded context. As a result of this fine decomposition based on bounded context, a need for intercommunication and cooperation between other microservices materialises to realise complex functionalities.

Enterprise applications can consist of hundreds of microservices distributed across multiple data centres. The communication network between microservices can become extraordinarily complex in such distributed applications. Consequently, designing and developing such complex distributed applications or migrating existing monolithic architectural style applications to MSA is a challenging task where inaccurate designs can give rise to severe performance issues and lead to the accumulation of technical debt for the future.

To avoid this, it is common in industries to test and create prototypes to provide proof of concept before adopting or implementing any major architectural changes. In such situations, the ability to model and mock MSA systems with the specific desired changes can help observe their behaviour in real-time, identify design issues and mitigate risks at earlier stages of software development/migration lifecycles.

In order to facilitate such novel use cases, in this thesis, we explore the idea of mocking MSAs by relying on message sequence models, which we use to model inter-service communications between microservices for *functional requirements* of MSAs. As part of this, we propose a metamodel to capture microservices data with associated communication to model MSAs. We also develop a prototype that transforms concrete instance models of our metamodel into mocked-up microservices. Lastly, we present an approach to build, compose and deploy observable polyglot microservice applications in an automated way to mock MSAs.

Contents

List of Tables

List of Figures

List of Source Codes

1. Introduction

> The formulation of a problem is often more essential than its solution, which may be merely a matter of mathematical or experimental skill. To raise new questions, new possibilities, to regard old problems from a new angle requires creative imagination and marks real advances in science.
>
> Albert Einstein

Contents

In an era where companies are shifting from "on-premise" based deployments to offering services in the cloud, distributed and cloud-native software applications are increasing. Microservice Architecture (MSA) is a more modern software development architectural style gaining wide adoption when developing cloud-native software applications [GBS17].

In the traditional monolithic style of software development, an entire software application gets packaged as a single unified unit, frequently bundling various functionalities carrying diverse responsibilities as part of the single application. However, with this approach, as applications grow and become more complex, they can quickly become incomprehensible "big ball of mud" [RS16]. In contrast, the core idea behind MSA style of software development is decomposing a single software application into a suite of smaller individual services known as a microservice. Each microservice is designed to be self-contained and has its realm of responsibility around a single business capability. It runs a separate process and can be independently deployed [LF] [RS16]. Some other characteristics of MSAs include decentralised governance and data management, designed for failure, infrastructure automation and evolutionary design [LF].

While MSAs offer several benefits like independent deployment, scalability, and technology heterogeneity. They also have drawbacks, such as increased design and operational complexities, security, testing, data consistency when using a database per service approach, monitoring and troubleshooting [New21]. In addition, due to the independent

and isolated design of microservices and the distributed architectural style of MSAs, the need for inter-service Application Programming Interface (API) communication between microservices arises when realising complex functional requirements. In enterprise-scale microservice applications, the sheer magnitude of inter-service API communications between the hundreds and thousands of microservices becomes complex and challenging. While MSA are better suited for developing complex, scalable and evolving enterprise applications, designing them can be challenging [RS16]. Furthermore, any poor architectural decisions taken during the design stages can significantly impact various performance, maintainability and quality aspects of a software application [Kou+15]. It is something which cannot be solved just by using a more modern architectural style like MSA.

When exploring unknown domains, techniques like mocking, prototyping, and simulations can provide means to gain valuable insights. Such techniques are particularly beneficial when adopting modern architectural styles like MSAs to develop distributed applications as they generally involve complex design and communication structures spreading across numerous microservices, introducing high possibilities for failures [LF] [RS16]. At the same time, these techniques may not provide answers to everything but can undoubtedly lower the possibilities of certain uncertainties from our approach. They can help us acquire a practical understanding of varied characteristics of existing or to-be architectures by adding significant value to our development process and approach. For example, mocked simulations of MSAs during the design stages can provide a way to determine potential issues in the design earlier by identifying MSA anti-patterns like cyclic dependencies between microservices. We can also use such approaches to test the designed/migrated architectures with user workloads that mimic our production scenarios to identify bottleneck or hotspot microservices and take proactive measures to prevent issues in the future. The opportunities to effectively apply such mocked simulation techniques are limitless and need to be explored in the context of MSAs.

While some similar approaches exist in the domain of MSAs, based on our investigation, research gaps still exist that have yet to be researched to their full potential. We found studies where researchers have conducted investigations to simulate various characteristics of MSAs utilising user-provided simulation data. However, the simulation approaches presented in these studies solely carry out virtual non-realtime simulations of MSAs.

Furthermore, during our investigation, we noticed the existence of several published studies that focused on the development of MSAs using *Model-Driven Development (MDD)* rather than from the perspective of mocking MSAs. Based on our investigation into these studies, we identified specific research gaps in this domain, which include the limited research supporting the modelling and generation of technology heterogenous microservices, also known as *polyglot microservices* to mock polyglot MSAs and the lack of research related to automated composition, deployment and observability of mocked MSAs.

To address the identified research gaps, in this thesis, we investigate a MDD approach to mock observable polyglot MSAs. As part of this, we research the definition of a

metamodel to capture microservice data and their inter-service API communications. Because the inter-service communications between microservices can be captured and represented in different ways, in this thesis, we decided to capture them through *message sequence models* when we model functional requirements of MSAs. Furthermore, To offer real-time interaction with MSAs, we explore the transformation of concrete instance models of our metamodel into deployable technology heterogeneous mocked-up microservices. Finally, we explore the automated composition, deployment and observability of the mocked-up microservices to compose observable polyglot MSAs that we wish to use in simulating various modelled functional requirements.

1.1. Research Questions

To bridge the identified research gaps, we formulated the following set of Research Questions (RQs) to investigate as part of this thesis so that answering them would help us to explore mocking of MSAs through message sequence models.

- **RQ1:** *How would a metamodel that can capture attributes of microservices exhibiting inter-service API communications materialise to be?*

  As part of this research question, we first aim to identify the attributes of microservices required to create their mocked versions. With this knowledge, we explore the definition of a metamodel capable of capturing the identified microservices attributes and their inter-service API communications.

- **RQ2:** *How can we use concrete instance models of our metamodel to construct MSAs composed of mocked-up microservices that can replicate the behaviour of modelled inter-service API communications?*

  In the second research question, we focus on defining concrete instance models from our metamodel and exploring the creation of a prototype that can parse such models and generate mocked-up microservices. Our approach must also support automated composition and deployment of the generated microservices to create MSAs where microservices can communicate with their dependency services to imitate the behaviour of modelled inter-service API communications.

- **RQ3:** *How can we automatically visualise the behaviour of the composed MSAs?*

  Once the microservices are rendered and composed into MSAs, as part of the third research question, we wanted to explore automated support for the observability of the composed MSAs. We plan to investigate the collection of telemetry data like metrics, logs and traces that provide a way to visualise the behaviour of the generated microservices and gain more insights into other functional and non-functional aspects of mocked MSAs.

- **RQ4:** *What are the learnings and challenges associated with generating polyglot MSAs?*

Microservice architectural style supports the development of applications using polyglot languages [Gar18] [BHJ16a]. As part of this research question, we try to understand and capture our learnings and challenges associated with supporting multiple microservice generation strategies for various technologies to mock polyglot MSAs.

- **RQ5:** *What are the benefits and drawbacks of modelling large-scale MSAs using our metamodel?*

  Enterprise MSAs are complex, and modelling them can be challenging. We formulated this research question to understand and capture the benefits and drawbacks of utilising our custom model for modelling such large-scale microservice applications.

1.2. Goals and Contribution

The primary goal of this thesis is to explore the capability to mock MSAs through message sequence models. As part of this, we define a process to model microservices and their inter-service API communication and use this data to automate the process of generating, composing and deploying large-scale observable polyglot microservice applications. By achieving this, we aim to facilitate a simulation approach for MSAs that can be used by researchers, architects and developers in academia and industry to unlock novel methods of investigating, testing and observing different characteristics of MSAs without developing fully functional microservice applications.

This thesis offers the following four principal contributions:

- Our first contribution is a metamodel whose definition is independent of message sequence model representation and microservice technology. Its core purpose is to capture both the structural definition of microservices and the relational aspect of their inter-service API communications in an input and technology-agnostic manner.

- Our second contribution is a prototype application capable of ingesting models conforming to our metamodel and generating, composing and deploying large-scale mocked MSAs in an automated manner.

- Our third contribution focused on introducing observability concepts as part of the generated and composed MSAs. It allowed us to collect various telemetry data to proactively observe the behaviour of the mocked MSAs.

- In our fourth contribution, we mock large-scale polyglot MSAs and capture our learnings and challenges.

1.3. Thesis Structure

The remainder of this thesis is structured as follows. We introduce foundational concepts relevant to our study in Chapter 2. In Chapter 3, we categorically review all the related work we identified as relevant and connected with this thesis to establish an understanding of the existing research and to outline the differentiating factor to our contributions as part of this thesis.

In Chapter 4, we first describe the phases of our research process, each aiming to tackle a part of our overall research objective and summarise the scope of this thesis. Second, we outline the conceptual approach to mocking MSAs. Third, we describe the process of generating microservices applications. Finally, we discuss the process followed to define our metamodel capable of capturing data of MSAs. In Chapter 5, we first examine the design of our developed prototype application that generates microservices. Next, we discuss various MSA design patterns used for composing, deploying and observing the generated microservice applications. Chapter 6 discusses the different notations, tools and frameworks used to define our metamodel's concrete instance models and develop the prototype application.

In Chapter 7, we detail our evaluation strategy and associated tasks. Next, in Chapter 8, we capture our evaluation results, examine and interpret the outcomes, and accordingly perform their reasoning. In Chapter 9, we answer our research questions and then discuss the threats to the validity of our findings. In the final Chapter 10, we summarise and draw our conclusion on this thesis and list the prospects of extending this thesis as part of future work.

2. Foundations

> Perfecting oneself is as much unlearning as it is learning.
>
> Edsger Dijkstra

Contents

This chapter provides the relevant background knowledge to understand the foundational concepts essential for this thesis.

2.1. Communication in Microservice Architecture (MSA)

MSAs are composed of numerous microservices that communicate with one another. Due to their distributed nature, they communicate with one another on the network level using lightweight synchronous or asynchronous inter-service communication protocols such as *Hypertext Transfer Protocol (HTTP)*, *gRPC Remote Procedure Call (gRPC)* or *Advanced Message Queuing Protocol (AMQP)* [LF]. As discussed in Chapter 1, in this thesis, we use *message sequence models* to capture inter-service API communication between microservices when modelling functional requirements of MSAs.

Definition 2.1.1: Message Sequence Models

Message sequence models illustrate the sequential ordering of all the inter-service API requests between microservices in an MSA to realise a specific functional requirement. One can represent these models in different styles, for example, in a written textual form, such as a *Domain-Specific Language (DSL)* or a diagrammatic form, like Unified Modeling Language (UML) sequence diagrams.

Figure 2.1, illustrate an example of a message sequence model represented as a UML sequence diagram depicting the various inter-service communication requests that occur between microservices “authentication-service”, “user-service” and “session-service” to realise login functional requirement in an E-commerce application.

From the message sequence model shown by Figure 2.1, we can see that microservices are independent and self-contained, as a result, in realising specific functional

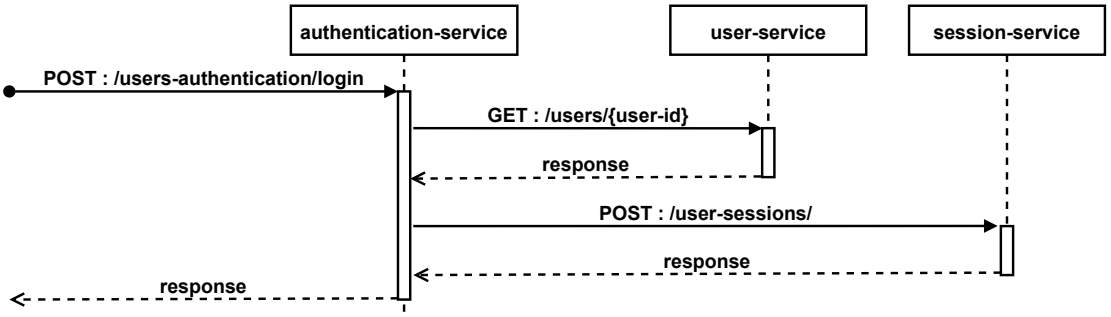

Figure 2.1.: Message sequence model illustrating the login functional requirement.

requirements, particularly complex ones, coordinated communication between multiple microservices becomes inevitable in any enterprise-scale microservice applications. This dependency on other microservices to complete a single business requirement can lead to a form of coupling known as domain-based coupling [New21] or efferent coupling [Ric16]. In this thesis, we refer to such inter-service communication dependencies as *inter-service API dependencies*.

Definition 2.1.2: Inter-service API Dependency

When two microservices communicate with each other via the *Remote Procedure Invocation (RPI)* pattern [Ric19] [RRS17], we say there exists an *inter-service API dependency* relationship between the pair of microservices.

For example, from Figure 2.1, since "authentication-service" consumes a "GET" API offered by "user-service", we say there exists an inter-service API dependency between them. Similarly, there is also inter-service API dependency between "authentication-service" and "session-service" as "authentication-service" consumes a "POST" API offered by "session-service".

Furthermore, coordinated communications in complex functional requirements can involve multiple API requests across numerous microservices. This results in a series of API requests occurring sequentially. In this thesis, we refer to such a series of API requests as *inter-service API dependency sequences*.

Definition 2.1.3: Inter-service API Dependency Sequence

Inter-service API dependency sequence is the sequential ordering of inter-service API dependencies between microservices to realise a specific functional requirement.

From the message sequence model illustrated by Figure 2.1, we can see that an API request made to the "authentication-service" for login results in the invocation of multiple inter-service API dependency requests, first to the "user-service" and second to the

"session-service". This invocation of inter-service API dependencies following a sequential order creates an inter-service API dependency sequence. Capturing this information becomes essential in this thesis when mocking MSAs.

2.2. Model-Driven Development

Model-Driven Development (MDD) is a software engineering approach that advocates the usage of models for capturing and managing essential complexities of software systems and reducing their accidental complexities [Rad+18] [Com+20]. We follow the view of Whittle, Hutchinson and Rouncefield and consider MDD to be a subset of *Model-Driven Engineering (MDE)* with a focus on generating implementations using models [WHR14] [Ame10]. In software engineering, a model is an abstraction of a software system in its reality from various viewpoints. We can use models to study and capture complex software more abstractly and generate software code and documentation artefacts in an automated manner [Rad+18] [Com+20].

Some of the generic perceived benefits of adopting MDD include abstraction of domain-specific complexities via model, increase in productivity, automated code generation that can embed any formulated architectural constraints and domain-specific best practices and design patterns to improve code quality [WHR14] [Rad+18].

Specific characteristics displayed by MSAs, like technology heterogeneity and resilience, facilitate the adoption of supportive MDD techniques like automated code generation when developing microservice applications [Rad+18]. Furthermore, MDD presents ways to address some of the challenges faced when designing MSAs [RSS18]. In this thesis, we aim to leverage some of these characteristics of MDD to mock MSAs through message sequence models.

2.3. Observability of MSAs

MSAs is an architectural style that fosters the development of distributed cloud-native applications [BHJ16b] [IS21]. Consequently, distributed systems bring complexity and comprehending the behaviour and state of such running systems is challenging. Many systems integrate monitoring tools to ease some of these complexities. These tools operate in parallel with such systems to supervise, record and analyse their operation [IEE90].

While monitoring can be a great way to collect and analyse application metrics, it becomes necessary to understand systems' behaviour. *Observability* provides this opportunity. It is a term that stems from control systems theory [Gop93] [Nie+19]. In software engineering, observability allows us to gain insights into software systems' working from the outside without understanding their inner working structure. While traditional monitoring is more reactive, observability is more about proactive systems monitoring [CK21]. The three pillars of observability include traces, metrics and logs [Gat21]. Capturing microservices data related to these aspects in this thesis can help us holistically map MSAs' runtime behaviour when mocking them.

3. Related Work

> If we knew what it was we were doing, it would not be called research, would it?
>
> Albert Einstein

Contents

This chapter presents various existing academic works and other literature related to our thesis topic that were collected by surveying diverse research databases and the internet. We discuss each scholarly work presenting the author(s) contribution(s) to the paper. Finally, we draw a comparison outlining how our work is distinguishable and what additional value it brings.

The first section discusses scholarly works and open-source tools concerning code generation using MDD techniques in the context of MSAs. The second section deals with scientific works which use simulation tools to simulate MSAs. The final section presents scholarly works on observability in MSAs.

3.1. Model-Driven Development of MSA

3.1.1. Academia

Sorgalla et al. [Sor+18] propose AjiL, a tool developed based on *Eclipse Modeling Framework (EMF)* for graphical modelling and generation of MSAs. It comprises an editor and a single code generator for generating Java Spring-based microservices. The authors also define AjiML, a DSL to capture microservice data. The graphical editor allows for simple and easy modelling of MSAs, including supportive infrastructure services like API Gateway, Discovery, Security, and User-Management Service. Nevertheless, the MSA generation process is not so trivial as the editor, and the code generator are not interconnected. The AjiML model generated by the modelling editor must be manually relocated to a specific code generator's project directory to generate MSAs.

Wizenty et al. [Wiz+17] propose the tool Maven Archetype for Generating Microservice Architectures (MAGMA). It uses the Maven build management system and seeks to

accelerate and ease the development of MSAs. This tool allows for generating microservice template projects and, like AjiL, the generation of supportive infrastructure services. It also provides a basic *User Interface (UI)* that offers an optional selection of infrastructural services and for creating service templates. Like AjiL, this tool supports the generation of templates for only Java Spring-based microservices. Furthermore, it only generates template projects for services, and users must manually code the microservices API details later in the generated service templates.

In the following study, Lange et al. [Lan+16] propose Community Application Editor (CAE), a tool for near real-time collaborative modelling and generation of microservices-based web applications. CAE offers UI based modelling of web applications. It comprises the following modelling views: a view to model the front-end components of a web application, a view to model the back-end microservices, and finally, a view to specify the communication between the front-end and back-end applications modelled as part of the previous views. The tool also provides various pre-defined front-end and back-end modelling elements that the users can use in respective modelling views to design the front-end and back-end of web applications and to interconnect them. The bundled generator in the tool then uses these modelled view definitions to generate corresponding front-end and back-end projects on GitHub [1]. This tool offers an excellent way for users to develop web applications collaboratively. Nevertheless, it requires manual effort to compose and deploy the generated web applications and similar to AjiL and MAGMA, this tool supports microservice generation only for a single programming language.

In the following study by Terzić et al. [Ter+17], the authors present MicroBuilder, a tool for modelling and generating *Representational State Transfer (REST)*-based MSAs. Similar to AjiL, this tool also uses EMF for modelling of MSAs. The tool comprises two modules, MicroDSL a DSL for modelling MSAs and MicroGenerator, which uses MicroDSL to generate Java Spring-based microservices. Furthermore, it pre-bundles supportive infrastructure services during the generation process. The authors also extend this tool in [Ter+18b] to support graphical modelling of MicroDSL similar to AjiL and add basic monitoring capabilities. As it uses EMF, it faces challenges similar to AjiL, and the authors designed the tool and DSL to support the generation of microservices only for a single technology. To overcome some of these drawbacks, the authors present a positional study in [Ter+18a] to discuss the challenges associated with modelling, development and deployment of MSAs and propose a conceptual approach to enhance MicroBuilder.

Düllmann and van Hoorn [DH17] present a generative platform for benchmarking performance and resilience of MSA environments under controlled conditions. The authors propose a DSL and its metamodel to capture microservice data and a generator tool that uses the DSL models to generate microservices in Java. This study also used EMF for modelling and code generation. In addition to the generated microservice code, the generator tool also generated Kubernetes [2] deployment artefacts for the generated mi-

[1] https://github.com/ (accessed on 18.05.2023)
[2] https://kubernetes.io/ (accessed on 18.05.2023)

croservices. In the proposed approach, following supporting services, Apache JMeter [3] and Kieker [4] are deployed manually later for user workload simulation and application monitoring. It enables running performance benchmarks on the MSAs.

Sorgalla et al. [Sor+20] present Language Ecosystem for Modeling Microservice Architecture (LEMMA), a language ecosystem that defines a set of textual *Architecture Modelling Languagess (AMLs)* for modelling MSAs from different architectural viewpoints. The authors have implemented all of LEMMA's AMLs using EMF. Each modelling language in LEMMA models data related to specific viewpoints like domain, service and operation, consequently targeting particular stakeholders and allowing for holistic modelling of MSAs [RSZ19] [Rad+21]. At the same time, to foster collaboration among the team, these modelling languages are also interconnected [Rad+19]

Montesi, Guidi and Zavattaro [MGZ14] propose Jolie, an open-source service-oriented programming language which allows defining microservices in a contract-first approach. Like any other programming language, it defines its own syntax and language constructs and comes with a language interpreter written in Java. Services written in Jolie comprise two parts, namely behaviour and deployment. The behaviour captures the code representing the functionality offered by the service. It includes primitives for defining communication and computational constructs. The deployment part captures the information necessary for establishing communication between services and architectural primitive data for specifying the application's structure.

A recent study by Suljkanović et al. [Sul+22] propose Silvera, an open-source declarative language developed for the domain of MSA. It comprises SilveraDSL, a modelling language for microservices and supportive infrastructural services, and a template-based code generator that generates microservices using SilveraDSL models. The authors have created a compiler based on textX [5] to parse SilveraDSL. Silvera supports the registration of new code generators as plugins and additionally evaluates the architecture of the designed MSA based on tailored MSA metrics.

In all of the examined studies, we see none exploring the concept of mocking MSAs through message sequence models, which is the primary differentiation of our research in this thesis. In addition, out of the many tools and modelling languages discussed, only Silvera and LEMMA support modelling microservices in polyglot technologies. Nevertheless, their case studies did not incorporate polyglot MSAs, which we include as part of our work. Furthermore, only studies [DH17] and [Ter+18b] examined some form of monitoring of the composed MSAs. But, none of the studies researches automated observability of the composed MSAs, which we investigate and contribute to in this thesis. Finally, among the discussed tools, we observed an overall lack of automated support in composing MSAs after the code generation step. Our work uniquely contributes here by supporting automated build, composition and deployment of MSAs using our prototype after the code generation process.

[3] https://jmeter.apache.org/ (accessed on 18.05.2023)
[4] https://kieker-monitoring.net/ (accessed on 18.05.2023)
[5] http://textx.github.io/textX/3.1/ (accessed on 18.05.2023)

3.1.2. Open-source Tools

JHipster [6] is an open-source development platform that allows generating, developing, and deploying fully functional web applications and MSAs containing front-end and back-end components. It defines a JHipster Domain Language (JDL) to model MSAs. Modelling of MSAs is easy using JDL because of its user-friendly syntax. JHipster generates executable microservice applications by transforming the JDL specifications in an automated manner. This tool supports multiple front-end technologies and a few specific back-end technologies that are Java-based, Node.js or .NET. Nevertheless, one needs to manually install respective generators for each technology to create microservice in that technology and compose them manually to create MSAs.

OpenAPI Generator [7] is an open-source application that uses OpenAPI [8] definitions to generate API client libraries, server stubs and documentation in an automated manner. It currently has code generators for numerous languages and technologies. The generated stubs can be used for prototyping applications or in mocking API functionalities. Nevertheless, the OpenAPI definition accepted by OpenAPI-Generator does not support capturing the inter-service API dependencies of microservices which is necessary to model the API communications of the MSAs.

The differentiating factor to our work in this thesis is that we focus on mocking MSAs through message sequence models, for which these open-source tools are not designed. Furthermore, OpenAPI models do not support modelling inter-service communication, which we address in our metamodel defined in this thesis. In the case of JHipster, there is only a focus on specific technologies and a lack of support for automated composition and observability of MSAs. In our work, we address these as part of our prototype, which would support heterogeneous programming languages and technologies and offer an automated way to build, compose and deploy observable MSAs.

3.2. MSA Simulation

Khan et al. [Gok+21] present PerfSim, a discrete-event simulation platform that approximately predicts the performance of cloud-native microservice chains for user-defined scenarios. The tool supports a set of modelling elements to model different aspects of cloud-native systems to capture the necessary input data for simulation. The authors also propose a systematic performance modelling approach outlining the creation of performance profile models to represent service chains, endpoint functions, hosts and network topology based on real microservice systems that are to be simulated.

Zhang, Gan, and Delimitrou [ZGD19] propose µqSim, a queuing network simulator tool designed to simulate interactive microservices. It requires simulation data concerning internal microservice details, inter-microservice topology, path indicating the flow of requests across microservices, system and resource information, and client requests load

[6] https://www.jhipster.tech/ (accessed on 18.05.2023)
[7] https://openapi-generator.tech/ (accessed on 18.05.2023)
[8] https://www.openapis.org/ (accessed on 18.05.2023)

pattern as part of multiple input files to run simulations and capture their operational performance.

The two discussed studies focus on performing a virtual event or queue-based simulation of microservice systems by collecting the necessary simulation data based on existing real systems. In contrast, this thesis focuses on generating observable mocked-up microservices that can be composed and deployed in an automated manner to mock MSAs. It allows us to interact with the mock MSAs and observe their real-time behaviour. Our approach also facilitates researchers to explore novel real-time simulation techniques using the mocked MSAs.

3.3. MSA Observability

Usman et al. in [Usm+22], conducted a comprehensive survey on the observability in the domain of distributed edge environments and container-based microservices. Their study investigated the requirements, best practices and challenges related to the observability of distributed systems. Their study revealed a collection of available state-of-art monitoring and observability tools from academia and industry, which they compare on set criteria. It also captures the fundamental characteristics essential to realise observability. Furthermore, their study also states that, unlike extensive research in the monitoring domain, the research on the observability of distributed systems like MSAs is still evolving and is primarily driven by industries.

Niedermaier et al. [Nie+19] conduct qualitative research on the observability and monitoring of distributed systems in the industry. In this paper, the authors perform semi-structured interviews with software professionals from various organisations to understand and capture the challenges and best practices related to monitoring and observability of distributed systems. Their study identified nine challenges and fourteen stakeholder requirements related to the monitoring and observability of distributed systems. For each, they also gathered possible solutions from the interview participants and presented the results. They conclude that this topic is not just technical but cross-cutting and strategic oriented and that there is a need for additional good practices for aligning technical and business objectives to facilitate effective development and operation.

Marie-Magdelaine, Ahmed and Astruc-Amato [MAA19] conducted a study to demonstrate an observability framework they implement in collaboration with an industrial partner. The study offers insights into this framework's architecture and various elements and outlines its capabilities. The authors also capture how one can implement a similar framework by integrating multiple open-source tools.

Heger et al. [Heg+17] propose a study to holistically monitor and observe an application's performance on various levels of abstraction during its operation using continuous Application Performance Management activities. It outlines the multiple activities involved in this approach and the tooling support available for its realisation.

The existing studies only conduct surveys to identify and discuss the concepts and challenges behind adopting observability in distributed systems, including MSAs, with only one providing a high-level demonstration of realising such an observability framework. In

contrast to these studies, in our thesis, we focus on generating observable mock MSAs in an automated manner to capture various real-time telemetry data and understand the behaviour of the composed distributed system during its runtime. Nevertheless, we draw inspiration from the existing research to gain more conceptual knowledge about observability in MSAs.

4. Research Approach and Solution Concepts

> Being abstract is something profoundly different from being vague... The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.
>
> Edsger Dijkstra

Contents

In this chapter, to begin with, we discuss our research approach and the scope of this thesis. Next, we present the process we defined on a conceptual level in order to achieve this thesis's vision of mocking MSAs. As part of this, we provide an overview of the process model and briefly discuss the individual execution steps. Secondly, we examine in detail the process we defined for generating microservice applications. Finally, we also present our modelling approach to creating our metamodel.

4.1. Research Phases and Scope

We divided our research into four phases, each contributing incremental value to realise our final vision. Some of these phases consisted of multiple smaller objectives to achieve the milestones defined for that phase incrementally. We aimed to incorporate

the systematic software development lifecycle process to streamline our development approach for all the software development activities involved in our research [Som10]. As we had only six months for the thesis, we decided to adopt the combination of rapid prototyping and an iterative agile-based development process to create the first proof of concept quickly and then work towards stable incremental expansion and delivery of our research artefacts [Som10] [Rup10]. During the research, we adopted the Notion [1] web application as our collaboration platform. We used it for project planning and tracking activities like defining roadmaps and milestones, for task management using Kanban boards and as a documentation space to capture all our discussions and design decisions concerning the thesis activities for its entire duration. Next, we briefly discuss each phase's research focus and milestones.

4.1.1. Phase 1: Microservice Attributes Identification and Metamodel Definition

In the first phase of our roadmap, we planned to identify various microservice attributes and define our metamodel capable of capturing the identified microservice attributes and inter-service API dependencies. Developing this metamodel would contribute to achieving our first milestone.

4.1.2. Phase 2: Generate and Compose Microservices to Mock MSAs

The second phase consisted of two focus areas. The first focus was to capture MSA data as part of concrete domain-specific models conforming to our defined metamodel and to identify an approach to transform these models into mocked microservices using a prototype application. Our second focus was investigating how to compose and construct MSAs using the generated mocked microservices. Completing the two goals would allow us to achieve our milestone of creating mocked MSAs.

4.1.3. Phase 3: Observability of mocked MSAs

In this third phase of our research, we aimed to focus on understanding and incorporating MSA observability design patterns to comprehend the behaviour of our mocked MSAs. Achieving this milestone would allow us to gather vital run-time insights on our mocked MSAs.

4.1.4. Phase 4: Generating and Mocking Polyglot MSAs

In our research's fourth and final phase, we focused on the support for generating polyglot microservices. It allows us to reap the benefits of numerous programming languages and technologies and select the most appropriate one for each microservice based on our

[1] https://www.notion.so/ (accessed on 18.05.2023)

use case. Generating polyglot microservices would allow us to achieve our phase milestone of composing and mocking observable polyglot MSAs, which are steadily becoming prevalent in industries.

4.1.5. Scope

In this thesis, we restrict the inter-service API communication between the generated microservices in the mocked MSAs to using HTTP. All APIs adopt the RESTful design principles for managing a resource's state. In addition, all the REST API requests between the microservices will be synchronous and blocking in nature, such that any requests received by microservices are processed immediately, and further code execution is blocked on the clients until the server returns a response or an error message. As well as, the generated microservices are strictly mocked versions of the existing/envisioned applications that do not encapsulate any business logic. Furthermore, this study was limited to simulating MSAs where the inter-service communication between microservices occurs only via the RPI strategy and not via message queueing or publish and subscribe patterns.

4.2. Conceptual Approach to Mocking MSAs

After understanding the different phases and scope of our research, in this section, we provide a conceptual understanding of the strategy adopted in the thesis to generate microservice applications to mock MSAs. Figure 4.1 offers a high-level illustration of our process outlining the necessary prerequisite data and the various artefacts that are involved in this mocking process.

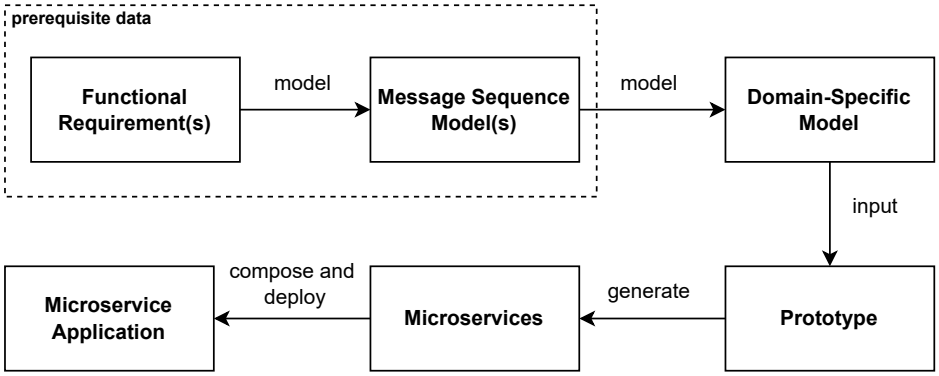

Figure 4.1.: A conceptual approach to generating microservice applications that can be used to mock MSAs.

- **Prerequisite data:** Firstly, we begin with the premise that the required prerequisite data is available in the specified format. In our case, this would be message

sequence model(s) that can be modelled in varied formats illustrating distinct functional requirement(s) of MSAs under study. These models would allow us to infer the inter-service API dependencies that exist between microservices.

- **Domain-Specific Model:** Since the message sequence models can have varied representations, we needed a representation format independent of the message sequence model representation. Furthermore, we realised that just having information on inter-service API dependencies would not be sufficient to generate microservices. We required a way to capture additional attributes of microservices in order to model them. For these reasons, we aim to define a metamodel based on the identified microservice attributes and create domain-specific models conforming to this metamodel. This way, these models would be independent of the message sequence models' representation. In addition, as they conform to our metamodel, they would also be capable of capturing additional attributes of microservices besides the information gathered on the sequential ordering of inter-service API dependencies derived from message sequence models. We discuss the metamodel definition process and its outcome in Section 4.4.

- **Prototype:** In our next step, we aimed to transform the domain-specific model containing the necessary MSA data to executable code. To achieve this, first, we planned to develop a prototype application capable of taking the domain-specific model as its input.

- **Microservices:** Once we had the MSA data from the domain-specific model inputted to our prototype, as the next step, we wanted to explore microservice code generation using MDD to generate deployable microservices. To achieve this, we planned to develop custom code generation logic as part of our prototype to parse MSA data and generate microservices.

- **Microservice Application:** Finally, in order to mock MSAs, we planned on composing the individual generated microservices to construct large-scale microservice applications such that the resulting architecture of these applications are capable of delivering the modelled functional requirement(s).

We gained a more profound understanding of our conceptual approach during the thesis. Consequently, it helped us to refine our approach and associated inputs and outputs. The following section discusses the process of generating microservice applications to mock MSAs.

4.3. Generating Microservice Applications

After getting a high-level understanding of our conceptual view to mock MSAs, in this section, we present the precise sequential flow of execution involved in the process of generating mocked microservices and composing them to construct elaborate large-scale microservice applications. We explain the overall execution flow by splitting it across

multiple parts and mapping them to our current knowledge of our conceptual approach to understand the generation process efficiently.

To begin with, the Figure 4.2 represents the first part of the microservice generation flow, where we use the prerequisite data to manually model a representation of our domain-specific model as per our metamodel, which is discussed further in Section 4.4. We named this domain-specific model discussed in our conceptual approach as Microservices API Dependency Model (MAPI-DM).

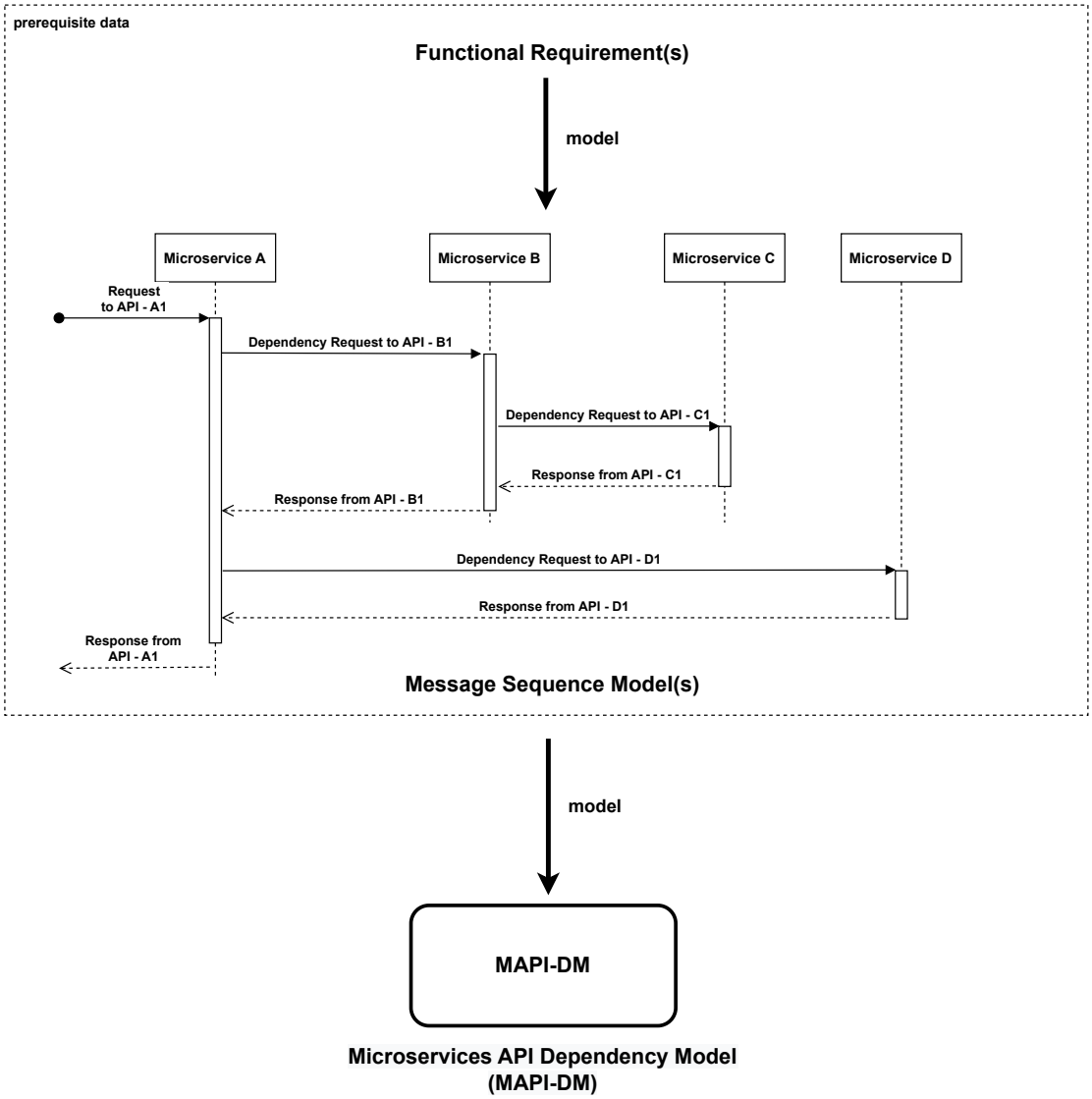

Figure 4.2.: The manual transformation of the prerequisite data is transformed into MAPI-DM

Definition 4.3.1: Microservices API Dependency Model (MAPI-DM)

Microservices API Dependency Model (MAPI-DM) is designed to capture the structural definition of microservices and the relational aspect of their inter-service API dependencies in sequential order.

In comparison to our conceptual approach, the prerequisite data remained unaltered. Accordingly, we expected MSAs' functional requirement(s) to be modelled as message sequence model(s). For example, the Figure 4.2 shows how a message sequence model can be represented in a diagrammatic form as a UML sequence diagram. It is essential to understand that irrespective of the representation format of the message sequence models, the inter-service API communications between microservices must be captured in a sequential order to mock the modelled functional requirement precisely. We use this information on the sequential ordering of communication between microservices captured as part of message sequence models to infer each microservice's inter-service API dependencies. Next, for each microservice, we combine the information on their inter-service API dependencies with other microservice-specific attributes as outlined in Section 4.4.2 in order to create MAPI-DM. Furthermore, in Section 4.4.1, we also present our approach to designing MAPI-DM.

In the next step of the microservice generation process represented by Figure 4.3, we needed a way to process MAPI-DM. We plan to achieve this by developing the prototype application mentioned in our conceptual approach. This prototype would have embedded business logic to parse and process MSA data modelled as MAPI-DM.

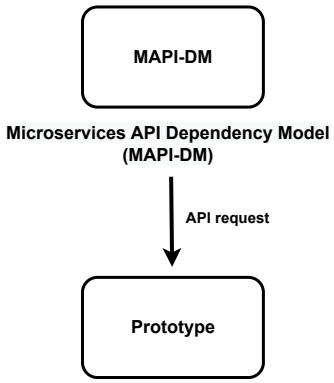

Figure 4.3.: The input of MAPI-DM to the prototype application via an API request.

As a next step, we had to solve the generation of mocked microservices. To achieve this, we planned to embed code generation logic in the prototype that we would develop, allowing it to use the processed MAPI-DM data from the step before to generate microservices for varied technologies and programming languages as depicted in Figure 4.4. This capability also ensures polyglot language support, one of the many essential advantages of adopting the microservice architectural style for developing enterprise software

applications. In addition, we also wanted to understand the behaviour of the generated microservices. Our planned approach to realising this is to use the prototype to instrument each generated microservice with a telemetry agent that can capture and export the microservice's various telemetry data like metrics, traces, and logs.

Furthermore, creating architectures adopting the best practices of MSA becomes essential for mocking any enterprise-scale microservice applications. As part of this, we wanted to adopt a way to avoid clients having to interact directly with individual microservices and also a way to allow each microservice to be discoverable to others within the MSA network. Therefore, as part of the generation process, we intended to pre-bundle the generation of specific supportive infrastructural microservices like an API-Gateway service that we can use to provide clients with a unified interface to interact with the microservices of the mocked MSAs and also a Service Discovery which we can use to facilitate microservices to discover other microservices within the MSA network.

Besides this, we also wanted to compose and deploy the generated microservices automatically. Our planned approach was to generate an MSA composition tool artefact and a deployment artefact using the same prototype as depicted in Figure 4.4.

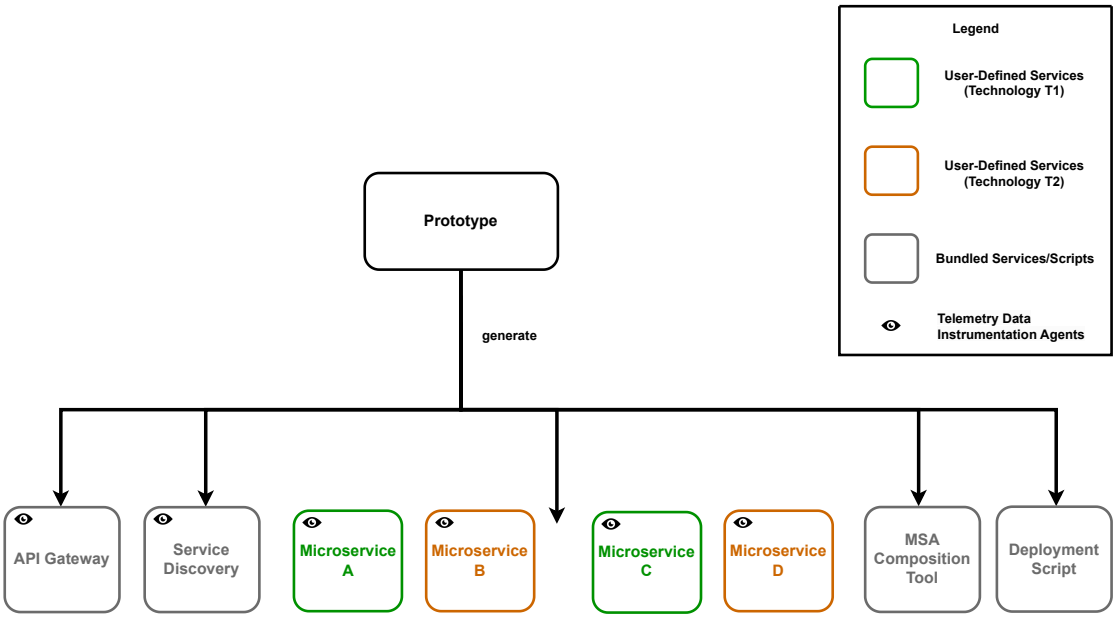

Figure 4.4.: Generation of various types of microservices by the prototype after parsing the MAPI-DM input.

In the final step of this generation process, as illustrated in Figure 4.5, we wanted to use the generated microservices to create large-scale microservice applications that we can use to mock MSAs. For this, we intended to use the MSA composition tool artefact generated by the prototype in the previous step to build and compose all the individual generated microservices into a more elaborate large-scale microservice application that we can deploy under a single MSA network. After this, by using the deployment script

the prototype generated in the previous step, we plan to deploy the composed microservice applications to mock the modelled MSAs and use it for simulation purposes.

In addition, we wanted to find a method to automatically collect and visualise the various telemetry data exported by the telemetry agents that are instrumented in generated microservices to create highly observable MSAs. Therefore, in this step, we planned to use the same deployment script to deploy an open-source MSA observability tool possessing a telemetry data collector capable of collecting the telemetry data of microservices exported by the telemetry agents.

In Figure 4.5, we have represented the infrastructural view of how an example microservice application after its composition and deployment using the composition tool and deployment script would look. Observing this view, we can infer that the resulting MSA would comprise the generated microservices, including the supportive infrastructural services and the open-source MSA observability tool, all deployed on a single network. The supportive infrastructural services include the API-Gateway service and the Service Discovery, whose purpose we have outlined in the step before. The multi-coloured and textured arrows represent the envisioned communication or data exchange between microservices and ancillary services to accomplish different goals like inter-service API communications, service registration or telemetry data collection. Next, we look at the various arrow types and their purpose.

- **Filled Red Arrow:** These arrows represent the various communications with the mocked MSAs having their origin external to the MSA network. These would be client requests interacting with the microservices of mocked MSAs or the MSA observability tool.
- **Filled Purple Arrow:** These arrows represent the inter-service API requests between microservices that would be internal to the MSA network. We would derive the collection of permitted communications between microservices from the data captured as part of MAPI-DM.
- **Dashed Blue Arrow:** This arrow collection represents the microservices' requests to register/deregister themselves from the Service Discovery's registry upon their startup/shutdown after deployment/un-deployment. We further discuss the service discovery MSA design pattern in Section 5.2.1.
- **Dashed Pink Arrow:** This arrow cluster represents the export of telemetry data from the telemetry agents instrumented to every generated microservice to the telemetry collector that would be present in the deployed open-source MSA observability tool. We discuss more on the concept of MSA observability and its associated design patterns further in Section 5.2.4

This infrastructural view also depicts a hypothetical example of how complex functional requirements generally require a sequence of API interactions between microservices for their realisation. After acquiring a solid foundation on the overall microservice application generation process and gaining knowledge on the different artefacts involved in this process, in the next section, we discuss the approach taken to defining MAPI-DM.

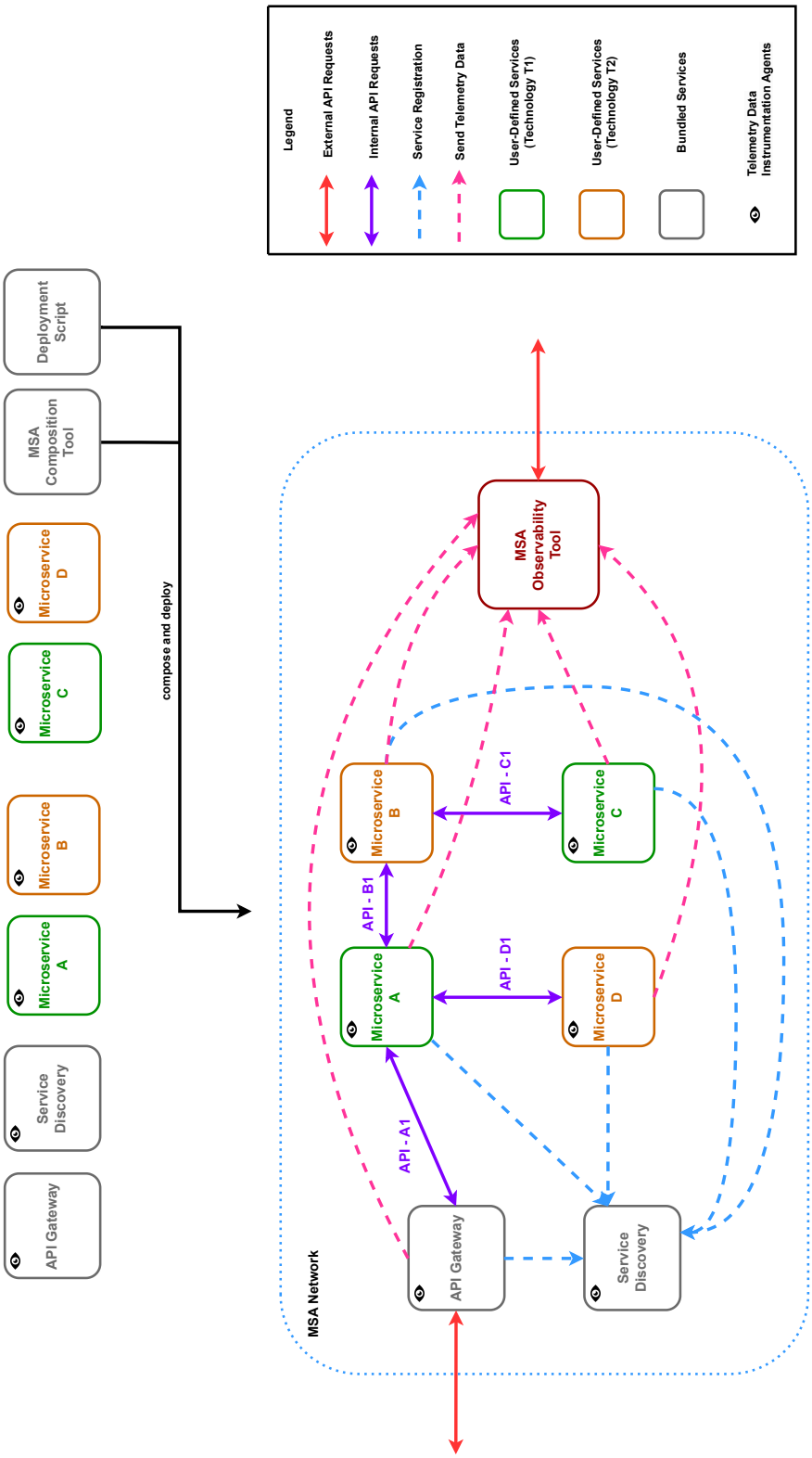

Figure 4.5.: An example architectural view of a composed and deployed microservice application with integrated MSA observability tool.

4.4. Modelling Microservices API Dependency Model (MAPI-DM)

MAPI-DM is one of the core artefacts we introduced as part of this thesis. From the microservice generation process discussed in the previous section, we understood that MAPI-DM is fundamental to the overall generation process, capturing the data necessary for generating microservice applications. To understand further details of this model, we first look into how we defined it, followed by the metamodel of MAPI-DM.

4.4.1. Overview of MAPI-DM Definition Process

In our approach to defining MAPI-DM, we followed an iterative top to bottom approach following the steps as described below:

- First, we created an example message sequence model for a hypothetical functional requirement to model our prerequisite data. We have illustrated this example as part of Figure 4.6.

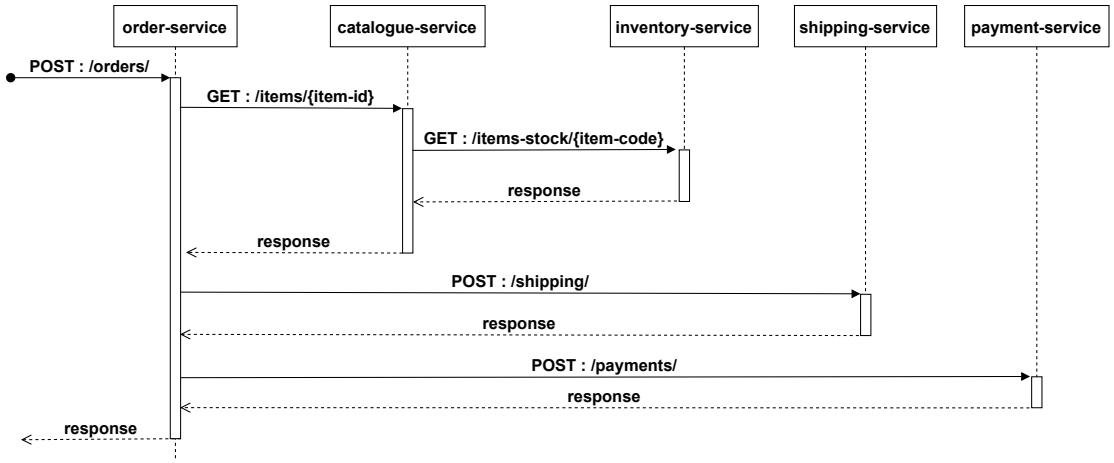

Figure 4.6.: Representative example of the message sequence model acting as our prerequisite data.

- Next, using this prerequisite data, we gathered the inter-service API dependencies for each microservice.

- As the next step, we focused on identifying the attributes necessary for modelling each microservice. Here we followed a recursive attribute identification approach. In this approach, we first determined the fundamental high-level attributes required to model microservices like their functional attributes API endpoints/interfaces and non-functional attributes like the microservice name and version. As the next step of this recursive approach, we recursively identified attributes for the identified

fundamental attributes, if any, to create a more granular decomposition of each attribute.

For example, as part of the API endpoint attribute, we identified further sub-attributes that model an API endpoint like the API name and the various path operations corresponding to the different HTTP verbs like GET, POST, PUT and DELETE. Next, we identified the need for attributes like the path value capturing path information such as "/items/item-id", parameters necessary for the operation, method name and dependencies to model each path attribute.

- We repeated this process over multiple iterations, and in each iteration, we added new attributes or modified existing attributes during the recursive attribute identification phase in order to improve and refine our model. For example, in our second iteration, to support polyglot microservice generation, we introduced the generator attribute as another fundamental attribute of microservices, allowing us to specify the technology to utilise when generating them.
- We stopped the iterations once we had identified and refined the attributes necessary to be able to generate deployable mocked versions of microservices for multiple technologies. As the next step, we expressed these attributes in a generic form to create the metamodel of MAPI-DM.

4.4.2. MAPI-DM Meta-Model

In this section, we describe the various components that compose MAPI-DM, their relationship with each other and their attributes. Figure 4.7 depicts the metamodel of MAPI-DM that we concluded in this thesis. Surely, in its current state, this metamodel is not final nor complete. It is extensible and adaptable based on the introduction of new feature requirements or modifications in the future. Next, we discuss each component of the MAPI-DM metamodel and their associated attributes in more detail.

- **MSA:** An MSA is composed of at least one microservice. We have yet to identify any common attributes across all the microservices. But it would be an ideal location to capture any externalised configurations that would be applicable to all the microservices in the architecture.
- **Service:** The Service component consists of elements applicable to an individual microservice. We have identified the following attributes under this component:
  - **serviceName:** This attribute captures the name of the microservice.
  - **generator:** This attribute captures the type of technology that will be utilised during the microservice generation process. This attribute has a default value set to "spring" if not specified.
  - **version:** This attribute represents the version of the microservice configuration. It was introduced to support configuration versioning in future. It has a default value set to "1" if not specified.

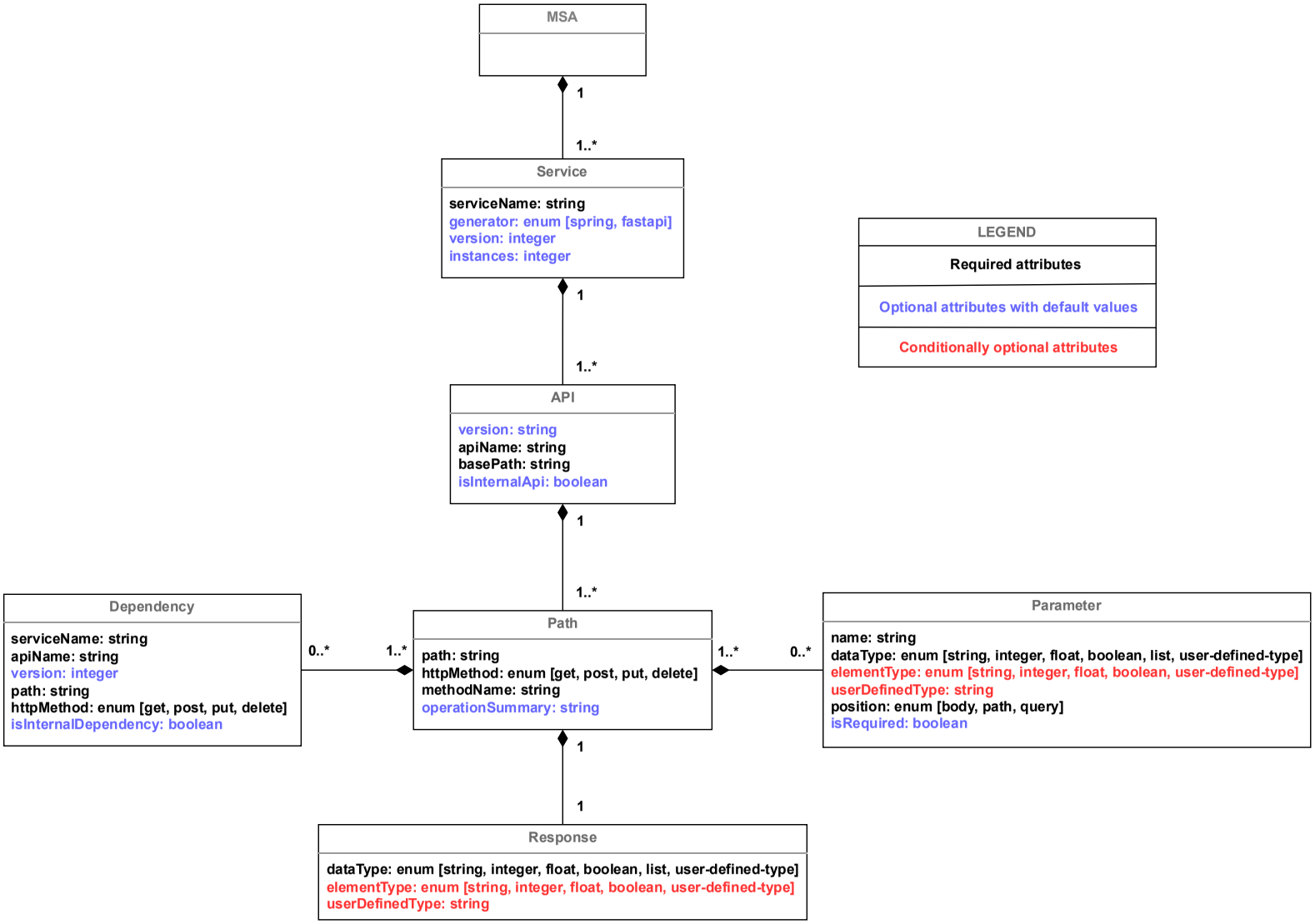

Figure 4.7.: State of the MAPI-DM metamodel as concluded in this thesis.

  - **instances:** The attribute records the initial number of to-be-deployed microservice instances. The default value for this attribute is set to "1" if not specified.

  A Service component must consist of at least one API endpoint that can be published externally or used for internal consumption. Next, we discuss the attributes of the API component.

- **API:** This component encapsulates all the common attributes of a microservice's API endpoint. Every API endpoint must be associated with a specific Service component representing a microservice. We have identified the following attributes as part of this component:
  - **version:** This attribute captures the version number of the API. A microservice can possess numerous APIs, each having a different version. It has a default value set to "1" if not specified.
  - **apiName:** We use this attribute to capture the API name.
  - **basePath:** APIs generally expose multiple paths, each performing a specific HTTP verb action like GET or POST. But these paths usually share a common path URL structure, for example, "/payments/" and "/payments/-paymentId". The basePath attribute captures a particular API endpoint's common portion of the path *Uniform Resource Locators (URLs)*. The base path value in our example will be "/payments". Separating this provides consistency and simplifies the path URL data across the API by preventing repetition.
  - **isInternalApi:** If set to "false", this boolean attribute indicates that the API is published externally and clients external to the MSA network can invoke it. On the contrary, if set to "true", it means the API is published internally and is accessible only to other microservices within the MSA network. If not specified, it has a default value of "false".

  Every API component has at least one Path component that expresses one of the allowed HTTP verb actions to manage the state of resources concerning the API.

- **Path:** Every Path component is associated with an API component, representing one of the allowed HTTP verb actions performed in order to manipulate an API resource's state. We have identified the following attributes as part of this component:
  - **path:** This attribute captures the distinct part of the path URL for a Path component realising a particular HTTP verb action.
  - **httpMethod:** The httpMethod attribute records the HTTP verb action performed by the API to manipulate the state of a resource.
  - **methodName:** This attribute captures a generic method name to represent the path action when translating the model to code.

  - **operationSummary:** This attribute captures a short description of the operation carried out by the path action for documentation purposes. It has a default value set to "operation-summary-not-defined".

  In addition to the above-discussed common attributes, every Path component consists of at most one Response component and zero or more Parameter and Dependency components. Next, we discuss these components' purpose and individual attributes in detail.

- **Response:** A Response component is always associated with a Path component. It consists of attributes that record the information on the payload returned by the API when the HTTP action represented by the Path component is executed. We have currently identified the following attributes as part of this component:
  - **dataType:** This attribute records the data type of the response that gets returned after the execution of a path action for an API endpoint. Currently, we support a predefined set of data types as documented in the metamodel in Figure 4.7.
  - **elementType:** This attribute captures the data type of the individual elements of a list. This attribute is conditional and is required if the dataType attribute is selected as a list.
  - **userDefinedType:** This attribute captures the data type of the user-defined types. This attribute is conditional and is required if the dataType attribute is selected as a user-defined-type.

- **Parameter:** The Parameter component captures the details of the various parameters the Path component requires during its execution. We have primarily identified the following attributes under this component:
  - **name:** This attribute records the parameter name.
  - **dataType:** This attribute records the data type of the response that gets returned after the execution of a path action for an API endpoint. Currently, we support a predefined set of data types as documented in the metamodel in Figure 4.7.
  - **elementType:** This attribute captures the data type of the individual elements of a list. This attribute is conditional and is required if the dataType attribute is selected as a list.
  - **userDefinedType:** This attribute captures the data type of the user-defined types. This attribute is conditional and is required if the dataType attribute is selected as a user-defined-type.
  - **position:** This attribute captures the parameter's position depending on whether the input is part of the request body, as a path variable in the path URL, or as a query parameter.

  - **isRequired:** This boolean-valued attribute indicates if the parameter is mandatory or optional. If not set, it has a default value of "false".

- **Dependency:** Finally, the Dependency component expresses the sequential ordering of the inter-service API dependency requests made during the execution of a particular Path component for a given API of a microservice. The different attributes part of this component are as follows:
  - **serviceName:** This attribute captures the name of the dependency microservice.
  - **apiName:** This attribute captures the name of the dependency API offered by the dependency microservice.
  - **version:** This attribute captures the version of the dependency API. It has a default value set to "1" if not specified.
  - **path:** This attribute captures the distinct part of the path URL for the executed Path component.
  - **httpMethod:** The httpMethod attribute records the HTTP verb action performed by the dependency API to manipulate the state of a resource.
  - **isInternalDependency:** If set to "false", this boolean attribute indicates that another microservice offers the dependency API, which makes it an inter-service API dependency. On the contrary, if set to "true", it means the dependency API is one of the APIs offered by the same microservice and is an internal API dependency. If not set, it has a default value of "false".

MAPI-DM Design Decisions

After understanding the metamodel of MAPI-DM, we now discuss the various design decisions we considered when defining it.

- **Microservice Dependencies:** When capturing dependency APIs for a microservice as part of MAPI-DM, we decided to capture only the direct API requests made by that microservice to other microservices in the sequential order of their invocation without involving any of the indirect API requests that further occur between other microservices to fulfil that request. For example, in Figure 4.6, when capturing the dependency APIs for "order-service", we consider only the direct API requests made by "order-service" to other microservices in their sequential order. It means our first dependency API entry would be the GET API request "/items/itemId" to the "catalogue-service", secondly the POST API request "/shipping/" to the "shipping-service", and lastly, the POST API request "/payments/" to the "payment-service". We need to understand here that we do not consider any indirect API requests made by other microservices as a dependency of "oder-service" when modelling the MAPI-DM data for "order-service". An example of such indirect API request would be the GET API request "/items-stock/item-code" made

by the "catalogue-service" to the "inventory-service" as illustrated in the same example in Figure 4.6.

Following this approach allowed each microservice to include only the fragment of dependency information that is necessary to model itself, thus keeping the dependency data compacted in the resulting model. Furthermore, we eventually model the required functional requirements when we realise these fragments of dependency API information for each microservice.

- **Data-type support:** In addition to any user-defined complex data types, we decided to support a subset of the standard set of data types available across many programming languages and used most commonly when developing REST APIs.
- **Input characters restrictions:** To keep input consistent and have predictability in code generation logic, we mandate input values of MAPI-DM to obey lowercase convention with additional rules on the allowed list of characters.

5. Design

> There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.
>
> Charles Antony Richard Hoare

Contents

Software design is an integral part of software engineering. It helps us define our application's overall structure and the organisation of its various components [Som10]. In this chapter, we first discuss the architecture of our prototype application, which we name *MSA-Gen*. Next, we discuss our different design decisions and the various design patterns we adopted during the development stages. Finally, we examine the various best practices and MSA design patterns we used to compose microservice applications and support observability.

5.1. Design of MSA-Gen Application

In this section, we first outline the MSA-Gen web application's architecture and discuss our design decisions.

5.1.1. Architecture

MSA-Gen, short for Microservice Architecture-Generator, is a web application that we developed according to the *Model-View-Controller (MVC)* architectural style[Fow+02].

In Chapter 6, we discuss the various technologies we used to develop the MSA-Gen web application. This web application exposes an API endpoint as captured in Figure 5.1 that accepts MAPI-DM as its input which contains data on microservices and their associated API dependencies. At the core of MSA-Gen is custom data transformation logic that parses and validates the MAPI-DM data using schema models conforming to our MAPI-DM metamodel. In addition, it also incorporates the following code generators that contain code generation logic for a specific technology. These code generators use microservice templates written using a templating engine and the parsed MAPI-DM data in order to render microservices and other supportive infrastructural services as illustrated in Figure 5.1. In addition to the microservice templates, each generator comprises generator-specific configuration files used to configure various generator parameters. Next, we briefly discuss each code generator and outline its purpose.

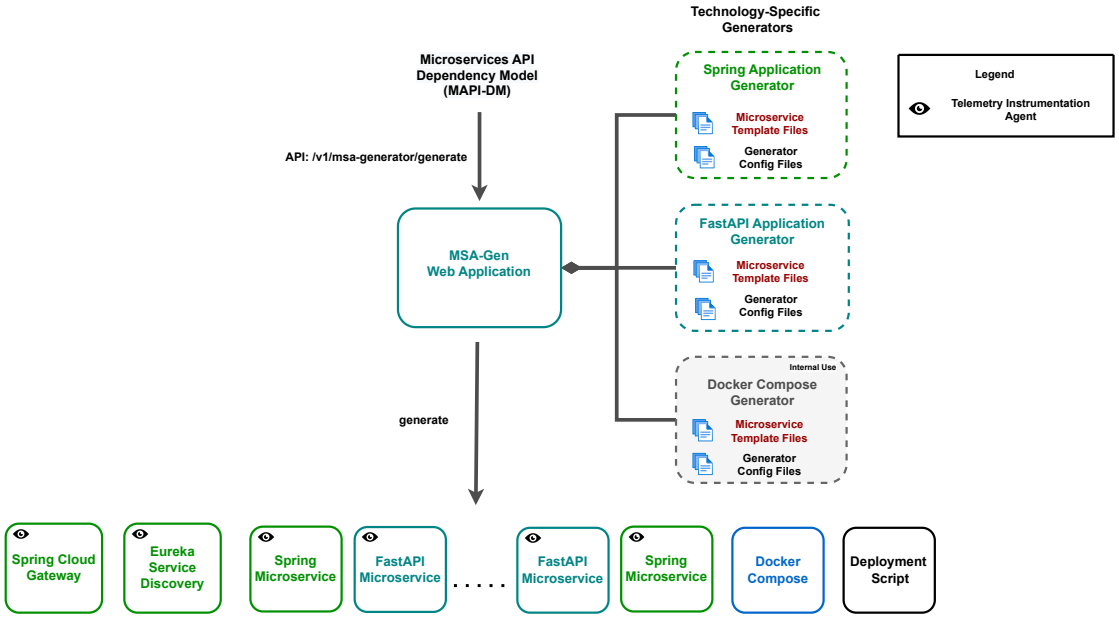

Figure 5.1.: The architecture of the MSA-Gen web application.

- **Spring Generator:** This generator is responsible for generating Spring Boot [1] technology microservices. Spring Boot technology allows the development of production-grade standalone applications. It comes bundled with embedded web servers like Tomcat, production-ready features like health checks and offers simplified build configuration by providing opinionated starter dependencies.

  Our Spring microservice generator comprises microservice template files written according to standard Spring Boot-based Java projects following the standard

[1] https://spring.io/projects/spring-boot (accessed on 18.05.2023)

technology-specific guidelines and best practices. This generator is also responsible for generating the following supportive infrastructural services:

  - **API Gateway Service:** This service provides a common external interface to access the different microservices and simple and resilient mechanisms to route and filter requests. We discuss this MSA design pattern and its realisation in this thesis in more detail in Section 5.2.1
  - **Service Discovery:** This service maintains a service registry to allow microservices to register and be discoverable to other microservices within the network. We discuss this MSA design pattern and present its realisation in more detail in Section 5.2.1

- **FastAPI Generator:** This generator handles the generation of all FastAPI technology microservices. It incorporates microservice template files written according to standard FastAPI-based Python projects following technology-specific guidelines and best practices.

- **Docker Compose Generator:** The Docker compose generator is a special generator that is available only for MSA-Gen internal consumption. It generates the Docker Compose file, which builds individual microservice containers and composes them as a single microservice application that can be deployed as part of a single shared network.

In addition to the different microservices and composition tools generated by the generators mentioned above, they also instrument each generated microservice with a telemetry instrumentation agent discussed in Section 5.2.4. This agent is responsible for collecting various telemetry data of the individual microservices. Furthermore, MSA-Gen also generates a deployment script for deploying the composed microservice application and also an open-source MSA observability tool discussed in Section 5.2.4. We discuss the technologies used to represent MAPI-DM data and to develop the MSA-Gen web application in greater depth in Chapter 6.

5.1.2. Design Decisions of MSA-Gen

When developing software applications, their design influences non-functional requirements, like performance, maintainability, scalability, and robustness. Developing an efficient design that structures the various components of a software application is by no means a trivial task, and it involves making many design decisions. This section presents the different design decisions taken when developing the MSA-Gen web application.

Object Oriented Design Patterns

Since their inception, design patterns have become the de facto standard when designing and developing software applications. They provide architects and developers with well-defined generic solutions to commonly recurring problems when designing and developing

new software applications. Next, we present the two Object Oriented design patterns that we incorporated in MSA-Gen during its development.

- **Strategy Design Pattern:** The Strategy design pattern belongs to the class of behavioural patterns as classified by Gamma et al., famously known as the Gang of Four (GoF). As per the authors' definition, the intent of this pattern is to create a family of encapsulated algorithms that are interchangeable in nature [Gam+94]. The participating components of this design pattern, as defined by the GoF, include the following:
  - **Strategy:** This component is the common interface for all the different concrete implementations of the strategy. The Context component uses the Strategy component to invoke any of the desired algorithms defined by a ConcreteStrategy component.
  - **ConcreteStrategy:** This component represents concrete algorithm realisations of the common interface component Strategy.
  - **Context:** This component uses one of the available ConcreteStrategy components via a reference carried by the Strategy component.

  Our realisation of the Strategy pattern represented in Figure 5.2 includes the "abstract_generator", acting as our Strategy component, which provides the common interface defining abstract methods that concrete generators must implement with specific generation algorithms. Apart from the common abstraction layer, we have created a second abstraction layer for the programming language represented by components "abstract_java_generator" and "abstract_python_generator", which act as secondary-level Strategy components. They allow defining a default implementation for any language-specific constructs or configurations that are common across all the concrete technology generators using that programming language. Next, we have the various concrete generators like "spring_generator", "fastapi_generator", and "docker_compose_generator" that act as ConcreteStrategy components implementing technology-specific microservice generation logic. Finally, we have "msa_generator" acting as the Context component.

  Using the Strategy design pattern allowed us to create a family of generator classes that offer microservice generation strategies for various technologies. Clients can specify any of the supported generator technology in MAPI-DM in order to take advantage of different generation strategies to create polyglot microservice applications. With this pattern, extending MSA-Gen to support microservice generation for new technologies in future becomes very simple. For example, one can extend MSA-Gen to support the Flask web framework by just creating a new concrete generation strategy component called "flask_generator" as a child component of "abstract_python_generator" containing the logic to generate Flask-based microservices.

- **Factory Method Design Pattern:** The Factory Method design pattern belongs to the class of creational patterns as classified by GoF. The primary intent of this

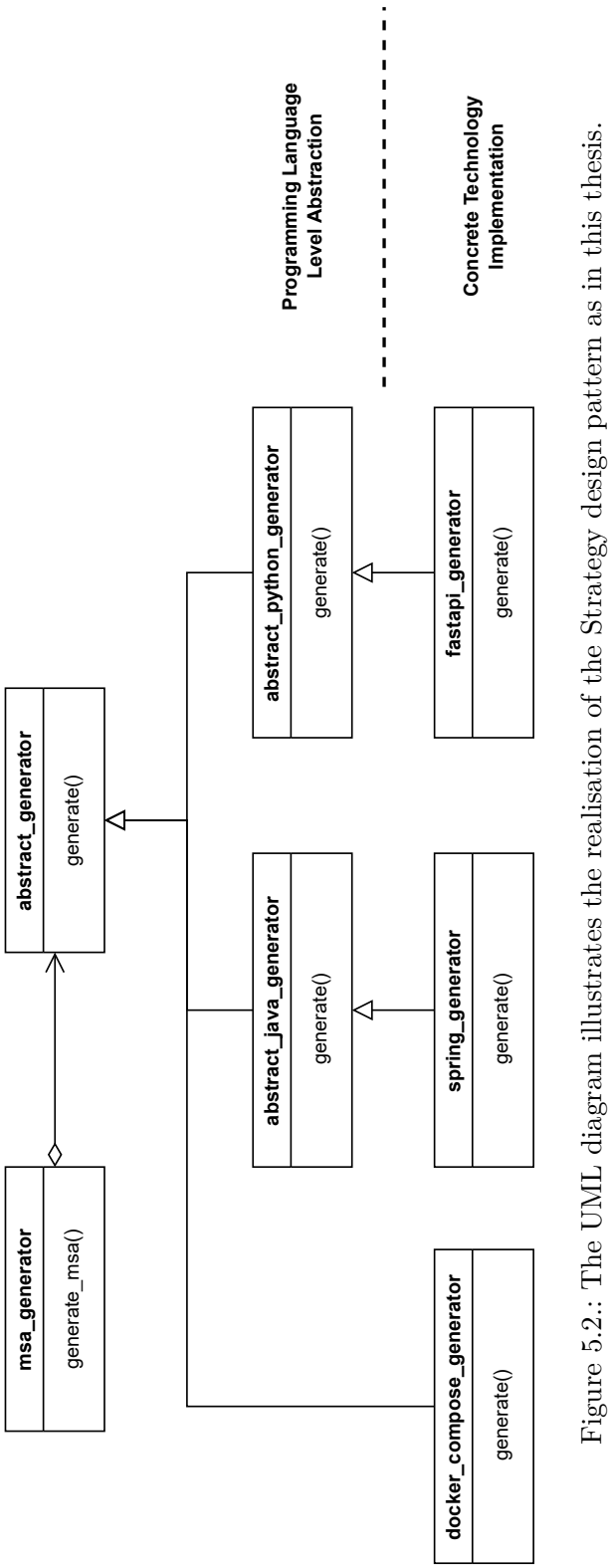

Figure 5.2.: The UML diagram illustrates the realisation of the Strategy design pattern as in this thesis.

pattern is to create a common interface for creating an object but delegate the task of object instantiation to the subclasses [Gam+94]. The participating components of this design pattern, as defined by the GoF, include the following:

- **Product:** This component is the common interface for all concrete objects created by the factory method.
- **ConcreteProduct:** This component represents concrete realisations of the common interface component Product.
- **Creator:** This component declares an abstract factory method that returns a Product object. This factory method can also have a default implementation that returns an instance of ConcreteProduct.
- **ConcreteCreator:** This component overrides the factory method defined in the Creator component to return an instance of a particular ConcreteProduct component.

Our implementation of the Factory Method pattern represented in Figure 5.3 includes the "abstract_generator", acting as the Product component, which is the common interface for all other abstract or concrete generators. In addition, we have also created an additional layer of abstraction at the programming language level represented by the components "abstract_java_generator" and "abstract_python_generator", which in turn have concrete generator components that are technology-specific implementations like "spring_generator" and "fastapi_generator" representing ConcreteProduct components. We also have a concrete generator implementation, "docker_compose_generator", for Docker Compose technology, another ConcreteProduct component. Finally, we have the "generator_factory" acting as our ConcreteCreator component.

When implementing this design pattern, we use the parameterised factory method variation described by GoF [Gam+94]. We pass a "service_data" parameter to the factory method "get_generator" in this variation. As represented in Figure 5.3, this parameter holds the relevant information for the factory method, which uses conditional logic to determine the type of generator instance to be created. Using this pattern not only allowed us to delegate the creation of technology-specific generator instances into a separate factory method but also to provide a common interface for the various concrete generators and eliminate the necessity to bind the logic of creating technology-specific generator instances in the code.

Generator Configuration

With the support of generating microservices in various programming languages using different technologies, we needed a strategy to handle their configurations efficiently in order to avoid hardcoding them. Each programming language can have language-specific configurations like, for example, the compiler/interpreter version to be used when building and running the generated microservices. Furthermore, each technology, like Spring and FastAPI, comes with numerous technology-specific configuration options

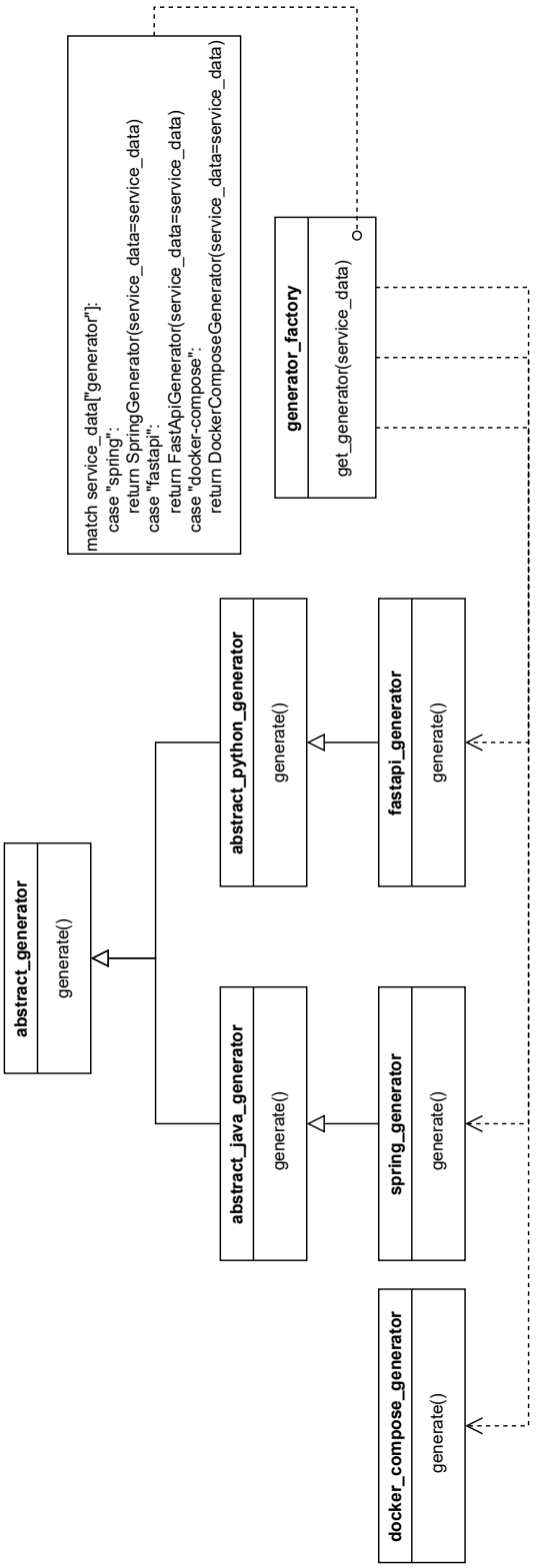

Figure 5.3.: The UML diagram illustrates the realisation of the Factory Method design pattern as in this thesis.

like version, logging, and project structure. Combining the configuration settings for diverse programming languages and technologies into a single source creates haphazard structuring of the configurations making it inefficient and leading to the anti-pattern "Big Ball of Mud" [FY97]. Our approach to avoid this hurdle was by maintaining individual configuration files for each language and technology supported by MSA-Gen. As a result, it allowed each configuration file to be compact and focused by adopting the concept of separation of concerns in the context of configuration.

In addition, we also support the option of configuration overriding in a hierarchical structure where any child configuration can always override the configurations of the parent by redefining them in its file with new values. This configuration overriding concept becomes necessary when we have multiple technologies depending on the configurations of a common programming language but simultaneously require different values to a part of the configurations based on their use case. For example, FastAPI and Flask are both technologies that use Python programming language and hence rely on its configuration. With the option to override configuration, these technologies can now override the default version of Python and use different versions of the programming language based on their individual generation strategy.

Selection of Generator Technologies

After understanding the necessity and benefits of having independent configurations for each technology, we next discuss the criteria that were considered when selecting Spring and FastAPI technologies as our starting point in this thesis.

- **Prior Experience:** We used this criterion to account for any existing experience in developing enterprise microservice applications using a given technology and programming language. Having a higher technology experience shortens the time required to get started and develop a new microservice generator for that technology which is advantageous when adopting a rapid prototyping development strategy as in this thesis.
- **Documentation and Learning Resources:** We used this criterion to evaluate the availability of proper documentation and learning resources for a given technology. Having this aids in faster learning and adoption of new technologies.
- **Supportive Tool Integrations:** As part of this criterion, we evaluated the support of different tools that we could rapidly integrate as part of a single ecosystem to realise common patterns when developing applications following microservice architectural style. For example, Spring is one of the most widely adopted frameworks for developing resilient enterprise Java-based applications. This technology offers a comprehensive ecosystem of supporting frameworks like Spring Cloud that offer tools to realise various distributed systems patterns like intelligent routing, service discovery, and configuration management.
- **Supported Features:** When evaluating this criterion, we performed a mutual comparison of the different features offered by related technologies and used the

collected information to select the technology that offered additional supporting features apart from the features relevant to satisfy the scope of this thesis. The result of such a comparison between FastAPI and Flask web frameworks for generating Python-based microservices is presented in Table 5.1. We decided to adopt FastAPI over Flask due to its advantages, like native support for both synchronous and asynchronous task executions, API documentation, and data validation and serialisation.

- **Future use cases:** Using this criterion, we evaluated technologies to understand their compatibility and support for future enhancements that can possibly be an extension of this thesis's work. For example, technologies that support asynchronous modes of inter-service communication and task executions in addition to the synchronous mode.

Upgrade of Third Party Libraries in Microservice Templates

All generated microservices depend on one or more third-party libraries during their runtime or compile time. It becomes necessary to upgrade these dependencies during new releases, which requires incrementing the dependency artefact's version number in the generator configuration. After exploring a few alternatives for performing an automatic upgrade of dependencies and testing them out, we decided to keep this as a manual approach for the following reasons:

- New releases of third-party dependencies can introduce breaking changes compared to their previous versions. As a consequence, it would require further changes to the existing microservice templates in order to function correctly. We could not identify any straightforward solution to automatically make these changes in the templates for different technologies.

- Alternative search and upgrade solutions were necessary when an automatic dependency upgrade mechanism was unavailable for a particular technology. Developing a service that offers a generic solution to this problem was beyond the scope of this thesis.

5.2. Microservice Application Composition and Observability

Developing enterprise-scale microservice applications with a sound architectural design is no trivial task, and MSA is not a silver bullet to everything. This architectural style also has its own issues, which must be addressed when designing an enterprise application. For example, due to the distributed nature of MSAs, reliable inter-service communication between microservices can be challenging. Furthermore, the architecture can be dynamic and constantly changing in such a distributed system. New microservices can be spawned up at any instant, or existing microservices can crash or get deleted. In

Criteria	Web Framework	
	FastAPI	**Flask**
Open Source	Yes	Yes
Gateway Interface Specification Type	Asynchronous Server Gateway Interface (ASGI) specifications	Web Server Gateway Interface (WSGI) specifications
Synchronous Task Executions	Yes	Yes
Asynchronous Task Executions	Yes	No
Native API Documentation Support	OpenAPI and ReDoc	No
Native Data Validation Support	Yes	No
Native Data Serialization Support	Yes	No
GitHub Stars (09.04.2023)	56.5k	62.5k
Community Support	Relatively small	Rich community support

Table 5.1.: Comparison between FastAPI and Flask web frameworks for developing Python-based microservices.

such a dynamic environment, it becomes necessary that each microservice be discoverable to other microservices within the network for successful inter-service communication to occur. To address such issues in the composed MSA, we adopted several MSA design patterns. These design patterns are part of the more extensive collection of MSA design patterns that together form the microservice architecture pattern language [Ric19].

5.2.1. Communication Patterns

In this section, we discuss the set of MSA design patterns related to microservices' communicational aspects. We first describe the mechanism that we adopted for realising inter-service communications between microservices within the MSA network. Next, we explain the design pattern adopted to streamline external communications between clients and microservices. Finally, we discuss a design pattern that facilitates the discovery of microservices within the MSA network for enabling a resilient form of inter-service communications.

- **Synchronous RPI using REST**: In Section 2.1, we learnt about the various protocols that microservices use for inter-service communication. In this thesis, we implement the inter-service communication between microservices using synchronous RPI pattern using REST for all microservices generated by MSA-Gen [Ric19]. This communication form follows the request/response style, where clients send requests to the services via the RESTful APIs exposed by the services to manipulate the state of resource objects. The service then processes these requests and sends back a response to the clients.

- **API Gateway Pattern**: In contrast to inter-service communication, external communication between clients and microservices also exists. In enterprise MSAs, it is challenging for clients to keep track of the microservices they must communicate with to realise specific functional requirements. Tracking and composing this information is undesirable as it fails to encapsulate and hide the application's decomposition from the clients. It creates a tight coupling between clients and the application's internal architecture, which can negatively affect future architectural changes. In order to overcome such issues, we adopted the API gateway pattern.

  In this thesis, we use the Spring Cloud Gateway [2] project offering an API Gateway service. This service delivers a unified interface for clients to communicate with the application's microservices as depicted in Figure 5.4. It is primarily responsible for API compositions and routing client requests to the appropriate microservice. In addition, API Gateway can also handle cross-cutting concerns like request rate limiting and implementing central security measures for every external request made to the MSA network.

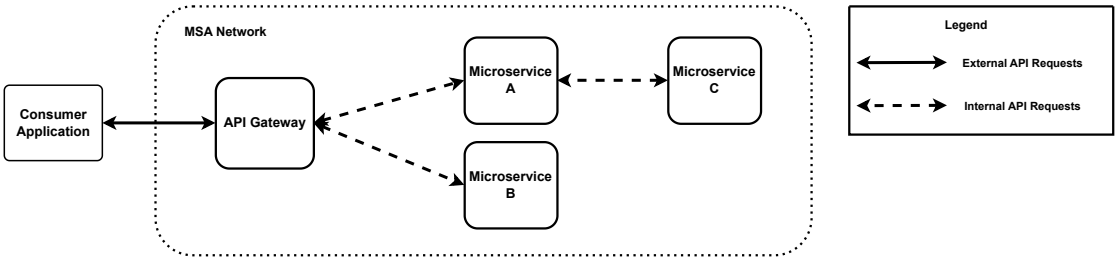

Figure 5.4.: API-Gateway design pattern offering clients a unified interface to communicate with application's microservices.

- **Service Discovery using Self-Registration Pattern**: Another issue in enterprise MSAs is that they can contain hundreds of microservices, each running multiple instances. These can change dynamically at any point in time whenever there are failures, upgrades or autoscaling. In such scenarios, to carry out inter-service communication, it becomes necessary for microservices to be able to discover other microservices within the MSA network. To address this problem, we

[2] https://cloud.spring.io/spring-cloud-gateway/reference/html/ (accessed on 18.05.2023)

adopt the Service Discovery pattern. In this pattern, we deploy a Service Discovery microservice that is responsible for maintaining a service registry containing the network addresses of all the registered and running microservices within the MSA network. There are several variations for realising this pattern [Ric19]. In this thesis, we adopted the self-registration approach, as illustrated in Figure 5.5.

We use the Spring Cloud Netflix [3] project that offers Eureka Service Discovery composed of server and client for realising the service discovery pattern. The Eureka server acts as our service registry, maintaining the network addresses of all the registered running microservices within the MSA. Eureka Service Discovery adopts the self-registration approach. In this variation, every microservice registers itself with the Eureka server microservice upon startup, making it discoverable to other microservices within the MSA network and deregisters itself during shutdown to convey its unavailability. In order to achieve this, we integrate the technology-specific Eureka client in each generated microservice using simple mechanisms like annotations. The client also has a bundled load balancer that performs load balancing in a basic round-robin manner. Once the microservices are registered, during inter-service communication, the caller microservice obtains the network address of the callee microservice by querying the service registry maintained by the Eureka server microservice.

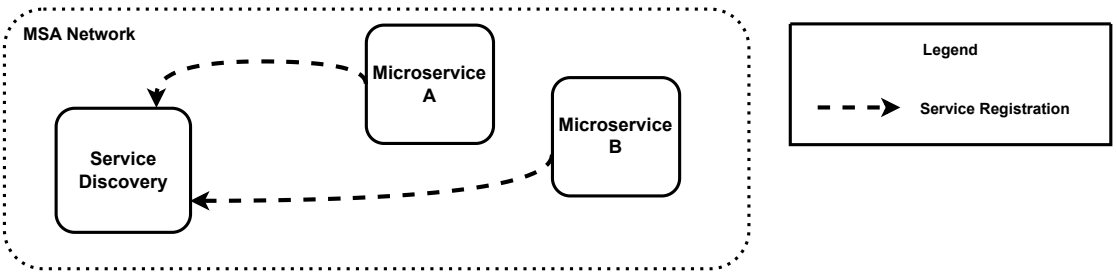

Figure 5.5.: Self-registration service discovery design pattern allows services to register with the service registry and be discoverable in MSAs.

5.2.2. Handling Cross-Cutting Concerns using Service Template Pattern

When developing enterprise microservice applications consisting of numerous microservices, a single technology may be used to develop a set of microservices. In such cases, creating production-ready microservice templates that handle common project aspects can be efficient and beneficial, which can be used to set up new microservices quickly. Drawing inspiration from this pattern, we created microservice project templates for different technologies that a templating engine can use to render microservices using MAPI-DM data. These templates address common project attributes like structure,

[3] https://cloud.spring.io/spring-cloud-netflix/reference/html/ (accessed on 18.05.2023)

build management, server stubs and other cross-cutting concerns as illustrated in Figure 5.6.

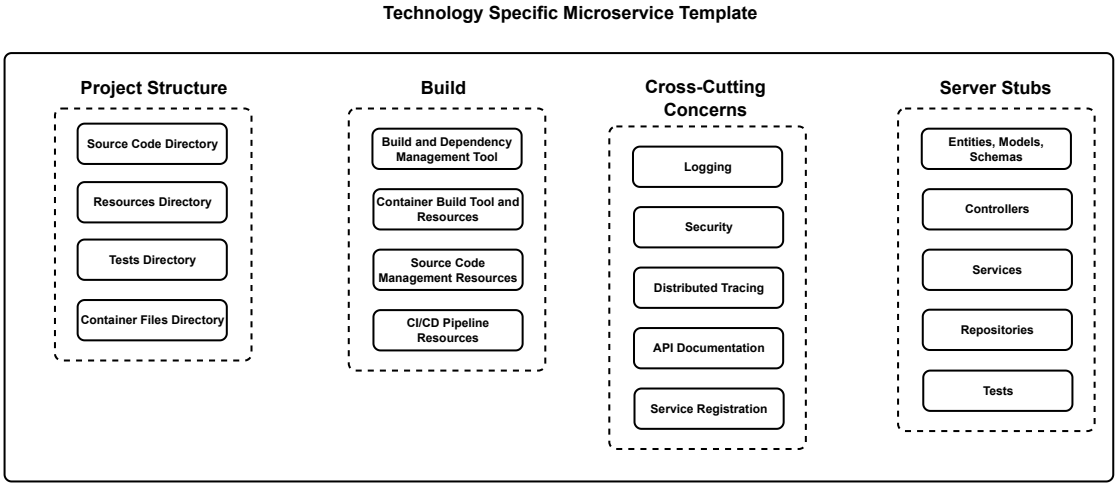

Figure 5.6.: Representative outline of a microservice template with the associated project attributes captured by it.

5.2.3. Deployment Pattern using Service Instance per Container

In this thesis, we adopted the "Service Instance per Container" deployment pattern to compose and deploy the generated microservices. As illustrated in Figure 5.7, during the build process, for every generated microservices, we create a container image. Container images are lightweight software units that package source code and associated dependencies like system libraries, third-party libraries and any other dependencies necessary to run a software application. When deployed into a container runtime engine like Docker, the container images become containers at runtime. These containers offer an isolated execution environment for software applications. As a consequence, these software applications can be executed consistently, quickly and reliably in various computing environments.

In this thesis, we have chosen Docker [4], particularly Docker Engine and Docker Compose, as our containerisation and composition tool as it is open source, widely adopted in the industry, and has extensive community support and documentation. Docker software is used to deliver software in packages known as containers using OS-level virtualisation techniques. These containers are hosted and run in the Docker Engine. This engine is an open-source containerisation technology. It is used to build docker container images and containerise software applications. We use this technology to containerise individual microservices and compose them all to create and deploy enterprise-scale microservice applications.

[4]https://www.docker.com/ (accessed on 18.05.2023)

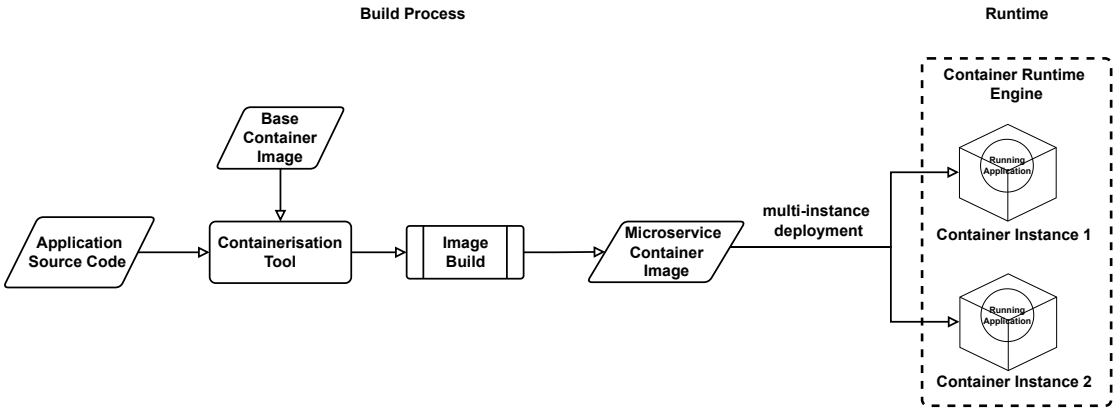

Figure 5.7.: Service instance per container design pattern to have isolated service deployment.

5.2.4. MSA Observability

After deploying enterprise microservice applications into production, observing and understanding what is happening in the system is usually challenging due to its distributed nature. Capturing data on microservices like resource consumption, network traffic, and service failures allows development teams to proactively improve their performance and identify issues. Consequently, to generate such observable microservices, in this section, we discuss the subset of MSA design patterns that we adopted from the ones proposed by Richardson in [Ric19] and Newman in [New21].

To realise these design patterns most efficiently, in this thesis, we integrated an MSA Observability tool that uses a telemetry collector to collect various telemetry data such as logs, traces and metrics from each deployed microservice. Every generated microservice is instrumented with a technology-specific telemetry agent that collects telemetry data of the microservice and transfers it to the telemetry collector as illustrated in Figure 5.8. The service consumer can then visualise this data via the UI offered by the MSA Observability tool. Next, we present the set of MSA Observability design patterns we adopted and then discuss the Observability framework and tool used in this thesis.

- **Log Aggregation:** This pattern uses a centralised logging service to aggregate the logs generated by every microservice into a single data store. These logs can be useful for analysing application errors, warnings, information, and debug messages.
- **Application Metrics:** The application metrics pattern allows us to collect microservice metrics that provide critical data concerning the health of running microservices. The metrics can include application-level metrics like request rate, request failures and latency or infrastructure-level metrics such as CPU and memory resource consumption data. After collection, the metrics are exported to a centralised metric collection service, where they can be visualised and monitored by service consumers or by dedicated system administrators.

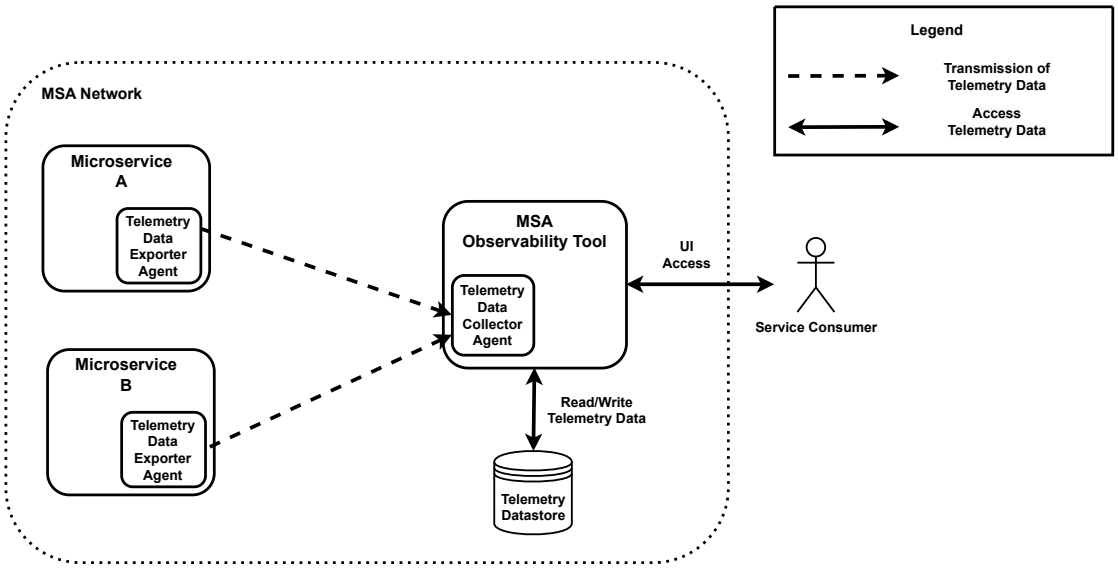

Figure 5.8.: Telemetry data collection from deployed microservices using an MSA Observability tool to comprehend their run time behaviour.

- **Distributed Tracing:** In distributed systems like microservice applications, a single request originating from the client generally traverses multiple microservices before returning a result. Tracing such requests and identifying the point of failure can be challenging. For this reason, we have adopted the distributed tracing pattern. In this pattern, we instrumented every microservice with an instrumentation agent. This agent assigns each external request called trace, a unique identifier called a trace identifier. For each operation performed when processing this request, called a span, gets a span identifier [Ric19]. We then injected these identifiers into the microservice's log entries. This data was then utilised to trace paths of external requests across multiple microservices and determine the point of failures in MSAs.

- **Exception Tracking:** Software applications are prone to a wide range of exceptions, and microservices are no different. In certain application domains, exceptions may require quick handling and alerting. For this reason, we have adopted the exception tracking pattern. In this pattern, an exception tracking service accumulates the exceptions that occurred during the microservices' runtime. This data can then be used to generate alerts for specific critical exceptions or whenever a high number of exceptions occur.

- **Health Check API:** In MSAs, microservices can sometimes be up and running but cannot handle any requests. The microservice is unhealthy in such situations as it cannot service any requests. In order to identify the health status of a microservice, we adopted the health check API pattern. In this design pattern,

each microservice exposes a health check API endpoint that provides the health status of the microservice. Supporting services like Service Discovery can query this endpoint at regular intervals to determine the health condition of the registered microservices and detect and remove any unhealthy services from the service registry.

OpenTelemetry Framework

OpenTelemetry [5] is a framework which is composed of APIs, tools and *Software Development Kits (SDKs)*. It offers mechanisms to instrument application code, making the systems more observable. This thesis uses OpenTelemetry as the observability framework because it is open-source and vendor-neutral. It is backed by several industry leaders in the observability space. Furthermore, as a *Cloud Native Computing Foundation (CNCF)* incubating project, it offers cloud-native support, which is extremely important in developing scalable enterprise-scale MSAs. In addition, it offers effortless integration of several technology-specific telemetry instrumentation agents. In this thesis, we instrumented a telemetry instrumentation agent to each generated microservice as part of its code. These agents collect various telemetry data like metrics, traces and logs of the instrumented microservices and export them over to an OpenTelemetry collector service via *OpenTelemetry Protocol (OTLP)* to HTTP or gRPC based endpoints.

SigNoz Application Performance Monitoring (APM) Tool

While OpenTelemetry provided the backbone for MSA observability, we needed a way to visualise and monitor the collected telemetry data, for which we integrated the SigNoz [6] tool. Before finalizing on using SigNoz, we made a mutual comparison with many other popular open-source observability tools based on various criteria as captured in Table 5.2.

SigNoz is an open-source application performance monitoring tool that uses distributed tracing to observe, monitor and troubleshoot any issues of software applications with OpenTelemetry instrumentation. This tool bundles an OpenTelemetry collector that collects telemetry data exported by OpenTelemetry agents instrumented in microservices. It then aggregates this telemetry data and offers various dashboards to visualise them and gain valuable insights into the application's behaviour. In this thesis, we use it to monitor application metrics, aggregate container logs, obtain the microservice application's architectural overview, trace external requests across microservices and also for exception tracking.

[5] https://opentelemetry.io/ (accessed on 18.05.2023)
[6] https://signoz.io/ (accessed on 18.05.2023)

Criteria	APM Tools					
	SigNoz	**Apache SkyWalking**[a]	**Elastic APM Server**[b]	**Zipkin**[c]	**Jaeger**[d]	**Prometheus**[e]
Installation Process	Simple	Simple	Complex	Simple	Simple	Simple
Native OTLP Support	Yes	No	Yes	No	Partial	No
Metrics	Yes	Yes	Yes	No	No	Yes
Logs	Yes	Yes req intg	Yes req intg	No	No	No
Dashboards	Yes	Yes	Yes req intg	No	No	Yes
Distributed Tracing	Yes	Yes	Yes	Yes	Yes	No
Service Map	Yes	Yes	Yes	Yes	Yes	No
GitHub Stars (09.04.2023)	12.4k	21.6k	1.1k	16.1	17.4k	47.5k

Table 5.2.: Comparison between different open source APM tools. The superscript "req intg" indicates that further integration with other tools is necessary to support the criterion in that tool.

[a] https://skywalking.apache.org/ (accessed on 18.05.2023)
[b] https://www.elastic.co/guide/en/apm/guide/8.7/apm-overview.html (accessed on 18.05.2023)
[c] https://zipkin.io/ (accessed on 18.05.2023)
[d] https://www.jaegertracing.io/ (accessed on 18.05.2023)
[e] https://prometheus.io/ (accessed on 18.05.2023)

6. Implementation

> Measuring programming progress by lines of code is like measuring aircraft building progress by weight.
>
> Bill Gates

Contents

This chapter discusses the technologies we used to develop MAPI-DM and the MSA-Gen web application.

6.1. Data and Configurations

This section discusses the various data and configuration representation formats and languages used in developing MAPI-DM and the MSA-Gen web application. For each representation format, we first give a brief description. After that, we discuss where we used it in this thesis and outline the reasons for choosing the formats.

JavaScript Object Notation (JSON)

In this thesis, for representing the data of MAPI-DM and also for all exchange of the payloads between the microservices, we use *JavaScript Object Notation (JSON)* [1], a lightweight data-interchange format. An example of MAPI-DM represented in JSON format is illustrated by Figure 8.1. We decided to use JSON because it is a language and framework-independent data representation format. Its self-describing and simple hierarchical data representation structure make it human-friendly to read and write. In addition, due to its compactness and simple textual representation, simple editors can be used to view and edit it easily. Furthermore, as its structure follows specific guidelines outlined by different standards, machines can efficiently and quickly parse and generate JSON files. It is also one of the most widely adopted data exchange formats for developing RESTful APIs over HTTP.

[1] https://www.json.org/json-en.html (accessed on 18.05.2023)

YAML Ain't Markup Language (YAML)

YAML [2] is a programming language agnostic data serialisation language. In this thesis, YAML is used to capture the configurations of the different microservice generators of the MSA-Gen web application. We used YAML for this purpose because its human-friendly design makes it easy to read and understand. It is also one of the most popular ways to represent the configuration files of applications. Figure 6.1 shows a snippet of the configuration file of the Spring microservice generator.

```yaml
spring:
  dependencies:
    springBootStarterParentVersion: "3.0.3"
    springCloudDependenciesVersion: "2022.0.1"
    springdocOpenapiStarterWebmvcUiVersion: "2.0.2"
    springCloudStarterOpenFeignVersion: "4.0.1"
    opentelemetryJavaInstrumentationJarVersion: "1.22.1"
  logging:
    level: "INFO"
    file:
      name: "application.log"
    logback:
      rollingPolicy:
        fileNamePattern: "application-%d{yyyy-MM-dd}.%i.gz"
        # The maximum size of log file before it is archived.
        maxFileSizeInMegaBytes: "10"
        cleanHistoryOnStart: "false"
        # The maximum amount of size log archives can take before being deleted.
        totalSizeCapInMegaBytes: "50"
        # The maximum number of archive log files to keep.
        maxHistoryInDays: "7"
```

Figure 6.1.: A snippet of the Spring microservice generator configuration written in YAML.

6.2. Frameworks

In this section, we discuss the web framework that we used to develop the MSA-Gen application. We first briefly describe the framework and then discuss the different framework features we leveraged in developing the MSA-Gen web application.

FastAPI

The core of the MSA-Gen web application is developed using FastAPI [3]. FastAPI is a high-performance web framework used to develop web applications in Python programming language. Learning it and developing lightweight, fast and robust production-ready

[2] https://yaml.org/ (accessed on 18.05.2023)
[3] https://fastapi.tiangolo.com/lo/ (accessed on 18.05.2023)

web applications is easy. In Table 5.1 presented in the previous chapter, we outlined the various advantages of this framework by comparing it to another popular Python-based web framework called Flask [4].

In this thesis, using FastAPI allowed us to document our microservice generation API endpoint using the built-in support for OpenAPI documentation. In addition, the built-in integration of the Pydantic [5] Python library as part of the framework allowed us to realise data parsing and validation effortlessly. We used Pydantic to define schema models for each component of the MAPI-DM metamodel containing the attributes defined for that particular component as shown in Figure 4.7. These schema models were also used to parse the MAPI-DM data represented in JSON format and also to perform validation checks during the parsing process to prevent the input of invalid MAPI-DM data.

6.3. Tools

This section discusses the various tools we used as part of the MSA-Gen application for its development and distribution. We first discuss the templating engine adopted in this thesis to create the technology-specific microservice templates used for microservice generation. After that, we briefly discuss the tool used to containerise the application for easier distribution.

Jinja Templating Engine

Jinja [6] is a widely used open-source templating engine for Python programming language, the underlying programming language used for developing the MSA-Gen web application. The advantages of using Jinja are that it is fast, expressive and extensible. It allows the creation of template files that carry special placeholders like "{% ... %}" and "{{ ... }}", as illustrated in Figure 6.2. The templating engine accepts rendering data as input that the engine will use to replace the special placeholders present in the template files to render final concrete files with the necessary data.

This thesis uses Jinja templating engine to write various template files for microservices. These template files differed based on the underlying technology used to develop the microservices. We use MAPI-DM as our input rendering data, which Jinja uses together with the technology-specific microservice templates to generate microservices for different technologies.

Furthermore, the extensible part of Jinja allowed us to write custom filters that we could later use in the microservice template files. We implemented filters for code case conversion as different programming languages specify different styles for writing code. For example, when writing code using Java programming language, the best practice is to use camel case style, while the best practice for writing Python language code is to use

[4] https://flask.palletsprojects.com/en/2.3.x/ (accessed on 18.05.2023)
[5] https://docs.pydantic.dev/latest/ (accessed on 18.05.2023)
[6] https://jinja.palletsprojects.com/en/3.1.x/ (accessed on 18.05.2023)

```
@Slf4j
@Service
public class {{ api.apiName|pascal_case }}Service {

    private String serviceName = "{{ serviceName }}";
    private String apiName = "{{ api.apiName }}";

    {% for dependency_feign_client in api.uniqueDependencyApiFeignClients %}
    @Autowired
    private {{ dependency_feign_client|pascal_case }}Client {{ dependency_feign_client|camel_case }}Client;
    {% endfor %}

    {% for internal_dependency in api.uniqueInternalDependencies %}
    @Autowired
    private {{ internal_dependency|pascal_case }}Service {{ internal_dependency|camel_case }}Service;
    {% endfor %}

    {% for path in api.paths %}
    public {% if path.response.dataType == "list" %}List<{% if path.response.userDefinedType is defined and path.response.elementType == "user-defined-type"
        log.info("Processing request in service {} for resource path: {} in the API endpoint: {}.", serviceName, "{{ path.path }}", apiName);

        {% for dependency in path.dependencies %}
        {% if dependency.isInternal is defined and not dependency.isInternal %}
        log.info("Making request to the dependent service {} for the resource path: {} in the API endpoint: {}.", "{{ dependency.serviceName }}", "{{ depende
        {% else %}
        log.info("Making an internal request to the resource path: {} in the API endpoint: {}.", "{{ dependency.path }}", "{{ dependency.apiName }}");
        {% endif %}
        {% if dependency.dependencyData.response.dataType == "list" %}        {% if dependency.dependencyData.response.userDefinedType is defined and depende
```

Figure 6.2.: A snippet of microservice template file written in Jinja.

snake case style. In addition, we also implemented filters that allowed us to convert the generic data type values specified in MAPI-DM into programming language-specific data types. This conversion was necessary because the keywords used to represent data types differ for different programming languages. For example, in Java, the keyword “String” is used to define a variable of string data type. In contrast, in Python, we use the keyword “string”, all in lowercase characters, to define a string variable. To conclude, Jinja templating engine offered many possibilities for extensions wherever necessary, allowing us to design complex and robust templates effortlessly.

Docker

We used Docker in MSA-Gen to containerise the application so that it can be run in an isolated and self-sufficient container environment and easily deployed on any operating system with support for running Docker. We have already discussed Docker and its advantages that we benefit from in Section 5.2.3.

7. Evaluation Approach

> Any program is only as good as it is useful.
>
> Linus Torvalds

Contents

This chapter first outlines our evaluation case study and strategy. Next, we list the different evaluation tasks and discuss their purpose.

7.1. Evaluation Strategy

Enterprise MSAs are generally composed of hundreds of microservices that interact with each other to form a complex system. In order to obtain a most practical understanding of the advantages and drawbacks when using our implemented approach to mock such MSAs, we designed a hypothetical E-commerce application for our evaluation case study following the exemplary MSA pattern where we broke down the entire application into a suite of autonomous services [Ric19].

7.1.1. Case Study E-commerce Microservice Application

The E-commerce case study application was partially inspired by the open-source project "e-commerce-microservices-sample" consisting of just five microservices [Rav]. We partially adopted this design and extended the case study architecture by introducing additional microservices to closely resemble a large-scale MSA. Our runtime MSA consisted of thirty microservices in total, of which nineteen were E-commerce application microservices captured as part of MAPI-DM, two other microservices included the API-Gateway Service and the Service Discovery, which are generated out of the box by MSA-Gen during the microservice generation process. Finally, the remaining nine microservices belonged to the SigNoz APM tool that was deployed for facilitating MSA observability [Ric19].

Next, we present the list of E-commerce microservices we defined, describe their purpose, and list the different API endpoints they expose.

- **Advertisement Service:** This microservice offers functionality for managing advertisements for the E-commerce microservice application.
  - **advertisement:** API endpoint to create, retrieve, update and delete advertisements.
- **Authentication Service:** This microservice provides all authentication-based functionality for users and merchants.
  - **user-authetication:** API endpoint to provide authentication for user login and logout.
  - **merchant-authetication:** API endpoint to provide authentication for merchant login and logout.
- **Blog Service:** This microservice offers functionality to manage the information published to the blogs of the E-commerce microservice application.
  - **blog-post:** API endpoint to create, retrieve, update and delete blog posts.
- **Career Service:** This microservice handles all career-related aspects like open positions management of the E-commerce microservice application.
  - **job:** API endpoint to create, retrieve, update and delete job posts.
- **Catalogue Service:** This microservice manages the catalogue of items sold by the E-commerce microservice application.
  - **item:** API endpoint to create, retrieve, update and delete items from the catalogue.
- **Customer Support Service:** This microservice offers the functionality for customers to raise their concerns with the customer support team of the E-commerce microservice application.
  - **support:** API endpoint to create, retrieve, update and delete customer support tickets for any issues.
  - **feedback:** API endpoint to add, retrieve and delete feedback provided by customers to the support they received from the customer support team.
- **Inventory Service:** This microservice manages the inventory data for each item in the catalogue of the E-commerce microservice application.
  - **item-stock:** API endpoint to create, retrieve, update and delete item stock value for items in the catalogue.

- **Loyalty Program Service:** This microservice allows users to register and participate in the loyalty bonus programmes offered by the E-commerce microservice application.
  - **loyalty-program:** API endpoint to register, retrieve, update and deregister users from the loyalty bonus programmes.
- **Merchant Service:** This microservice allows the onboarding of new merchants to the E-commerce platform where they can sell their products.
  - **merchant:** API endpoint to register, retrieve, update and deregister merchants from the E-commerce microservice application.
- **Notification Service:** This microservice subscribes, creates and forwards all notifications for the E-commerce microservice application.
  - **subscription:** API endpoint to subscribe, retrieve and unsubscribe to different notifications.
  - **notification:** API endpoint to provide the functionality to send notifications to specific users or broadcast them to all registered users.
- **Order Service:** This microservice is responsible for processing orders placed by the users of the E-commerce microservice application.
  - **order:** API endpoint to create, retrieve, update and cancel orders placed by users.
- **Payment Service:** This microservice handles all the payment functionality when users proceed to pay for their orders on the E-commerce microservice application.
  - **payment:** API endpoint to create, retrieve, update and cancel user payments.
- **Policy Service:** This microservice manages all the user-relevant policies-related data outlined by the E-commerce microservice application.
  - **policy:** API endpoint to create, retrieve, update and delete policies.
- **Review Service:** This microservice provides functionality to manage user reviews for the list of items in the catalogue.
  - **review:** API endpoint to create, retrieve, update and delete user reviews only for the products they have purchased.
- **Search Service:** This microservice offers search-related functionality to users.
  - **search:** API endpoint to search for items in the catalogue.
- **Session Service:** This microservice manages session data for all the currently logged-in users and merchants interacting with the E-commerce microservice application

  - **user-session:** API endpoint to create, retrieve, and delete session data for logged-in users.
  - **merchant-session:** API endpoint to create, retrieve, and delete session data for logged-in merchants.

- **Shipping Service:** This microservice provides functionality for managing order shipping and tracking.
  - **shipping:** API endpoint to create, retrieve, update and cancel order shipping for specific users.
  - **tracking:** API endpoint to retrieve shipping statuses of placed orders for specific users.

- **Shopping Cart Service:** This microservice offers the shopping cart and wishlist functionalities to the users of the E-commerce microservice application.
  - **cart:** API endpoint to create, retrieve, update and delete items from a user's shopping cart.
  - **wishlist:** API endpoint to create, retrieve, update and delete items from a user's wishlist.

- **User Service:** This microservice allows users to sign up for the E-commerce platform to buy products online.
  - **user:** API endpoint to register, retrieve, update and deregister users from the E-commerce microservice application.

7.1.2. Modelling the Prerequisite Data for Evaluation

Until now, we have obtained an understanding of the E-commerce microservices that we will be generating and their functionality. Next, we model the prerequisite data necessary to perform the evaluation. We first define an example scenario that captures the user's interaction with the E-commerce application. Then, we extract several functional requirements based on this scenario to simulate it.

User Interaction with the E-commerce Application

The customer's interaction with the E-commerce application begins with a new registration on the platform. After registering, the customer signs up for the loyalty program to gain bonus points. After this, the customer logs in to the created account and browses the catalogue of available products under different categories and adds a few products to the wishlist for making a future purchase. The customer subsequently receives an advertisement from the E-commerce platform about an upcoming sale where products in the wishlist will be available for a steep discount.

After that, the customer adds the items from the wishlist to the shopping cart and places the order during the sale. Later, the customer realises that some of the ordered

products are unneeded. As a result, the customer creates a support ticket to enquire about the cancellation policy. Based on the input received from the support team, the customer provides feedback to them and views the cancellation policies of the E-commerce platform. The customer then cancels the existing order and places a new order for other products added to the cart. Finally, the customer writes reviews for the purchased products after their delivery and signs out of the application.

Functional Requirements of the E-commerce Application

Using this example scenario, we next derive several functional requirements as captured in Table 7.1, which involve the user's interaction with the designed E-commerce application to simulate the scenario. We capture these functional requirements on a high level and avoid any implementation-level details as the primary intent is to simulate and not to implement them.

With all the functional requirements listed to realise the example scenario, we next model each of them as a message sequence model. It is essential to understand that message sequence models can have varied representation formats, and also, the approach taken to mock the functional requirements can differ. Next, we present one such possible approach to mock the discussed functional requirements. In this approach, we use UML sequence diagrams to model message sequence models. They illustrate the inter-service API communications between microservices, from which we infer the inter-service API dependencies for our E-commerce microservices. Due to spatial limitations, we illustrate and discuss only two of them here, and we document the remaining as part of Appendix A.

Figure 7.1 depicts the API communications that occur between microservices when mocking user registration functional requirements. The "user-service", after receiving the request to create a new user, executes the business logic for the user creation and then makes an inter-service API dependency request to the "notification-service" to send the registration notification to the newly registered user. After receiving the "notification-service" response, the "user-service" returns its response to the registration request.

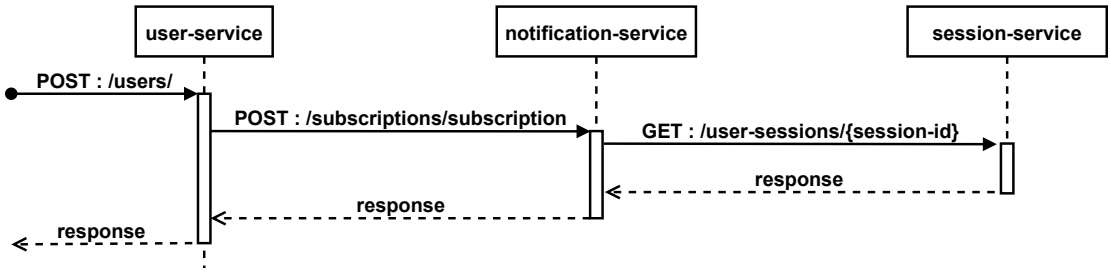

Figure 7.1.: Message sequence model illustrating registration of a new user (FR1).

Next, Figure 7.2 depicts the API communications that occur between microservices when mocking purchase order creation functional requirements. The "order-service", after receiving the request to create a new purchase order, makes an API request to the

ID	Functional Requirement (FR)
FR1	The E-commerce microservice application must allow new users to register to the platform.
FR2	The application must also allow users to register for loyalty-bonus programs offered by the application.
FR3	The application must allow users to log in to the application after a successful registration.
FR4	The application must allow users to log out of the application after a successful login.
FR5	The application must offer search functionality for users to perform item searches in the catalogue.
FR6	The application must support users in creating wishlists and adding items to their wishlists.
FR7	The application must support users in adding items either from the catalogue to their shopping cart directly or from their existing wishlists.
FR8	The application must be able to deliver generic and focused advertisement notifications to the registered users.
FR9	The application must allow users to place purchase orders for the items in their shopping cart.
FR10	The application must allow users to view consumer policies outlined by the company.
FR11	The application must allow users to cancel created purchase orders that are still in the processing stage.
FR12	The application must support users to view all the purchase orders created by them.
FR13	The application must allow users to write reviews for the items that they have purchased.
FR14	The application must offer users to create customer support tickets to raise any issues or concerns.
FR15	The application must allow users to provide feedback to the customer support they received from the support team.

Table 7.1.: The list of functional requirements derived from an example user interaction with the case study E-commerce MSA.

"session-service" to receive user session data. After receiving a response from "session-service", it makes subsequent API requests to the "catalogue-service" for fetching information on the cart items. The "catalogue-service", in turn, makes API requests to the "inventory-service" to get the current stock for those items. Following the chain of events, subsequent API requests are executed by the "order-service" to the "shipping-service", "loyalty-program-service", "payment-service", and the "notification-service" in a sequential order to fetch shipping details, loyalty-bonus data, initiate payment for purchase order and to subscribe and send notifications respectively. After receiving responses from all the dependency API requests, the "order-service" might execute any remaining business logic to process the creation of the purchasing order and return the response to the user.

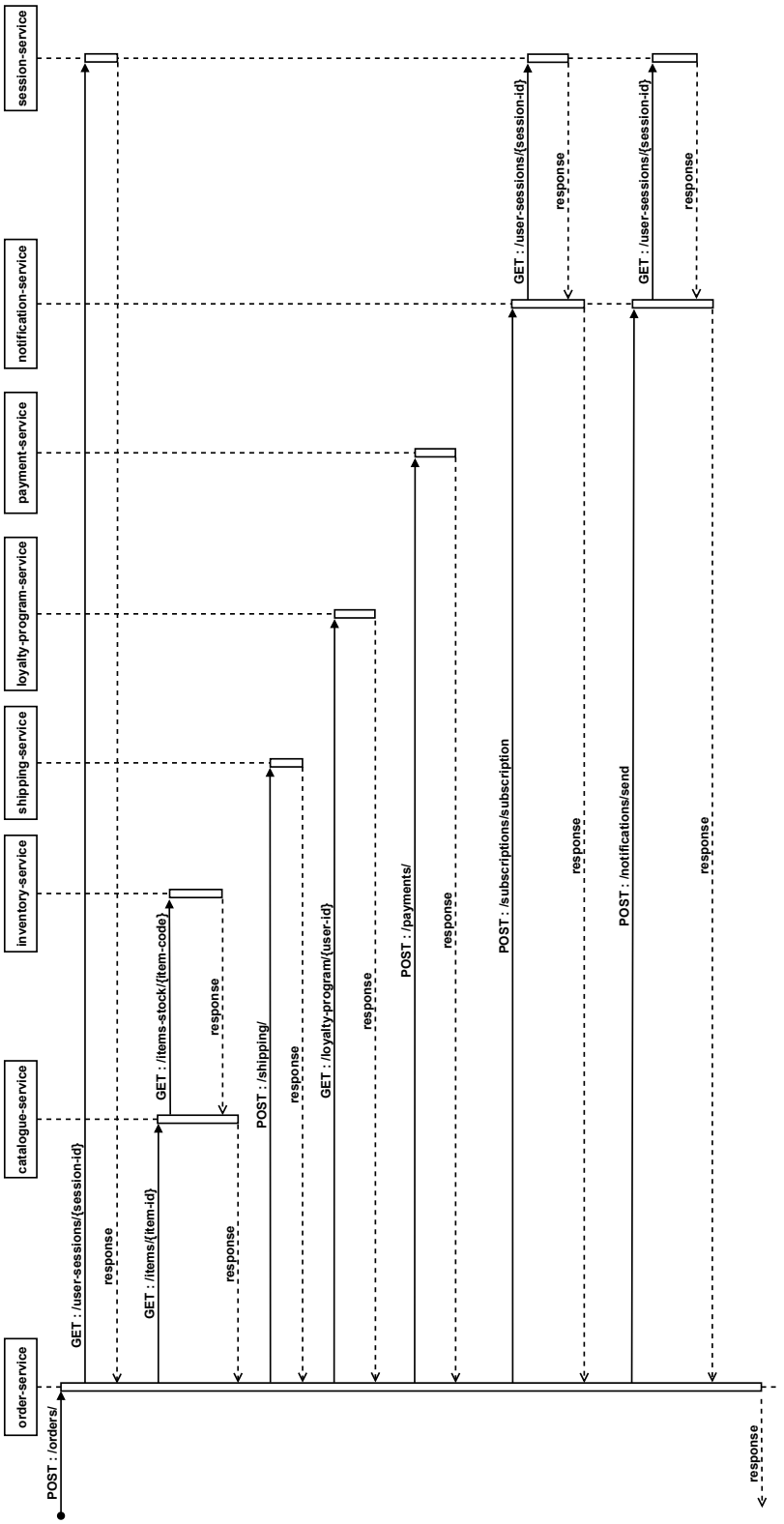

Figure 7.2.: Message sequence model illustrating purchase order creation (FR9).

Figure 7.3 represents the envisioned architectural view of the E-commerce application, illustrating the different microservices and their interactions, which we interpreted based on the modelled message sequence diagrams of all the functional requirements listed in Table 7.1. We also capture the API-Gateway service and Service Discovery and their associated interactions like service registration to correspond to our conceptual infrastructural view of a composed and deployed MSA that we had depicted in Figure 4.5.

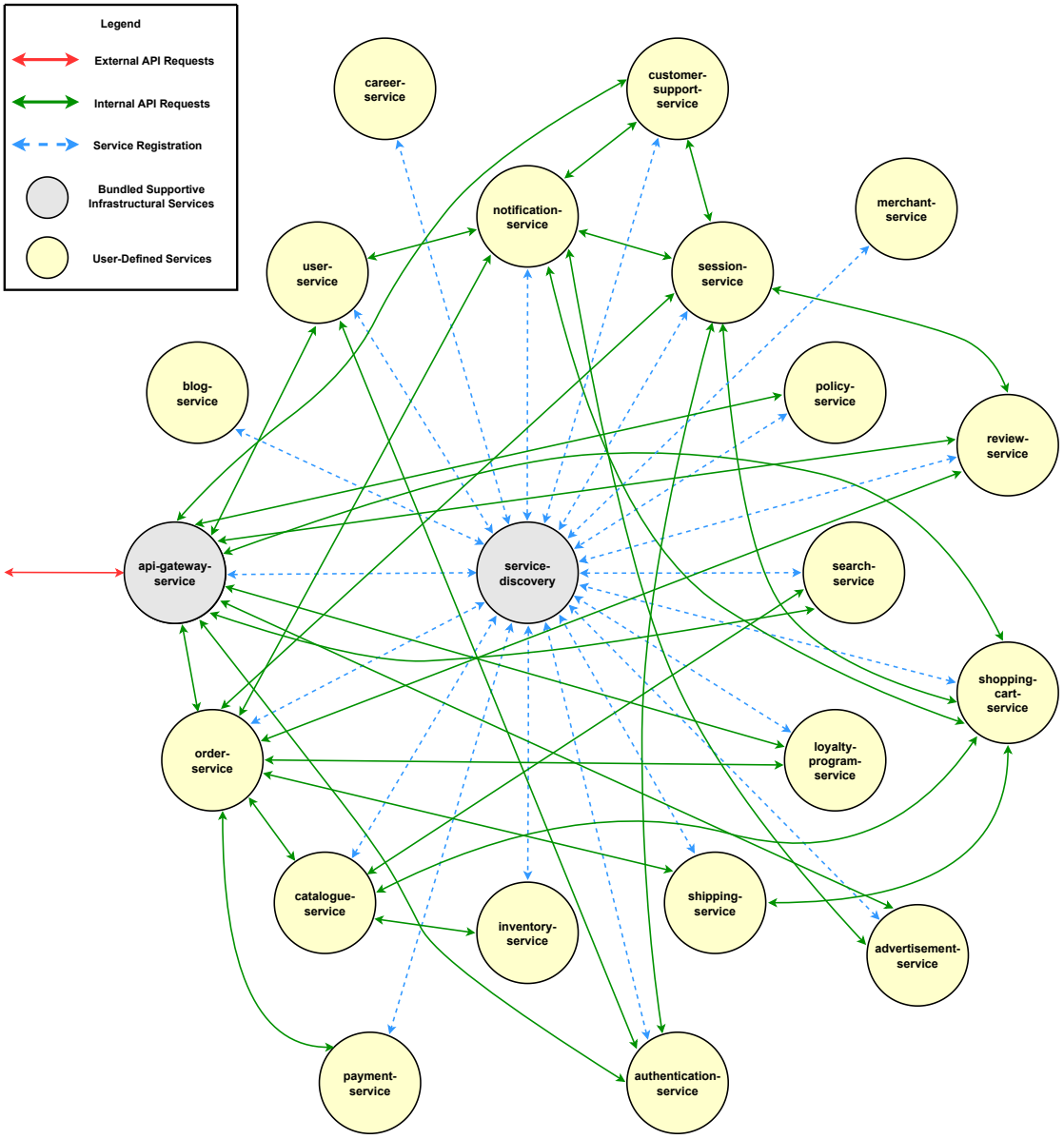

Figure 7.3.: Envisioned architectural view of the modelled E-commerce microservice application illustrating various microservice interactions.

Evaluation Steps and Tools

After modelling all the necessary prerequisite data for our evaluation and visualising the expected architectural view of our case study application, we briefly outline the evaluation procedure and list down the tools used.

- In the first step, we modelled the MAPI-DM for the E-commerce application. Based on the API endpoints exposed by each microservice that is part of the E-commerce application, we defined appropriate values for the different attributes as outlined in the MAPI-DM metamodel as defined in Section 4.4.2. In addition, for each microservice, the information on inter-service dependencies was obtained using the prerequisite data represented as message sequence models.

- As the second step to mock user interaction with the E-commerce microservice application, we created a test plan. It comprises API requests made to the application in an automated way to mock all the functional requirements captured in Table 7.1.

- In the third step, we deployed the MSA-Gen application and used the modelled MAPI-DM to generate, compose and deploy the E-commerce application onto the test system.

- The fourth and final step was executing the different evaluation tasks and capturing the results. The evaluation tasks are defined in Section 7.2

In the following section, we have documented the test setup configuration.

7.1.3. Test System Configuration

In this section, we discuss the test system configuration, where we record the hardware specification of the system, followed by the list of additional software and tools we installed and used when performing the evaluation.

Operating System and Hardware Configuration

- **Operating System:** Ubuntu 20.04 LTS version running on a WSL2 environment of Windows 10.

- **Processor:** Intel Core i5-8600 CPU @ 3.10GHz (6 CPUs)

- **RAM Capacity:** 24GB.

- **Network Speed** Download: 132.2 Mbit/s ; Upload: 25.2 Mbit/s.

Additional Software and Tools

- **Docker:** We used the Docker software to run the MSA-Gen application in a container environment, build the microservice container images, and compose and deploy the containers onto the test system for evaluation.
- **Postman[1]:** We used the Postman tool to make API requests to the MSA-Gen application for sending the MAPI-DM payload.
- **Apache JMeter[2]:** We used the Apache JMeter tool to create an API execution plan. This API execution plan allows us to make API requests to the composed E-commerce microservice application in a specified sequential order to mimic the previously defined user scenario. We configured the execution plan to simulate a hundred users concurrently making requests to the E-commerce application for five minutes.

7.2. Evaluation Tasks

This section presents the different evaluation scenarios we will execute as part of our evaluation process.

- **Task 1:** The first evaluation task involved modelling the E-commerce microservice application as MAPI-DM. From this, we wanted to evaluate the benefits, drawbacks and the total time effort required for the modelling process.
- **Task 2:** As part of the second task, we wanted to evaluate the generation and composition of the E-commerce microservice application for the following four scenarios.
  - **Scenario 1:** The first scenario comprised generating and composing the E-commerce application where all microservices use Spring technology.
  - **Scenario 2:** The second scenario constituted generating and composing the E-commerce application where all microservices use FastAPI technology.
  - **Scenario 3:** For the third scenario, we wanted to evaluate generating and composing a polyglot E-commerce application where microservices are a mix of Spring and FastAPI technologies.
  - **Scenario 4:** In the fourth scenario, we mainly wanted to experiment and understand the impact of having cyclic inter-service API dependencies in MAPI-DM definition and whether we can identify them in the mocked MSAs using the observability tool. In order to evaluate this, we made simple modifications to add a cyclic dependency to the MAPI-DM used in the first scenario.

[1]https://www.postman.com/ (accessed on 18.05.2023)
[2]https://jmeter.apache.org/ (accessed on 18.05.2023)

We wanted to evaluate the performance of the microservice generation approach and the generated microservice application for the first three scenarios based on the following Key Performance Indicators and also capture other interesting and relevant findings observed during the evaluation process. It is to be noted that when measuring the KPIs, we do not consider the auxiliary microservices that are part of the APM tool.

- **Microservices Generation Time:** This KPI captures the total microservice generation time, which includes the whole request roundtrip time from making the API request to the MSA-Gen application to processing the microservice generation.
- **Microservices Build and Deployment Time:** This KPI captures the total time required to build and deploy only the collection of microservices defined in MAPI-DM onto the container runtime engine.
- **Microservices Startup and Registration Time:** This KPI measures the total time taken for all microservices to start and register themselves to the Service Discovery.
- **Idle Microservices Resource Consumption Statistics:** This KPI captures the resource consumption statistics of the deployed microservices in their idle state for resources CPU, Memory usage and Network I/O.
- **Non-idle Microservices Resource Consumption Statistics:** This KPI captures the resource consumption statistics related to CPU, Memory usage and Network I/O of the deployed microservices when they receive continuous API requests from 100 concurrent users for five minutes.

In order to get an average estimate for the three temporal KPIs, namely "Microservices Generation Time", "Microservices Build and Deployment Time", and "Microservices Startup and Registration Time", we intended to execute three trial runs each for the scenarios one, two and three and capture the ceiled average for the temporal KPIs.

- **Task 3:** Finally, as part of the third task, we wanted to evaluate the extent of observability of the generated E-commerce microservice application on the following criteria.
  - Collection and visualisation of microservice metrics
  - Logs aggregation
  - Distributed tracing of external requests
  - Exception tracking
  - Microservices health check

8. Evaluation Outcomes

> Premature optimization is the root of all evil.
>
> Donald Knuth

Contents

After performing the evaluation approach presented in the previous chapter, we present and discuss the evaluation outcomes in this chapter.

8.1. Evaluation Results

In this section, we record in sequential order the results obtained from the execution of each evaluation task discussed in Section 7.2.

8.1.1. Task 1: Modelling MAPI-DM

As part of the first task, we modelled three variations of MAPI-DM to represent scenarios one, two and three that are part of the second evaluation task. In terms of the modelling effort, the entire modelling process took us eight hours. Due to large model size and spatial limitations, we have decided to illustrate only a portion of the overall MAPI-DM from the all Spring microservices scenario. Figure 8.1 represents one of the modelled API paths for the "advertisement-service". We have captured the complete MAPI-DM representation of scenarios one, two and three as part of the thesis's GitLab codebase [1].

In order to test scenario four, we introduced slight contrived modifications to the MAPI-DM of scenario one intending to create an artificial cyclic dependency. We

[1] https://git.rwth-aachen.de/swc-theses/master/pavan-nadkarni/msagenerator/-/releases/v1.0.0-master-thesis with evidence collection SHA: 122930e0032438b0e65b53040fd17ac455130e8c47ed (accessed on 25.05.2023)

```
{
  "services": [
    {
      "serviceName": "advertisement-service",
      "generator": "spring",
      "apis":[
        {
          "version": 1,
          "apiName": "advertisement",
          "basePath": "/advertisements",
          "paths":[
            {
              "path": "/",
              "httpMethod": "post",
              "methodName": "create-advertisement",
              "parameters":[
                {
                  "name": "advertisement-request",
                  "dataType": "user-defined-type",
                  "userDefinedType": "advertisement-request",
                  "position": "body"
                }
              ],
              "response": {
                "dataType": "user-defined-type",
                "userDefinedType": "advertisement-response"
              },
              "dependencies": [
                {
                  "serviceName": "notification-service",
                  "apiName": "notification",
                  "version": 4,
                  "path": "/broadcast",
                  "httpMethod": "post"
                }
              ]
            },
```


Figure 8.1.: A snippet of E-commerce MAPI-DM model that represents one modelled path of the “advertisement-service”.

achieved this by changing the inter-service API dependencies of “catalogue-service” and “inventory-service”.

8.1.2. Task 2: Microservice Generation Scenarios

We executed the microservice generation for the following four scenarios as part of the second task. Here, we captured the measurements of the discussed KPIs metrics for the first three scenarios to compare and assess the impact of the selected generator technology on the microservice generation process. But before we discuss our observations of each scenario, Figure 8.2 helps us visualise the service map view of the generated E-commerce microservice application in its idle state after its deployment. From this, we can ascertain that all the microservices register themselves with the Service Discovery after deployment. Furthermore, Figure 8.3 depicts the service map view of the E-commerce microservice application during a simulation run where the application processes external client requests. Next, we explain the results of different microservice generation scenarios.

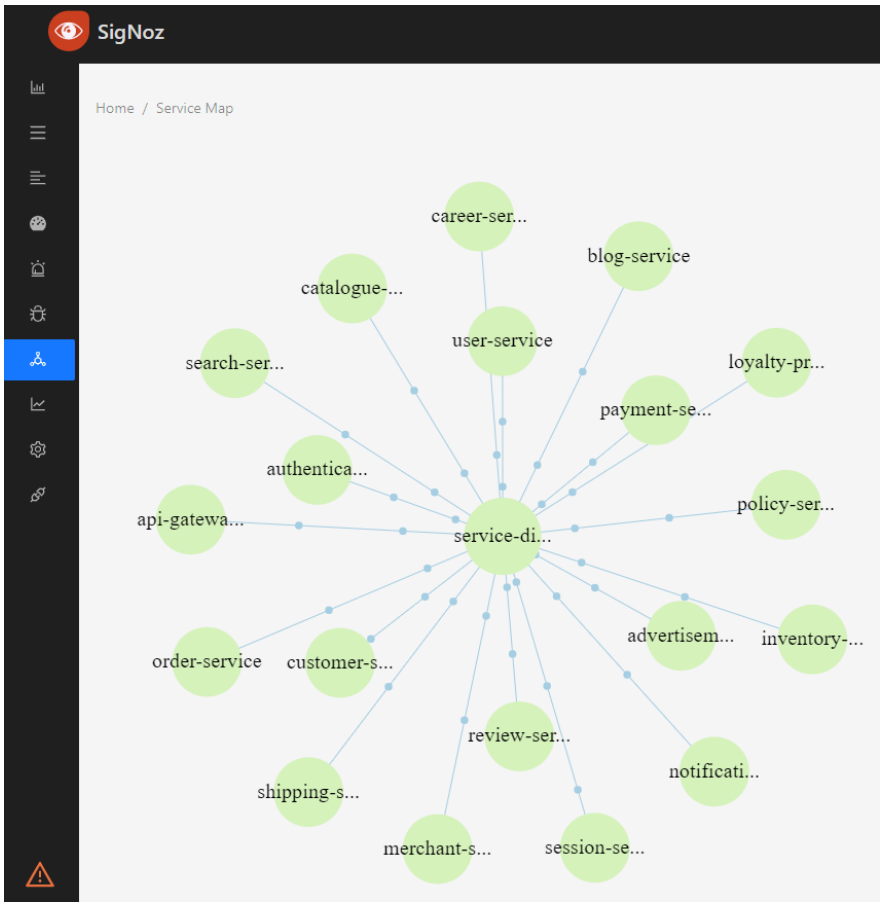

Figure 8.2.: Service map view of E-commerce application in idle state after deployment.

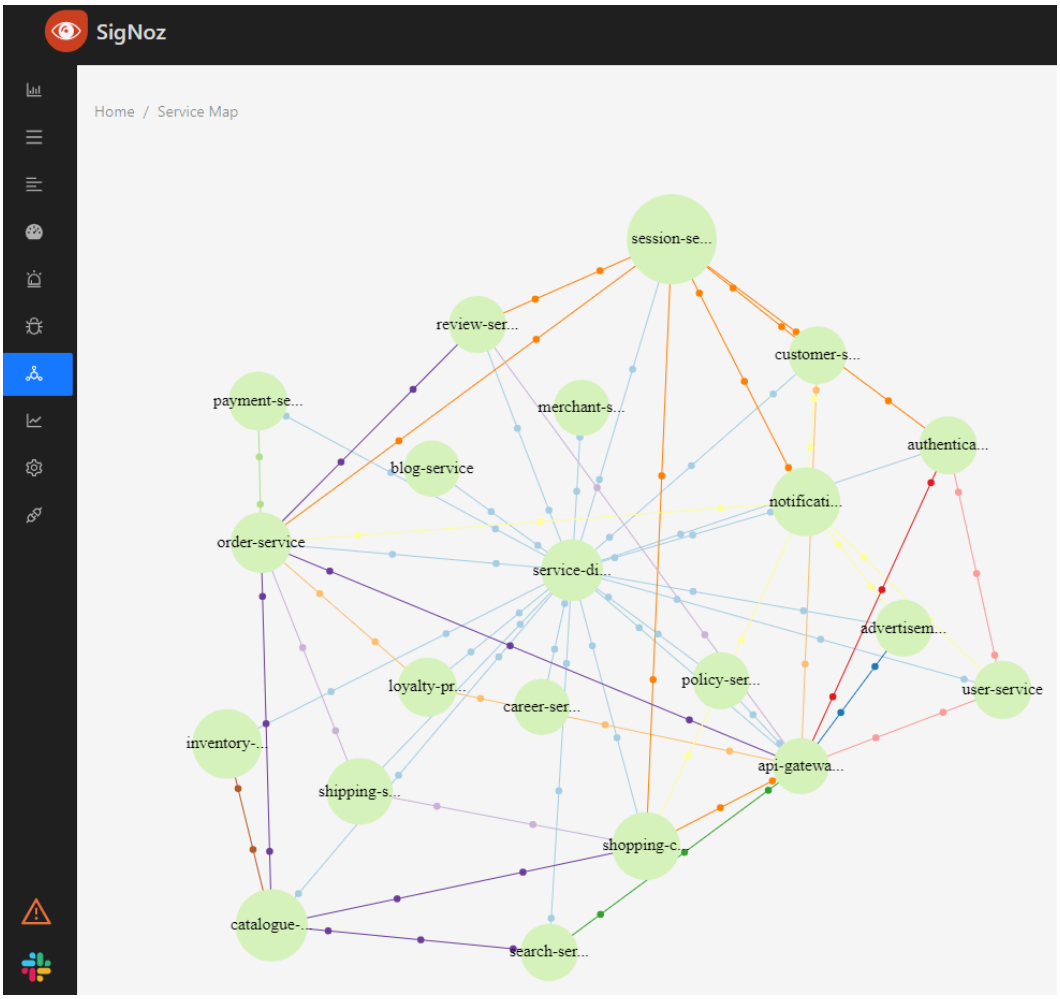

Figure 8.3.: Service map view of E-commerce application showing communication between microservices when receiving API requests.

Scenario 1: All Spring Microservices

In scenario one, we generated the E-commerce microservice application composed of all spring microservices. We recorded the temporal KPIs measurements in Table 8.1 for the three trial runs performed. For this scenario, we observed the average value for "Microservices Generation Time" to be 419ms, "Microservices Build and Deployment Time" to be 206s and "Microservices Startup and Registration Time" to be 128s. In addition, Table 8.2 represents the container resource consumption statistics of the different microservices in their idle state, and Table 8.3 depicts the container resource consumption statistics during non-idle conditions when external API requests were being executed. Comparing the two states, we see an increase in the container resource consumption of microservices in the non-idle state compared to the idle state.

Trial	KPI		
	Microservices Generation Time (ms)	**Microservices Build and Deployment Time (s)**	**Microservices Startup and Registration Time (s)**
Trial 1	510 ms	205 s	125 s
Trial 2	377 ms	214 s	130 s
Trial 3	371 ms	199 s	127 s
Ceiled Average	419 ms	206 s	128 s

Table 8.1.: Table capturing the data for temporal KPIs for Spring microservices scenario.

Microservice Name	Resources		
	CPU (%)	**Memory Usage (MiB)**	**Network I/O (MB)**
advertisement-service	0.08	372.3	0.0178
api-gateway-service	0.06	403.8	0,0606
authentication-service	0.08	340.6	0.0179
blog-service	0.34	352.2	0.0178
career-service	0.14	347.7	0.0176
catalogue-service	0.08	378.3	0.0172
customer-support-service	0.08	317.9	0.0181
inventory-service	0.09	324.6	0.0181
loyalty-program-service	0.29	334.1	0.0182
merchant-service	0.14	352.5	0.0176
notification-service	0.08	535.2	0.0177
order-service	0.11	363.2	0.0176
payment-service	0.08	367.9	0.0173
policy-service	0.09	343.2	0.0179
review-service	0.08	393.7	0.0179
search-service	0.22	349.1	0.0178
service-discovery	1.16	433.8	0.0174
session-service	0.08	375.7	0.0175
shipping-service	0.29	325.8	0.0179
shopping-cart-service	0.19	351.4	0.0177
user-service	0.07	333.6	0.0185

Table 8.2.: Resource consumption statistics for Spring microservices when in idle state.

Scenario 2: All FastAPIMicroservices

In scenario two, we generated the E-commerce microservice application, which was entirely composed of FastAPI microservices. In order to realise this, we extended the MSA-Gen web application by introducing a new microservice generator to generate FastAPI-based microservices. It took us two weeks to learn the basics of FastAPI and develop the generator code with the associated definition of microservice template files.

Microservice Name	Resources		
	CPU (%)	Memory Usage (MiB)	Network I/O (MB)
advertisement-service	0.46	384.9	0.435
api-gateway-service	33.23	490.1	6.8
authentication-service	0.10	418	0.724
blog-service	0.21	390.6	0.0412
career-service	0.09	343.3	0.0409
catalogue-service	22.16	388.2	3.16
customer-support-service	8.18	374.5	0.552
inventory-service	10.36	404.9	1.63
loyalty-program-service	3.16	380.1	0.462
merchant-service	0.09	372	0.0413
notification-service	11.16	425,9	3.53
order-service	41.52	427.3	2.76
payment-service	6.88	363.3	0.407
policy-service	7.34	386.3	0.194
review-service	0.10	363.9	0.287
search-service	0.09	342.6	0.328
service-discovery	1.09	462.7	0.46
session-service	8.31	369	3.68
shipping-service	12.64	438.3	1.19
shopping-cart-service	83.08	401.7	6.78
user-service	0.09	376.7	0.633

Table 8.3.: Resource consumption statistics for Spring microservices when receiving requests.

Next, we present the recorded temporal KPIs measurements in Table 8.4 for the three trial runs performed. For this scenario, we observed the average value for "Microservices Generation Time" to be 286ms, "Microservices Build and Deployment Time" to be 116s and "Microservices Startup and Registration Time" to be 43s. Furthermore, Table 8.5 represents the container resource consumption statistics of the different microservices in their idle state, and Table 8.6 depicts the container resource consumption statistics during non-idle conditions when external API requests were being executed. Comparing the two states, we again notice an increase in the container resource consumption of microservices in their non-idle state compared to its counterpart.

Scenario 3: Polygot Microservices

The successful extension of MSA-Gen to support the generation of microservices in Spring and FastAPI technologies allowed us to evaluate scenario three. Here, we generated a polyglot E-commerce microservice application composed of nine FastAPI microservices and ten Spring microservices. We recorded the temporal KPIs measurements in Table 8.7 for the three trial runs we executed. For this scenario, we observed the av-

Trial	KPI		
	Microservices Generation Time (ms)	Microservices Build and Deployment Time (s)	Microservices Startup and Registration Time (s)
Trial 1	381 ms	154 s	40 s
Trial 2	240 ms	102 s	45 s
Trial 3	237 ms	90 s	42 s
Ceiled Average	286 ms	116 s	43 s

Table 8.4.: Table capturing the data for temporal KPIs for FastAPI microservices scenario.

Microservice Name	Resources		
	CPU (%)	Memory Usage (MiB)	Network I/O (MB)
advertisement-service	0.11	45.17	0.0474
authentication-service	0.14	45.2	0.0475
blog-service	0.13	44.31	0.0471
career-service	0.18	46.23	0.0468
catalogue-service	0.17	43.78	0.0471
customer-support-service	0.15	44.44	0.0471
inventory-service	0.17	47.14	0.0471
loyalty-program-service	0.15	44.26	0.0471
merchant-service	0.18	44.4	0.0473
notification-service	0.17	44.3	0.0473
order-service	0.15	46.36	0.047
payment-service	0.16	43.79	0.0473
policy-service	0.17	44.27	0.047
review-service	0.18	43.82	0.0473
search-service	0.17	43.73	0.0471
session-service	0.16	43.88	0.0471
shipping-service	0.14	44.4	0.0474
shopping-cart-service	0.2	44.38	0.0473
user-service	0.20	46.25	0.0474

Table 8.5.: Resource consumption statistics for FastAPI microservices when in idle state.

erage value for “Microservices Generation Time” to be 380ms, “Microservices Build and Deployment Time” to be 149s and “Microservices Startup and Registration Time” to be 95s.

Apart from this, Table 8.8 represents the container resource consumption of the different microservices in their idle state, and Table 8.9 depicts the container resource consumption during non-idle conditions when external API requests were being executed. Comparing the two states, we again see an increase in the container resource consumption of microservices in the non-idle state compared to the idle state.

Microservice Name	Resources		
	CPU (%)	Memory Usage (MiB)	Network I/O (MB)
advertisement-service	0.23	48.41	0.479
authentication-service	0.18	47.64	0.719
blog-service	0.16	44.3	0.189
career-service	0.18	46.23	0.186
catalogue-service	23.19	47.15	3.23
customer-support-service	0.24	46.98	0.425
inventory-service	12.92	50.49	1.85
loyalty-program-service	0.19	47.03	0.494
merchant-service	0.13	44.4	0.188
notification-service	0.16	47.75	3.37
order-service	0.15	49.61	1.92
payment-service	0.13	45.86	0.441
policy-service	0.20	45.49	0.255
review-service	0.24	46.28	0.34
search-service	0.14	46.45	0.376
session-service	16.14	46.98	3.88
shipping-service	15.98	47.47	1.24
shopping-cart-service	41.08	49.76	5.73
user-service	0.17	49.49	0.897

Table 8.6.: Resource consumption statistics for FastAPI microservices when processing requests.

Trial	KPI		
	Microservices Generation Time (ms)	Microservices Build and Deployment Time (s)	Microservices Startup and Registration Time (s)
Trial 1	509 ms	153 s	95 s
Trial 2	321 ms	146 s	96 s
Trial 3	310 ms	148 s	94 s
Ceiled Average	380 ms	149 s	95 s

Table 8.7.: Table capturing the data for temporal KPIs for Polyglot microservices scenario.

Scenario 4: Cyclic Dependency Check

When evaluating this scenario, we observed no issues in MSA-Gen to process the MAPI-DM with cyclic inter-service API dependencies and generating microservices. But the composition and deployment of the microservice application using the Docker composition tool failed due to the presence of cyclic dependency.

Microservice Name	Resources		
	CPU (%)	Memory Usage (MiB)	Network I/O (MB)
advertisement-service[s]	0.09	357	0.0791
api-gateway-service[s]	0.07	411.6	0.277
authentication-service[f]	0.14	46.42	0.0979
blog-service[s]	0.09	347.8	0.0795
career-service[f]	0.14	46.39	0.0973
catalogue-service[s]	0.10	363.6	0.0791
customer-support-service[f]	0.18	46.52	0.0983
inventory-service[s]	0.10	337.7	0.0789
loyalty-program-service[f]	0.18	44	0.0973
merchant-service[s]	0.09	330.9	0.0793
notification-service[f]	0.19	46.41	0.983
order-service[s]	0.11	346.2	0.0778
payment-service[f]	0.17	46.14	0.097
policy-service[s]	0.09	327.2	0.0791
review-service[f]	0.15	44.52	0.0976
search-service[s]	0.10	371.5	0.0774
service-discovery[s]	0.90	501	1.27
session-service[f]	0.19	48.25	0.0976
shipping-service[s]	0.10	343.3	0.0795
shopping-cart-service[f]	0.16	44.91	0.0978
user-service[s]	0.09	350.6	0.0787

Table 8.8.: Resource consumption statistics for Polyglot E-commerce application when idle. Superscript:- “s”:Spring ; “f”:FastAPI.

Graphical Visualisation of Tabular Data

In addition to this, we created graphs to visualise the tabular data that contain detailed KPI and resource measurements. Figure 8.4 compares the average microservice generation time. Next, Figure 8.5 helps us visualise and compare the outcome for average microservice build and deployment times. After that, we captured the result comparing the average microservices startup and registration time measures in Figure 8.6. Finally, Figure 8.7 provides an overview of all the temporal KPIs as part of a single graph for the Spring, FastAPI and Polyglot microservice generation scenarios while Figure 8.8 draws a comparison between the memory resource consumed by Spring and FastAPI microservices during both idle and non-idle operation states.

8.1.3. Task 3: Microservice Application Observability

In the third evaluation task, we assessed the observability of the generated microservice application on the five criteria mentioned before by using the observability framework and tool that we integrated as part of this thesis. Next, we present our observations and

Microservice Name	Resources		
	CPU (%)	Memory Usage (MiB)	Network I/O (MB)
advertisement-service[s]	0.12	379.6	0.412
api-gateway-service[s]	19.47	496.9	3.12
authentication-service[f]	0.16	50.38	0.855
blog-service[s]	0.09	349	0.143
career-service[f]	0.18	46.39	0.176
catalogue-service[s]	13.06	402	10.8
customer-support-service[f]	0.97	50.03	0.667
inventory-service[s]	6.45	364.4	4.24
loyalty-program-service[f]	4.17	47.09	0.529
merchant-service[s]	0.10	332.7	0.143
notification-service[f]	18.10	51.54	14.6
order-service[s]	42.53	439.5	2.17
payment-service[f]	7.48	49.09	0.472
policy-service[s]	0.98	338.2	0.272
review-service[f]	0.22	45.51	0.312
search-service[s]	0.11	384.4	0.385
service-discovery[s]	1.29	511.5	2.57
session-service[f]	33.22	51.97	15
shipping-service[s]	12.41	365.5	1.45
shopping-cart-service[f]	52.29	52.95	26.3
user-service[s]	0.13	430.4	0.776

Table 8.9.: Resource consumption statistics for Polyglot E-commerce application when receiving requests. Superscript:- “s”:Spring ; “f”:FastAPI.

results for each criterion. Due to spatial limitations, we capture only a subset of the snapshots here and the remaining as part of Appendix B

- **Collection and Visualisation of Microservice Metrics:** Integrating OpenTelemetry observability framework and SigNoz APM tool allowed us to automatically collect and visualise application metrics for different latency measures, request rate, error percentage and key operations for each microservice. Furthermore, the framework and tool also offered the possibility to define custom metrics per microservice and the ability to create custom dashboards to visualise the data collected for the metrics defined.

- **Logs Aggregation:** Since we adopted service instance per container deployment pattern as discussed in Section 5.2.3. Consequently, each container constitutes a single service configured to write application logs to a log file and the container's standard input, output and error consoles. In order to achieve logs aggregation, we again used the observability framework OpenTelemetry and SigNoz APM tool to automatically collect and aggregate all the container logs and visualise them in

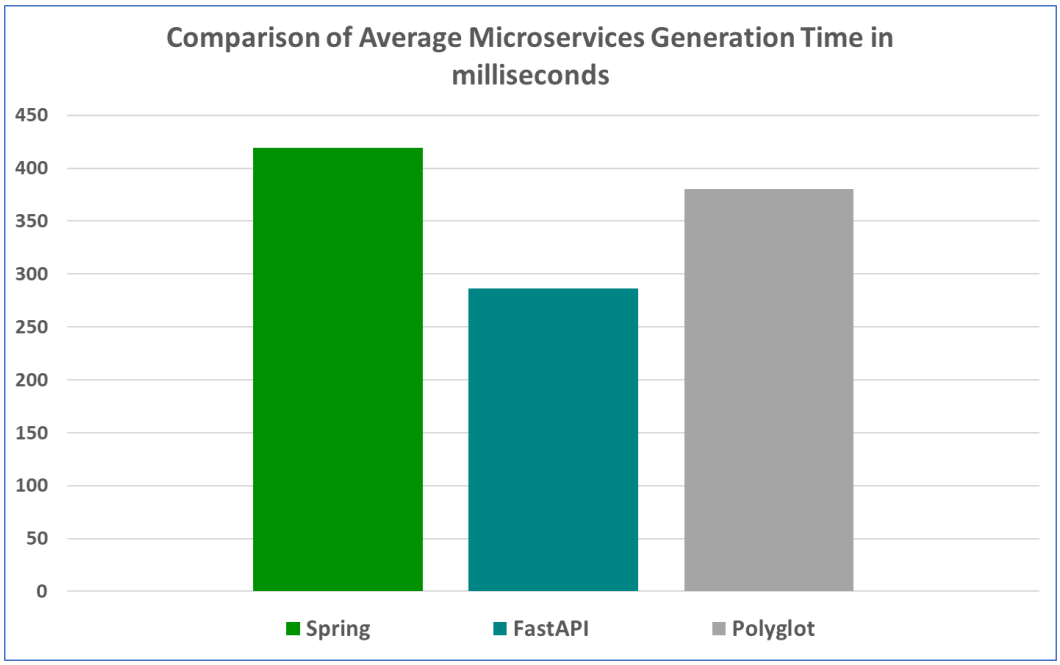

Figure 8.4.: Comparison of average microservice generation time for Spring, FastAPI and Polyglot generation scenarios.

a pre-defined logs dashboard.

- **Distributed Tracing of External Requests:** By using the instrumented Open-Telemetry telemetry instrumentation agent, we collected trace telemetry data to achieve distributed tracing of external requests. This agent assigned each request with a unique Trace identifier to map the entire trace of a request. In addition, it also sets a unique Span identifier for all the intermediary operations performed to fulfil the request. These identifiers are then mapped accordingly by the Observability tool to create visualisations showcasing the trace of an external request as it traverses across multiple microservices. This detailed trace data also allows us to pinpoint the location of errors or issues in servicing requests that require coordination across numerous microservices. Figure 8.9 shows an example of such a trace visualisation for a create purchase order API request where a drill down of the trace reveals the failure of the request due to an error at “shipping-service”.
- **Exception Tracking:** The observability tool SigNoz integrated as part of the deployed MSA offered the capability to isolate exceptions from the collected logs in order to visualise them separately under a different dashboard. We leveraged this feature to achieve exception tracking. In addition, to identify error-prone services on a high level, we also used the service-map view. This view provides a dynamic structural overview of all the microservices in the MSA. In this view, SigNoz represents error-prone microservices in red, which are otherwise in green.

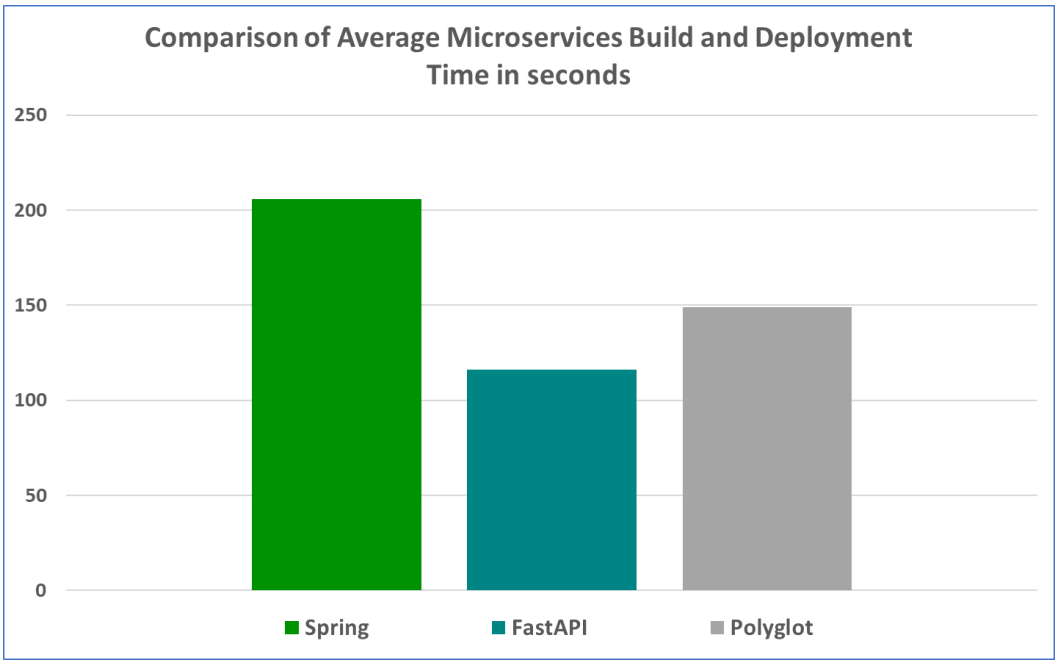

Figure 8.5.: Comparison of average microservice build and deployment time for Spring, FastAPI and Polyglot generation scenarios.

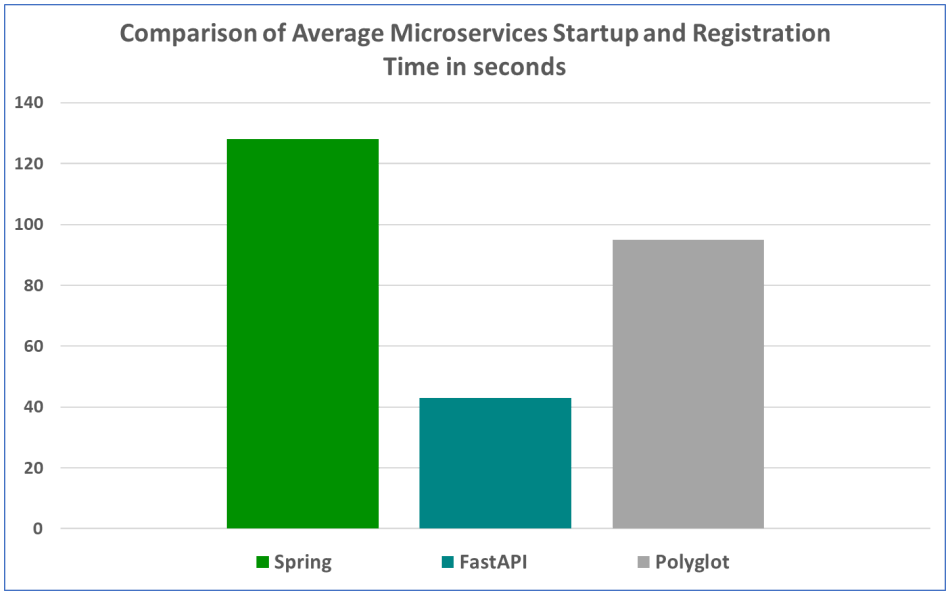

Figure 8.6.: Comparison of average microservice startup and registration time for Spring, FastAPI and Polyglot generation scenarios.

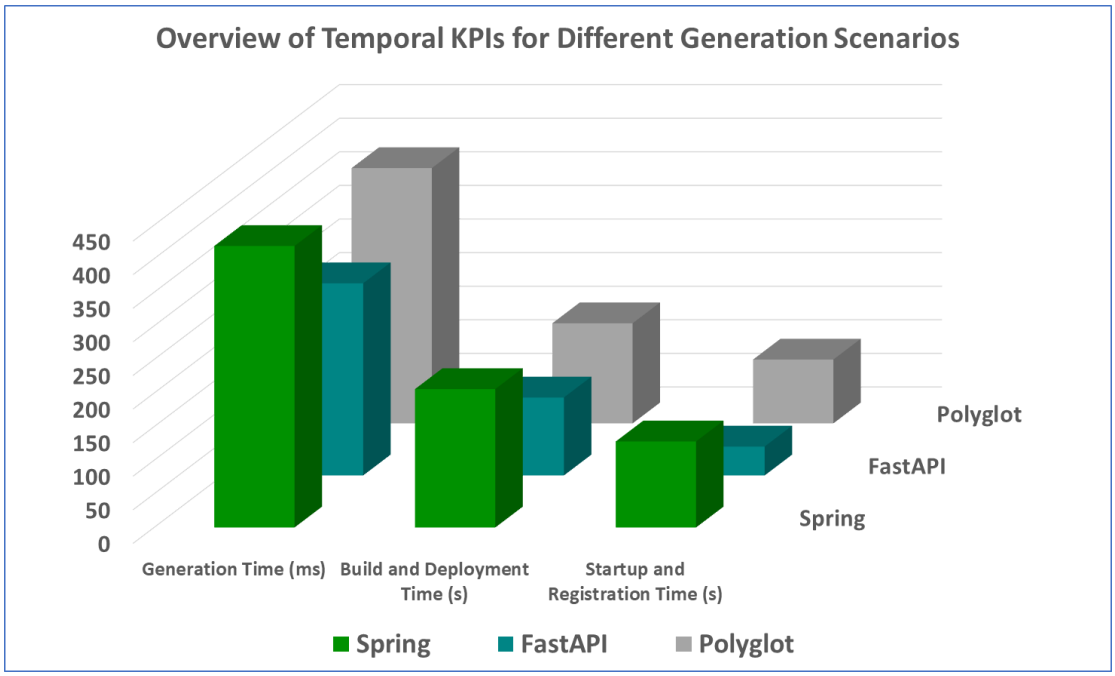

Figure 8.7.: Overview of all the temporal KPIs for Spring, FastAPI and Polyglot generation scenarios.

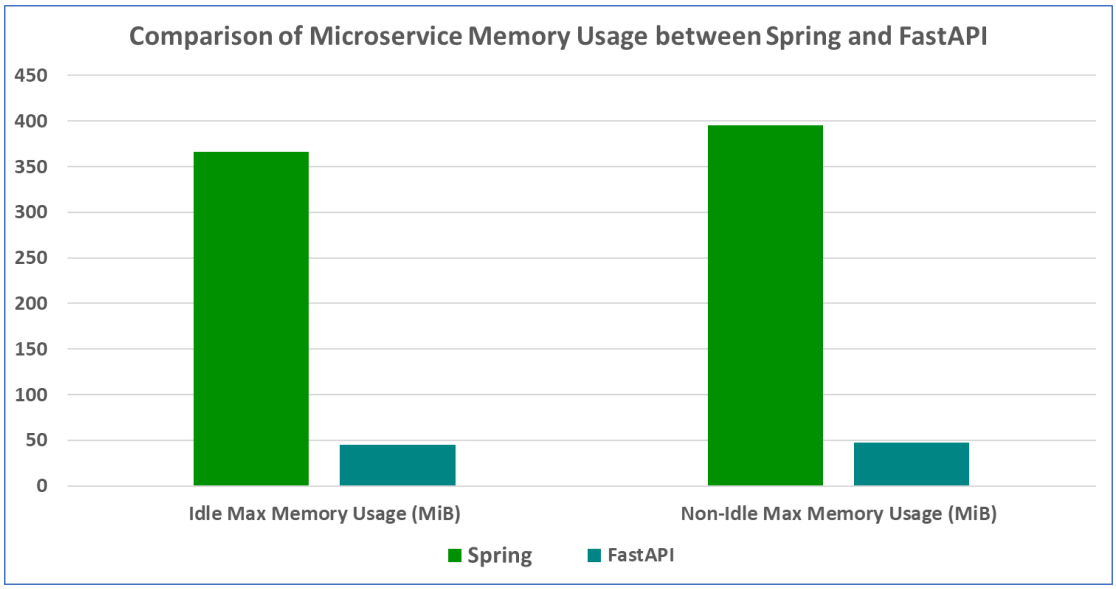

Figure 8.8.: Memory usage comparison between Spring and FastAPI microservices.

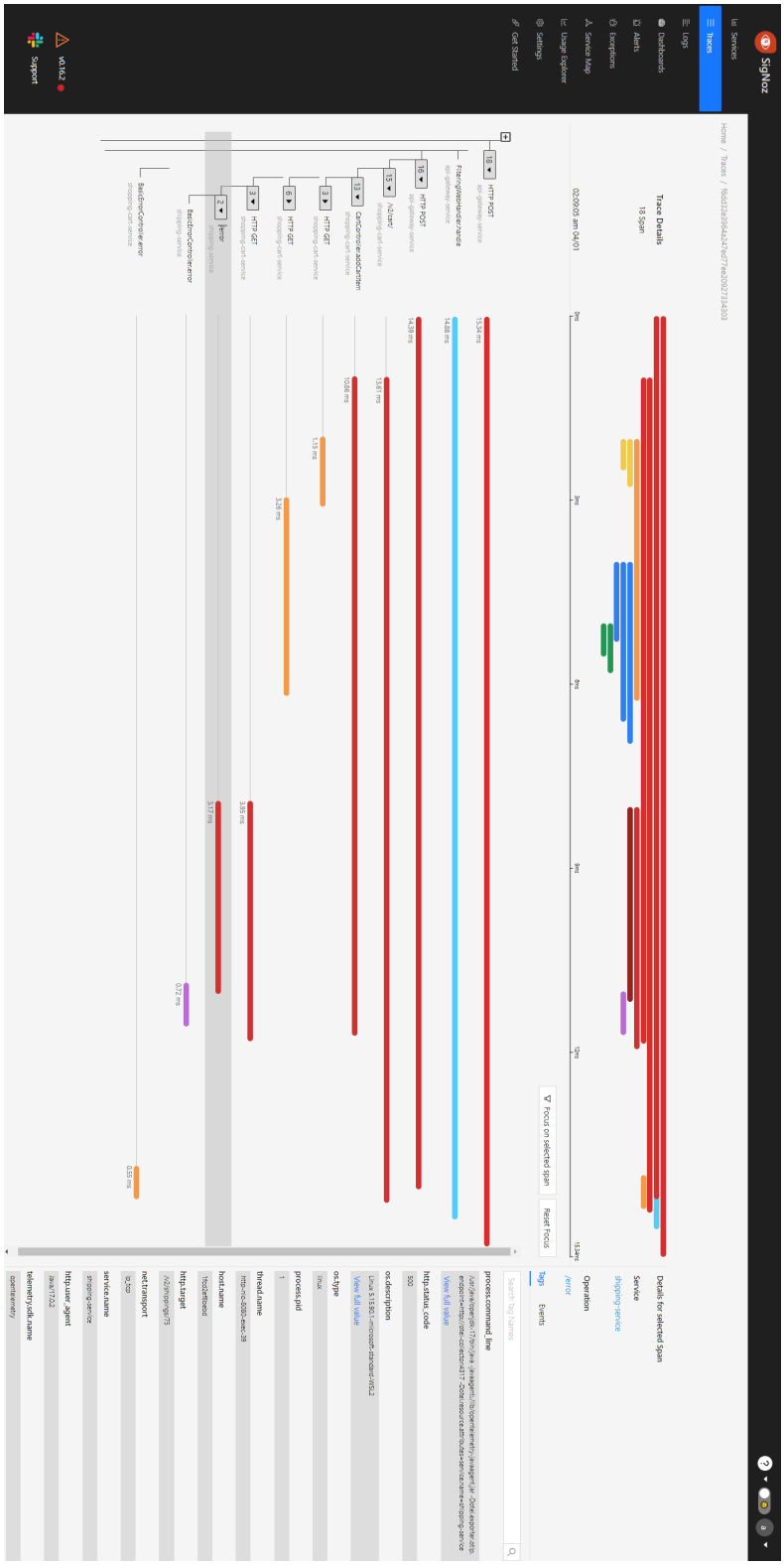

Figure 8.9.: Dashboard to visualise and trace external requests received by microservice applications.

- **Microservices Health Check:** Each microservice registers itself after startup with the Service Discovery. It then sends periodic heartbeats to the Service Discovery to indicate its health status. The Service Discovery maintains a list of healthy registered services and offers a dashboard to visualise this list.

After recording the results for each evaluation task, in the following section, we dive deeper into the evaluation results to analyse and interpret the outcomes.

8.2. Discussion of the Evaluation Results

This section examines the results captured for each evaluation task in Section 8.1 to gain profound insights into the outcomes of the evaluation process. For each task, we interpret the results, perform mutual comparisons wherever applicable and finally summarise and document our understanding and arguments.

8.2.1. Task 1: MAPI-DM Modelling

Based on our evaluation task of modelling MAPI-DM for the fictitious E-commerce microservice application, we have noted our observations of the benefits and drawbacks of using MAPI-DM. The benefits and drawbacks captured are exclusively based on our experience using MAPI-DM to model the E-commerce case study microservice application. Our claims, nonetheless, need to be qualitatively validated externally in the future.

Benefits

- The model has a simplistic structure, which makes it easy to learn and adopt.
- With a short learning curve, one can use it to model large-scale microservice applications quickly, ranging from a few hours to one or two days, considering the availability of the necessary prerequisite data.
- The model is also independent of the prerequisite data representation, allowing users the flexibility to choose appropriate representations to define the prerequisite data.
- The model has a technology and programming language independent representation. As a result, we can use it to represent microservice applications written in any programming language using any technology.
- It offers an extensible design allowing to add any newly identified attributes in the future.
- Finally, the model also encapsulates both the structural and relational data aspects of microservices when modelling them.

Drawbacks

- For enterprise-scale applications, the model can grow significantly in size as the overall number of microservices and their associated API endpoints increase. For the modelled E-commerce microservice application consisting of nineteen microservices, the model reached a total size of 2271 lines. Considering this, it will be a challenging and complicated task to manually model any enterprise-scale microservice applications consisting of hundreds to thousands of microservices.
- It can also be challenging to keep track of inter-service API dependencies and their sequential ordering as they increase over time.

8.2.2. Task 2: Microservice Generation Scenarios

We first compare our envisioned E-commerce architecture captured by Figure 7.3 to the generated counterpart illustrated by Figure 8.3, and we observe that they are very much alike. This similarity confirms that the MSA generated and composed by MSA-Gen successfully replicates all the modelled inter-service API communications between microservices which we interpreted through message sequence models and captured as part of the MAPI-DM data. This outcome also successfully demonstrates the mocking of our MSA case study through message sequence models. Next, we discuss our learnings from extending MSA-Gen by supporting FastAPI microservice generation.

- With the addition of a new microservice generator, we verified the extensibility characteristic of the developed MSA-Gen web application.
- During expansion, we faced several impediments when integrating the new technology with supporting frameworks like OpenTelemetry and Spring Eureka Service Discovery. To resolve them, we first referred to official documents and then to other online resources and experimental methods when there were gaps in the documentation.
- We also assimilated that it is necessary to understand the underlying programming language constructs and other technology-specific best practices to write efficient microservice templates for any technology.

After discussing our learnings on supporting polyglot microservice generation in MSA-Gen, we compare the different temporal KPIs measurements for microservice generation scenarios one, two and three and discuss our observations.

- As captured in Figure 8.4, we can visualise that, on average, the generation of Python-based FastAPI microservices is close to one and a half times quicker compared to the generation of Java-based Spring microservices. This significant time difference can be attributed to the different project structures each technology advocates, resulting in differences in the number of files generated for each technology. Comparing the Polyglot generation scenario, we can see that it has a generation

time greater than FastAPI but less than Spring. This measurement meets the expected outcome as it comprises a nearly equal distribution of microservices based on the two technologies.

- Visualising the results for the build and deployment time KPI as depicted in Figure 8.5, we again notice a similar overall trend where Java-based Spring microservices take the highest time due to the involvement of Maven build to generate project sources and artefacts for deployment. Since Python programs do not require a build of the project, their measurement for this KPI is significantly less, making their deployment twofold quicker as Spring microservices. Finally, Polyglot records a build and deployment time between the Spring and FastAPI results.

- Figure 8.6 depicts the graph showing that FastAPI microservices start and register three times quicker than Spring microservices. This considerable time difference is because Spring-based applications require a proportional amount of time for startup and initialising the bundled web application server. In contrast, the startup and initialisation of the bundled web application server in FastAPI is relatively instantaneous.

- Comparing the resource consumption between Spring and FastAPI microservices, we notice that the utilisation of both Network I/O and CPU resources are comparatively similar for the microservices of both these technologies and these values for each microservice fluctuate depending on the number of API requests it receives at any given time which we can corroborate by comparing the idle and non-idle trials captured for each generation scenarios in Section 8.1.2.

  In contrast, comparing the memory usage of Spring and FastAPI microservices, we notice an evident difference in value as illustrated in Figure 8.8. We can observe that the memory consumption of Spring microservices is eight times that compared to FastAPI microservices. This difference is significant, considering these microservices are only mocked versions that simulate inter-service API communications without any business logic getting executed in them.

  From this observation, we can conclude that any enterprise-scale microservice application consisting only of Spring-based microservices would require a significantly higher memory availability for its operation than FastAPI-based microservices. This high resource requirement leads to higher operational costs for enterprises as they need to rent more powerful machines to deploy their cloud-native microservice applications or higher development and maintenance costs concerned with optimising applications based on these technologies.

  However, we need to acknowledge that while FastAPI outperforms Spring in all the measured KPIs and has significantly lower memory consumption statistics, we do not say that FastAPI is more performant when compared to Spring. We cannot benchmark the holistic performance of these technologies solely based on mocked versions of microservices that carry no business logic.

For scenario four, we observed that it was impossible to compose and deploy MSAs that consisted of cyclic dependencies since we used the "depends_on" condition attribute offered by the composition tool. This attribute is used to express dependency between microservices. The composition tool then uses this information during deployment in order to deploy the microservices based on their dependency order. The deployment failure with cyclic dependency conditions is one of the substantial benefits of using our approach when mocking MSAs, as it provides an additional layer of check to prevent the deployment of microservice applications that included the cyclic dependency microservices anti-pattern [TLP20].

8.2.3. Task 3: Microservice Application Observability

Based on the evaluation results of task three, we documented our overall thoughts on the support of microservice application observability based on the considered observability parameters.

- The instrumentation of telemetry agents in each microservice and the integration of an observability tool allowed us to generate highly observable mocked MSAs.
- The various metrics collected for each microservice allowed us to infer its overall performance and identify bottleneck services.
- Tracing external requests allowed us to observe their flow through the mocked E-commerce MSA. In addition, it also allowed us to visualise the breakup of the processing time of individual requests as they passed through multiple microservices. Furthermore, this capability allowed us to pinpoint the exact position of error occurrences during the sequential invocation of inter-service API dependencies.
- Monitoring the health status of each microservice allowed us to identify service failures and take appropriate actions to rectify the situation.
- Incorporating log aggregation and exception tracking allowed to development of microservices that support faster debugging in case of errors.

The results successfully showcased the efficacy of observable microservices when developing large-scale MSAs. Nevertheless, selecting the appropriate telemetry framework like OpenTelemetry and an observability tool like SigNoz is essential to develop observable microservices that allow you to monitor the behaviour of such distributed systems.

9. Discussion

> In programming, the hard part isn't solving problems, but deciding what problems to solve.
>
> Paul Graham

Contents

In this chapter, we answer our research questions formulated in Section 1.1 and discuss the various threats to the validity of our research work in this thesis.

9.1. Answers to Research Questions

In this thesis, we embarked on the journey to find an approach to mock MSAs from message sequence models. We defined the following five research questions to achieve this. Next, we answer each research question using the knowledge we obtained from our research work in this thesis.

- **RQ1:** *How would a metamodel that can capture attributes of microservices exhibiting inter-service API communications materialise to be?*

  To answer **RQ1**, we adopted an iterative top to bottom modelling approach as discussed in Section 4.4 to identify different microservice attributes necessary to create their mocked representation. Using the various attributes identified as part of the iterative modelling approach, we defined our MAPI-DM metamodel, presented in Section 4.4.2. It can capture the structural definition of microservices and the relational aspect of their inter-service API dependencies in sequential order.

- **RQ2:** *How can we use concrete instance models of our metamodel to construct MSAs composed of mocked-up microservices that can replicate the behaviour of modelled inter-service API communications?*

  To answer **RQ2**, we first created MAPI-DM definitions which are concrete instances of our defined metamodel containing MSA data. Next, we investigated

ways to transform MAPI-DM into mocked MSAs. For this, we explored creating a prototype application capable of performing this transformation to generate mocked-up microservices. As part of our investigation, we adopted the rapid prototyping development strategy to design and develop a prototype web application MSA-Gen as explained in Section 5.1. This application uses MAPI-DM definitions to generate, compose and deploy MSAs in an automated manner which can be interacted with in real-time as described in Section 4.3. Finally, using the inter-service API dependency data obtained from MAPI-DM, we mocked the modelled functional behaviours in the generated MSAs.

- **RQ3:** *How can we automatically visualise the behaviour of the composed MSAs?*

  Next, to answer **RQ3**, we researched the concept of observability in MSAs. As part of our investigation, we adopted several MSA observability design patterns as presented in Section 5.2.4. It led to the instrumentation of the generated microservices with telemetry agents and the integration of an observability framework and tool described in Section 5.2.4 to automatically collect telemetry data exported by the microservices and use them to visualise and comprehend the behaviour of the composed MSAs. Section 8.1.3 presents the various telemetry data we automatically collect from the generated microservices allowing us to design highly observable MSAs.

- **RQ4:** *What are the learnings and challenges associated with generating polyglot MSAs?*

  To answer **RQ4**, we explored the expansion of MSA-Gen that was capable of generating only Spring-based MSAs to additionally generate MSAs composed of microservices implemented using more than one technology. In order to achieve this, we extended the MSA-Gen application by developing a second microservice generator to support the generation of FastAPI microservices. This extension allowed us to generate polyglot MSAs using MSA-Gen and to evaluate it against other deployment scenarios as presented in Section 8.1.2. In addition, we also captured our learnings from supporting polyglot microservice generation as part of Section 8.2.2

- **RQ5:** *What are the benefits and drawbacks of modelling large-scale MSAs using our metamodel?*

  Finally, to answer **RQ5**, our last research question, we developed a case study to generate and mock a fictitious large-scale E-commerce microservice application as discussed in Section 7.1.1. As part of this, we modelled the MAPI-DM for the case study application, which captures the MSA data. It gave us first-hand insights into the benefits and drawbacks of using our metamodel for modelling large-scale MSAs, which we discuss as part of Section 8.2.1.

9.2. Threats to Validity

In this section, we discuss the different threats to the validity of our research approach and the results presented in this thesis. We evaluate the applicability of the following three validity concepts based on the definitions presented by Wohlin et al. in [Woh+12] and by Feldt and Magazinius in [FM10].

9.2.1. Internal Validity

Internal validity focuses on the confidence associated with the technique used in an experiment to that of the observed outcomes. In our research, we defined MAPI-DM metamodel to capture microservice data. Since one can develop microservices using various technologies, we perceived a threat of unknowingly introducing technology-specific bias when defining our metamodel. To mitigate this, when modelling, we took a technology-agnostic approach. Our added support for a second microservice generator to generate polyglot microservice applications eliminated this threat.

9.2.2. External Validity

External validity examines if the results of an experiment can be generalised outside the context of a study. In our research, we realised that since we build and deploy the MSAs in real-time on a test system, factors like the test system configurations and network speed influenced the outcomes measured for the temporal KPIs. We conducted three trial runs for each deployment scenario to try and mitigate this risk and captured our test system configurations and network speed information in Section 7.1.3. This information allows other studies to repeat the case study evaluation with comparable test system configurations and compare the outcomes. However, even in this case, we acknowledge that the results might not be a replica of our observations but would predominantly fall closer to the average values captured in this thesis.

9.2.3. Construct Validity

Construct validity is concerned with assessing the extent to which the observations of an experiment conform to its underlying theory. In our research, we initially proposed a conceptual approach to use message sequence models to generate microservice applications to mock MSAs. We sensed a threat regarding the effectiveness and validity of our approach when generating and mocking enterprise-scale MSAs. In order to mitigate this risk, we defined a case study microservice application that resembled an enterprise-scale E-commerce MSAs and used this to evaluate the fundamental concepts defined to mocking MSAs.

10. Conclusion and Future Work

> The best way to predict the future is to invent it.
>
> Alan Kay

Contents

After gaining answers to the research questions we formulated in this thesis and discussing the threats to the validity of our research approach and findings, in this chapter, we first summarise our work by outlining the contributions of this thesis. Finally, we mark the end of our journey by proposing possible paths for future work that can mark the beginning of many new ones.

10.1. Summary

In this thesis, our primary aim was to discover a way to mock MSAs using message sequence models that illustrate the functional requirements of MSAs. In order to achieve this, we first defined MAPI-DM metamodel. This metamodel allows us to define MAPI-DM whose declaration is independent of the representation of message sequence models. It can capture various structural attributes of microservices and their relational aspects of inter-service API dependencies that are essential for mocking functional requirements of MSAs.

Next, we investigated code generation using MDD concepts to generate deployable microservices from MAPI-DMs. We developed a prototype application MSA-Gen, capable of parsing MAPI-DMs and automatically generating, composing and deploying observable microservice applications. Once the applications were composed and deployed, we could interact with them, mock the modelled functional requirement of MSAs and observe their behaviour in real-time.

We next investigated supporting polyglot MSAs in MSA-Gen. We achieved this by extending MSA-Gen to support microservice generation in multiple technologies, primarily Spring and FastAPI. It allowed us to go beyond our primary aim and mock polyglot MSAs. Then, we focused our research on enhancing the observability aspects of the mocked MSAs that would help us to observe their behaviour during runtime. In order to achieve this, we explored the integration of observability frameworks and tools

to automatically collect various telemetry data of microservices like logs, metrics, and traces.

Finally, to evaluate our contributions to this thesis, we present a case study where we define, generate and mock a large-scale E-commerce microservice application using the artefacts developed as part of this thesis. To conclude, we answer our research questions based on our findings from the evaluation of our case study.

10.2. Future Work

The scope of extension and future work to extend this thesis are many. In this section, we briefly discuss some possible areas one can explore to extend our work in this thesis to answer many new and novel research questions.

- **Automated Generation MAPI-DM:** In this thesis, we manually modelled the MAPI-DM by capturing information from message sequence models. Based on our learnings from the case study, manual modelling of large-scale microservice applications can become tedious, complex and time intensive. Future studies can explore the automated generation of MAPI-DM to address some of these concerns and make the modelling process more user-friendly and straightforward.

- **New Microservice Generators:** In this study, we demonstrated the automated generation of microservices developed in Spring and FastAPI technologies. Prospective research extending this thesis can explore supporting new microservice generators that implement different popular technologies used in the industry to develop microservice applications.

- **Beyond Mocking MSAs:** The aim of this thesis was to mock MSAs using message sequence models. Accordingly, the current realisation is capable of generating mocked microservices. Future research can go beyond mocking to study and investigate the automated generation of fully functional microservices possessing rudimentary business logic.

- **Container Orchestration Support:** As part of this thesis, we generate Docker compose files for automatic composing and deploying microservice applications. Future studies can explore the automatic generation of orchestration resources using technologies like Kubernetes [1], which allows for finer orchestration of containers and offers many benefits like simplified cloud deployment, horizontal scaling, secrets and configuration management, automated rollbacks and many other features that are relevant for cloud-native enterprise applications to support.

- **Asynchronous Communication Support:** In this thesis, we restricted the scope of communication between microservices to include only the REST-based synchronous communication style. This scope limitation presents future studies

[1] https://kubernetes.io/ (accessed on 18.05.2023)

with the prospects of extending our work in this thesis by generating microservices to support other synchronous, asynchronous and messaging modes of communication widely used in industry to develop enterprise-scale MSAs.

- **Generating Secure MSAs:** As part of this thesis, we incorporated non-root users in the deployed containers to generate secure microservices. Apart from this, other security-related aspects of the generated microservices remain largely unexplored. It offers future studies an enormous potential to close this gap by researching ways to generate highly secure microservice applications.
- **Validation with Domain Experts and Industries:** As part of this thesis, we formulated a case study to validate the thesis artefacts. Future research can take this further by conducting interviews with domain experts in the fields related to MSA to obtain more profound and holistic insights into the benefits and challenges in using MAPI-DM and MSA-Gen for mocking MSAs. In addition, industrial collaborations can be conducted to test the prototype with more realistic data to assess its benefits and relevance in industries and identify limitations to determine further research gaps in this domain.

A. Message Sequence Models of E-commerce Application Case Study

This chapter illustrates the remaining message sequence models representing the inter-service API dependencies for the functional requirements identified in Table 7.1 to model the discussed example user interaction with the E-commerce microservice application.

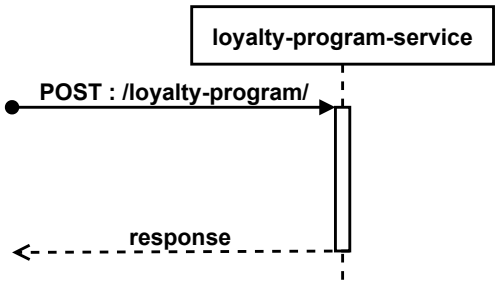

Figure A.1.: Message sequence model illustrating the loyalty-bonus program registration functional requirement (FR2).

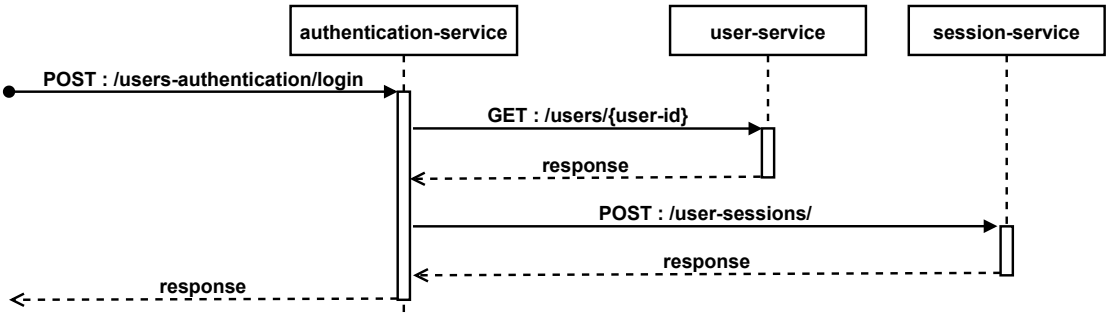

Figure A.2.: Message sequence model illustrating the login functional requirement (FR3).

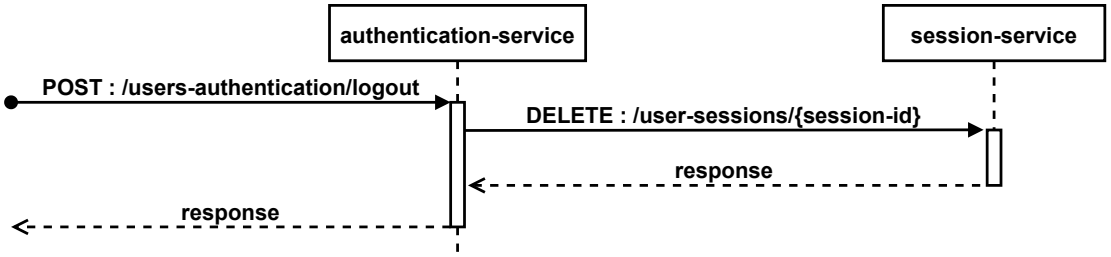

Figure A.3.: Message sequence model illustrating the logout functional requirement (FR4).

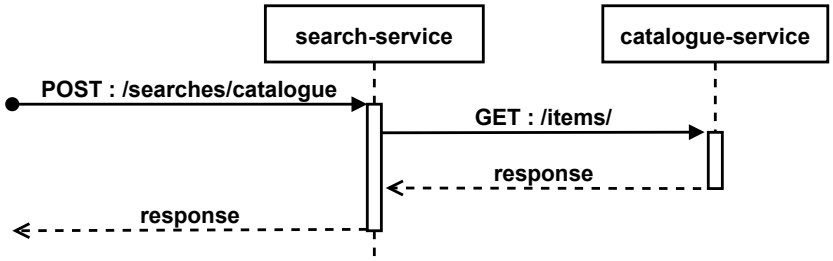

Figure A.4.: Message sequence model illustrating the catalogue search functional requirement (FR5).

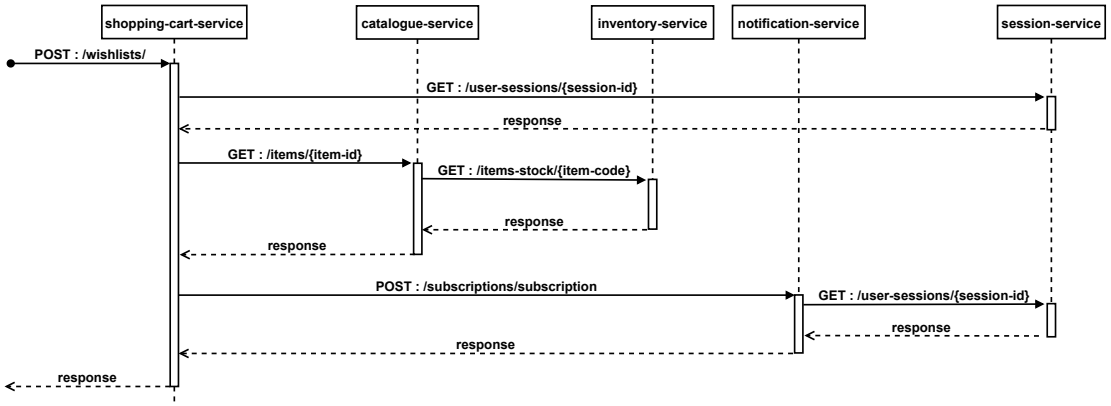

Figure A.5.: Message sequence model illustrating wishlists creation functional requirement (FR6).

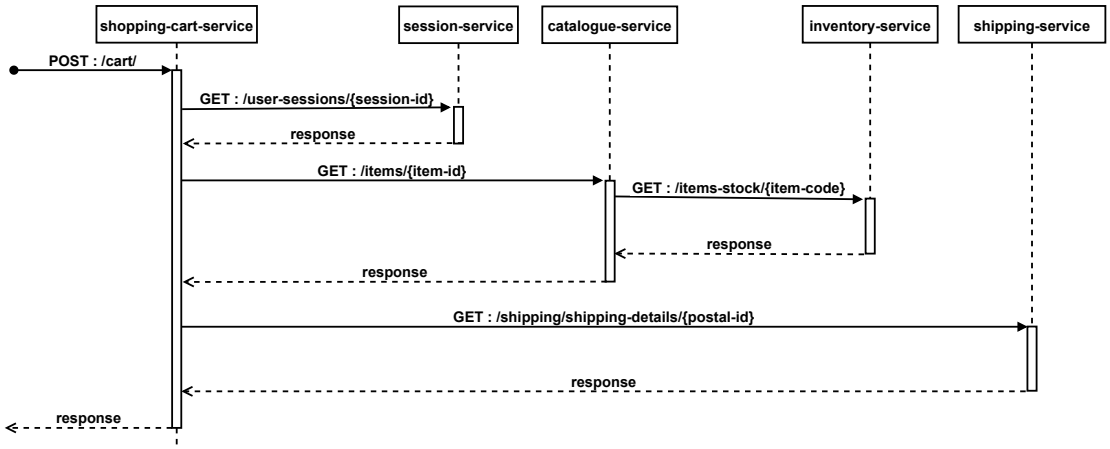

Figure A.6.: Message sequence model illustrating the functional requirement to add items to the shopping cart (FR7).

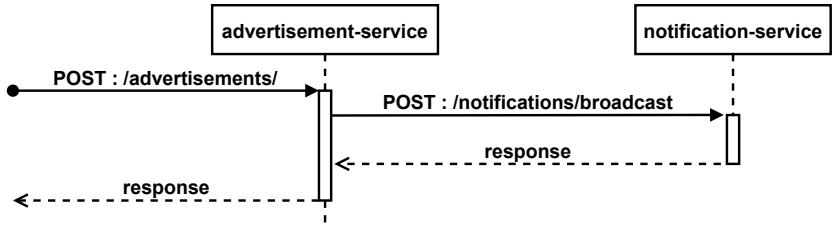

Figure A.7.: Message sequence model illustrating the functional requirement to create and publish advertisements (FR8).

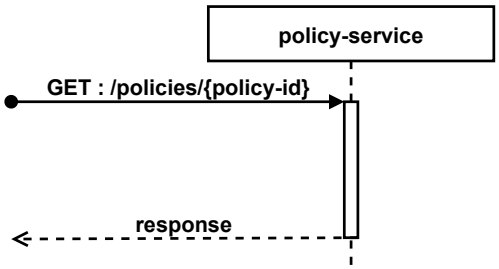

Figure A.8.: Message sequence model illustrating the functional requirement to retrieve policies (FR10).

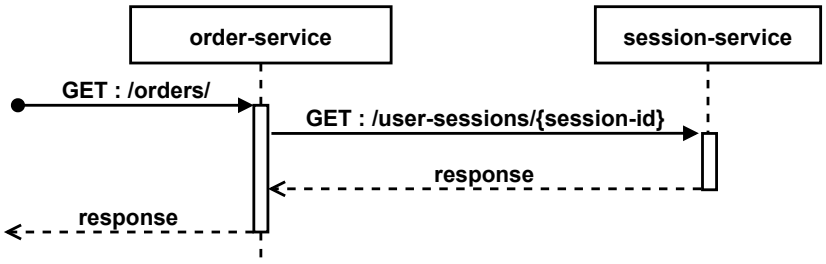

Figure A.9.: Message sequence model illustrating the functional requirement to retrieve all purchasing orders of a given user (FR12).

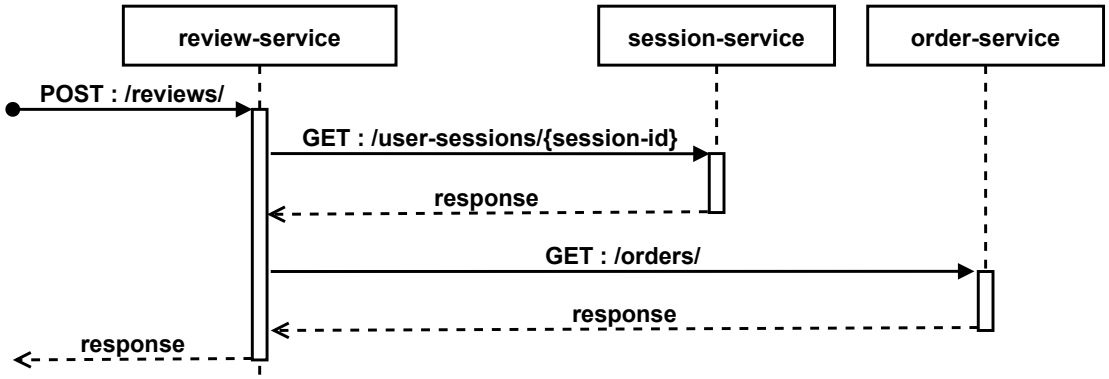

Figure A.10.: Message sequence model illustrating the functional requirement to add reviews to the purchased products (FR13).

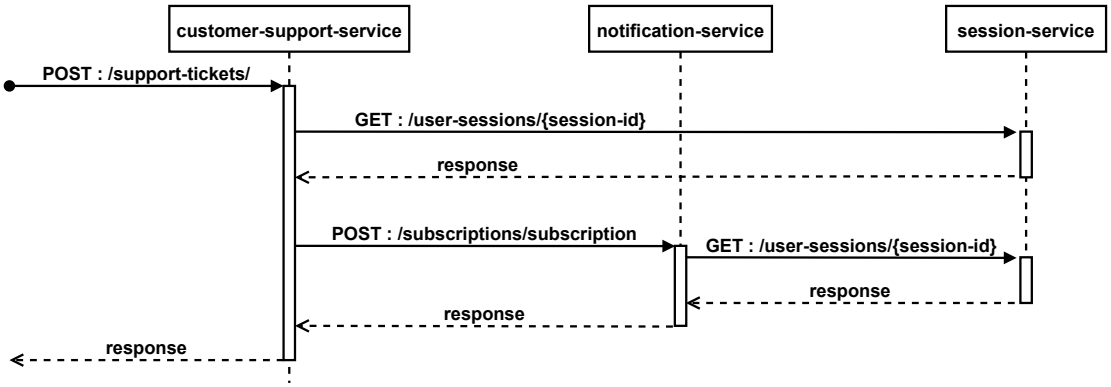

Figure A.11.: Message sequence model illustrating the functional requirement to create customer-support tickets (FR14).

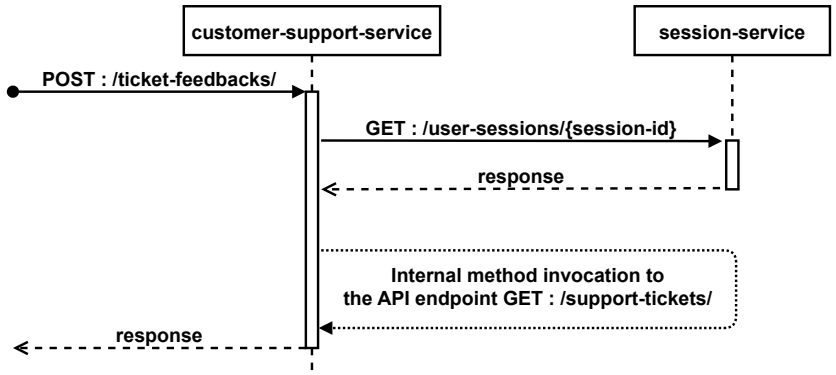

Figure A.12.: Message sequence model illustrating the functional requirement to provide feedback to customer support (FR15).

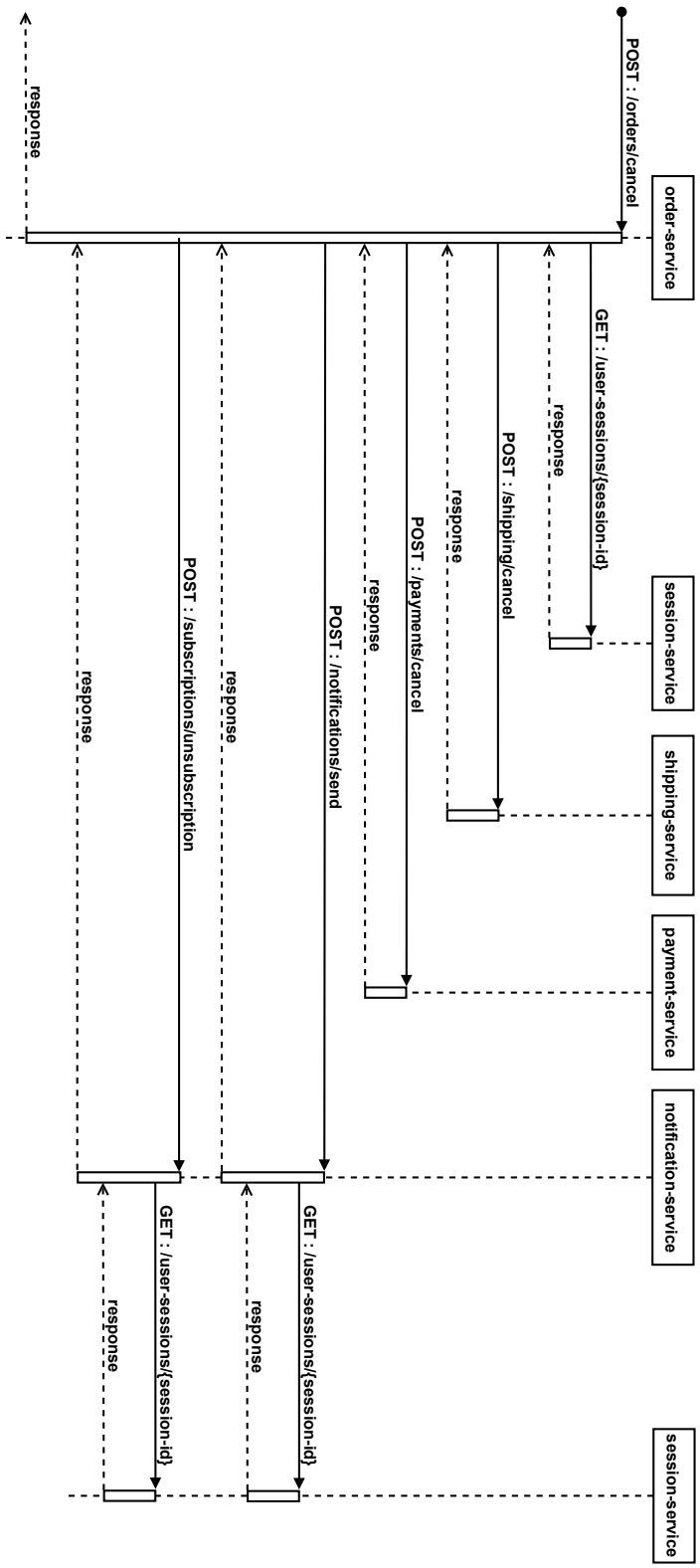

Figure A.13.: Message sequence model illustrating the functional requirement to cancel a purchasing order (FR11).

B. MSA Observability Snapshots

This chapter presents the detailed snapshots we captured while evaluating task three concerning microservice application observability as discussed in Section 8.1.3.

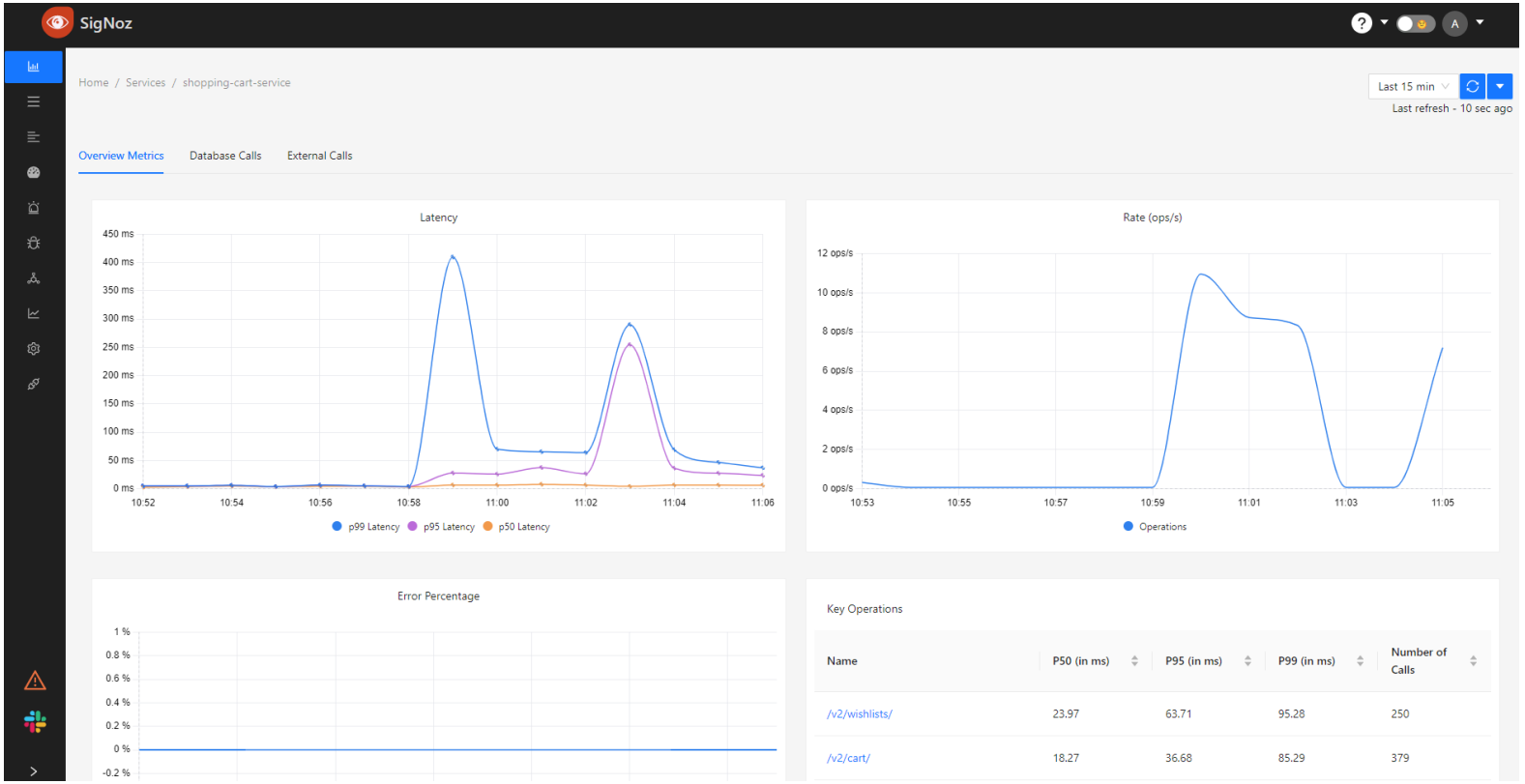

Figure B.1.: Dashboard showing latency measures, request rate, error percentage and key operations metrics collected for "shopping-cart-service" during the non-idle state. We can infer that as the request rate to the microservice increases, the latency also spikes.

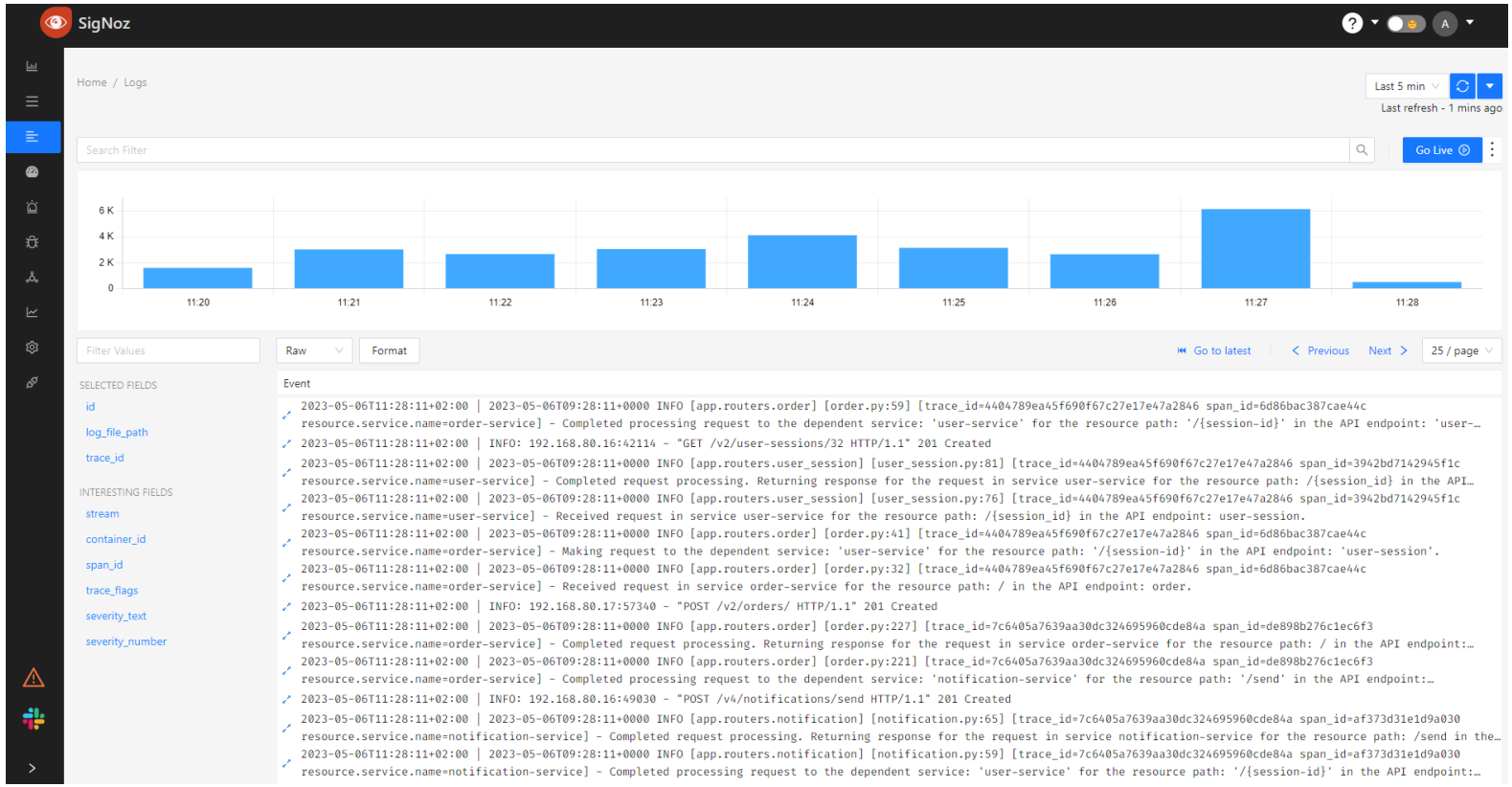

Figure B.2.: Log aggregation dashboard that allows visualisation of logs collected for all deployed microservices. From the figure, we can see the logs events of “session-service”, “order-service”, and “notification-service”

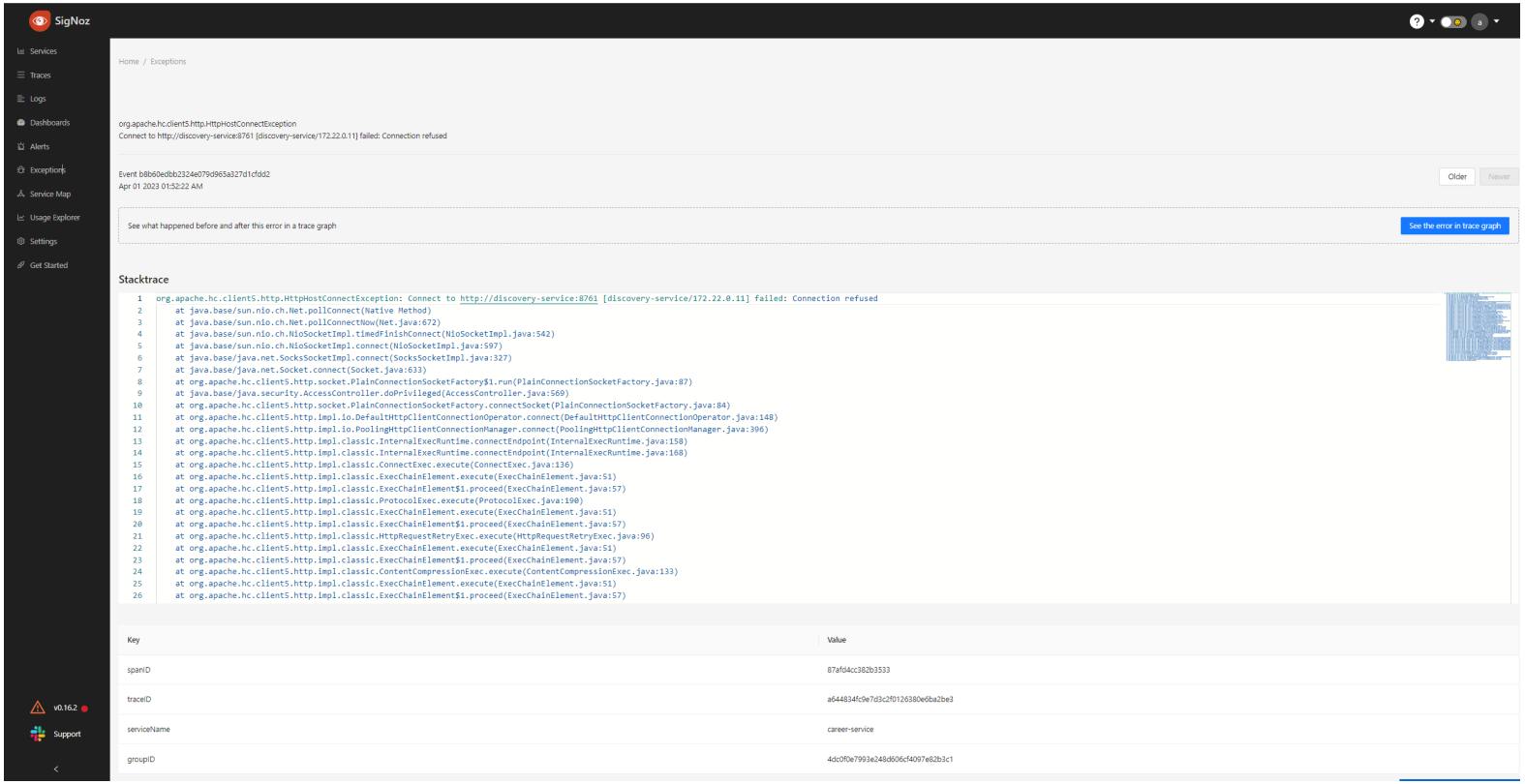

Figure B.3.: Snapshot of the exceptions dashboard that allows us to visualise and track microservice exceptions. For each exception, relevant details are captured and displayed with the associated event identifier allowing us to trace the exception to its origin.

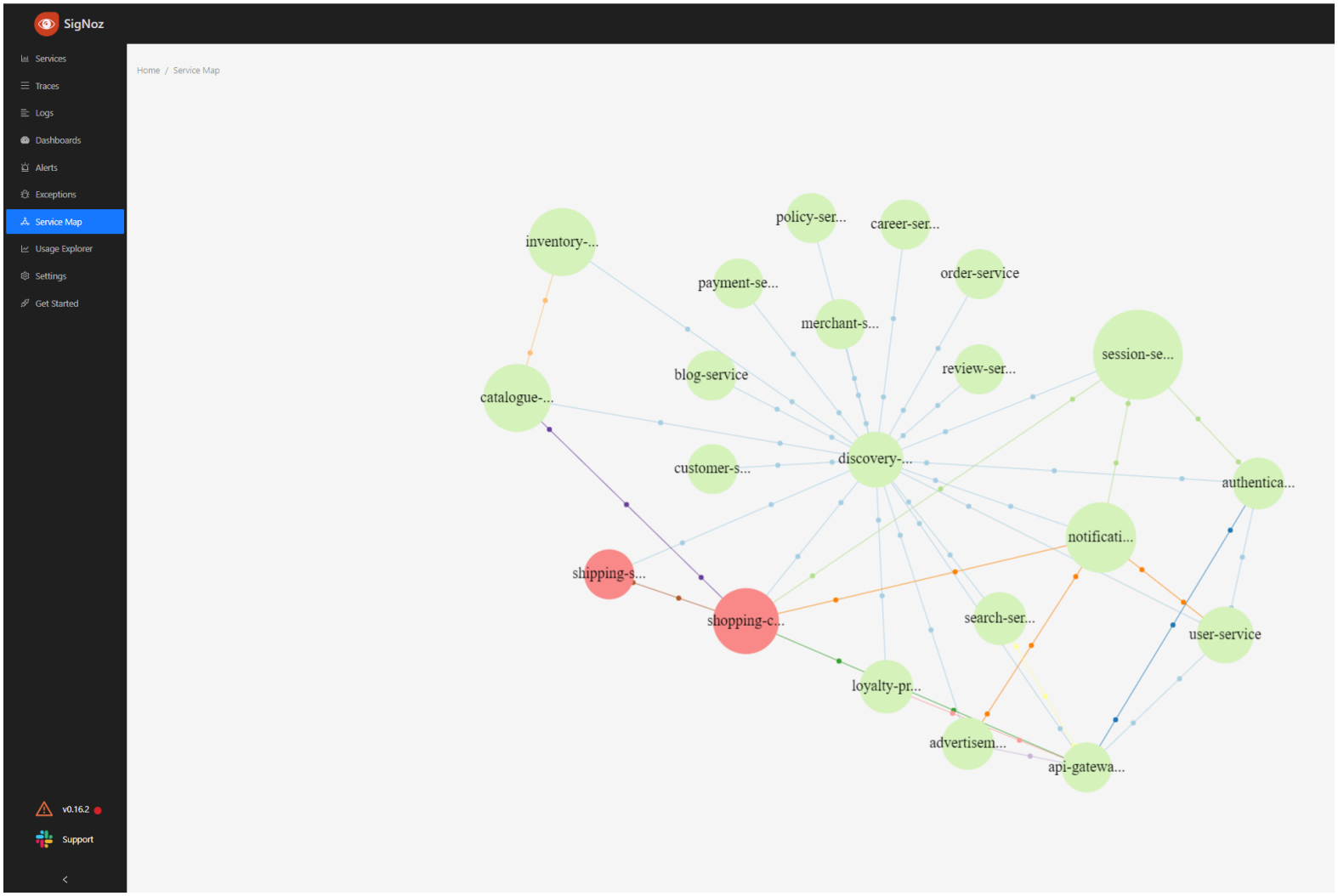

Figure B.4.: Service map view of the E-commerce microservice application representing the error-prone microservices in red and the error-free services in green.

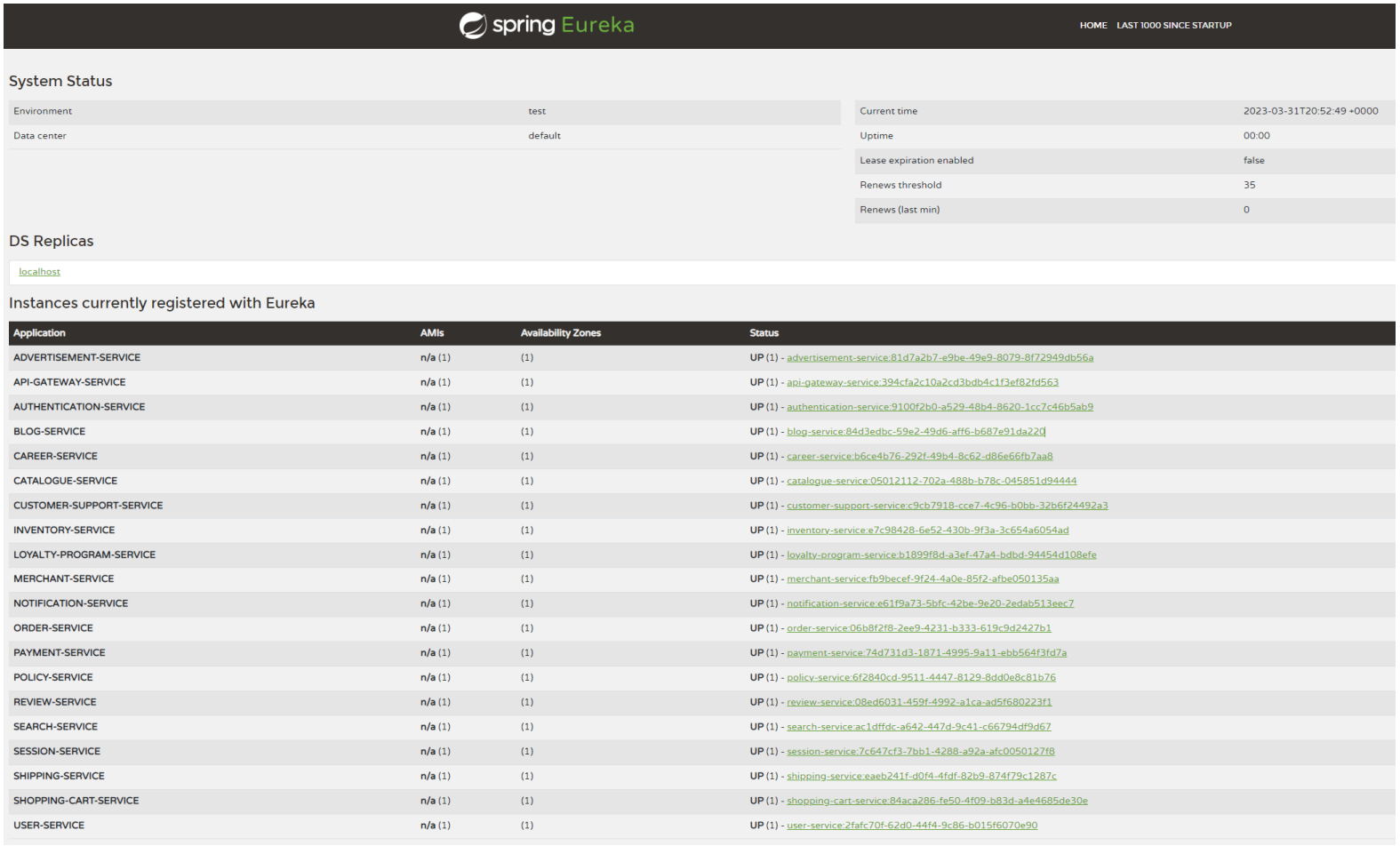

Figure B.5.: Service Discovery dashboard lets us visualise the list of registered microservices and their health status. The status column displays “UP” for reachable healthy microservices and “DOWN” for unreachable unhealthy microservices.

Bibliography

[Ame10] D. Ameller. "SAD: Systematic Architecture Design: A Semi-automatic Method." 2010 (cit. on p. 9).

[BHJ16a] A. Balalaie, A. Heydarnoori, and P. Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture." In: *IEEE Software* 33.3 (May 2016), pp. 42–52. ISSN: 1937-4194. DOI: 10.1109/MS.2016.64 (cit. on p. 4).

[BHJ16b] A. Balalaie, A. Heydarnoori, and P. Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture." In: *IEEE Software* 33.3 (May 2016), pp. 42–52. ISSN: 1937-4194. DOI: 10.1109/MS.2016.64 (cit. on p. 9).

[CK21] M. Chakraborty and A. P. Kundan. "Introduction to Modern Monitoring." In: *Monitoring Cloud-Native Applications: Lead Agile Operations Confidently Using Open Source Software*. Berkeley, CA: Apress, 2021, pp. 3–24. ISBN: 978-1-4842-6888-9. DOI: 10.1007/978-1-4842-6888-9_1. URL: https://doi.org/10.1007/978-1-4842-6888-9_1 (cit. on p. 9).

[Com+20] B. Combemale et al. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. CRC Press, June 2020. ISBN: 9780367574215 (cit. on p. 9).

[DH17] T. F. Düllmann and A. van Hoorn. "Model-Driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches." In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE '17 Companion. L'Aquila, Italy: Association for Computing Machinery, 2017, pp. 171–172. ISBN: 9781450348997. DOI: 10.1145/3053600.3053627 (cit. on pp. 12, 13).

[FM10] R. Feldt and A. Magazinius. "Validity Threats in Empirical Software Engineering Research - An Initial Survey." In: *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010)*. Knowledge Systems Institute Graduate School, 2010, pp. 374–379 (cit. on p. 87).

[Fow+02] M. Fowler et al. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002, pp. 330–332. ISBN: 0321127420 (cit. on p. 33).

[FY97] B. Foote and J. W. Yoder. "Big Ball of Mud." In: *Proceedings of the 4th Pattern Languages of Programming Conference (PLoP'97/EuroPLoP'97)*. Washington University Technical Report, Sept. 1997 (cit. on p. 40).

[Gam+94] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Vol. 47. Addison-Wesley Professional, Oct. 1994, pp. 107–116, 315–323. ISBN: 0201633612 (cit. on pp. 36, 38).

[Gar18] M. Garriga. "Towards a Taxonomy of Microservices Architectures." In: *Software Engineering and Formal Methods*. Ed. by A. Cerone and M. Roveri. Springer International Publishing, 2018, pp. 203–218. ISBN: 978-3-319-74781-1 (cit. on p. 4).

[Gat21] R. Gatev. "Observability: Logs, Metrics, and Traces." In: *Introducing Distributed Application Runtime (Dapr): Simplifying Microservices Applications Development Through Proven and Reusable Patterns and Practices*. Berkeley, CA: Apress, 2021, pp. 233–252. ISBN: 978-1-4842-6998-5. DOI: 10.1007/978-1-4842-6998-5_12. URL: https://doi.org/10.1007/978-1-4842-6998-5_12 (cit. on p. 9).

[GBS17] D. Gannon, R. Barga, and N. Sundaresan. "Cloud-Native Applications." In: *IEEE Cloud Computing* 4.5 (Sept. 2017), pp. 16–21. ISSN: 2325-6095. DOI: 10.1109/MCC.2017.4250939 (cit. on p. 1).

[Gok+21] M. Gokan Khan et al. "PerfSim: A Performance Simulator for Cloud Native Microservice Chains." In: *IEEE Transactions on Cloud Computing* (2021), pp. 1–1. ISSN: 2168-7161. DOI: 10.1109/TCC.2021.3135757 (cit. on p. 14).

[Gop93] M. Gopal. *Modern Control System Theory*. Wiley, 1993. ISBN: 9788122405033 (cit. on p. 9).

[Heg+17] C. Heger et al. "Application Performance Management: State of the Art and Challenges for the Future." In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. L'Aquila, Italy: Association for Computing Machinery, 2017, pp. 429–432. ISBN: 9781450344043. DOI: 10.1145/3030207.3053674 (cit. on p. 15).

[IEE90] IEEE. "IEEE Standard Glossary of Software Engineering Terminology." In: *IEEE Std 610.12-1990* (Dec. 1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064 (cit. on p. 9).

[IS21] K. Indrasiri and S. Suhothayan. *Design Patterns for Cloud Native Applications*. O'Reilly Media, Inc., May 2021. ISBN: 9781492090717 (cit. on p. 9).

[Kou+15] E. Kouroshfar et al. "A Study on the Role of Software Architecture in the Evolution and Quality of Software." In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. May 2015, pp. 246–257. DOI: 10.1109/MSR.2015.30 (cit. on p. 2).

[Lan+16] P. d. Lange et al. “Community Application Editor: Collaborative Near Real-Time Modeling and Composition of Microservice-based Web Applications.” In: *Modellierung 2016, 2.-4. März 2016, Karlsruhe - Workshopband*. Ed. by S. Betz and U. Reimer. Vol. P-255. LNI. GI, 2016, pp. 123–128. URL: https://dl.gi.de/20.500.12116/844 (cit. on p. 12).

[LF] J. Lewis and M. Fowler. *Microservices*. URL: https://martinfowler.com/articles/microservices.html (visited on 04/03/2023) (cit. on pp. 1, 2, 7).

[MAA19] N. Marie-Magdelaine, T. Ahmed, and G. Astruc-Amato. “Demonstration of an Observability Framework for Cloud Native Microservices.” In: *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. Apr. 2019, pp. 722–724 (cit. on p. 15).

[MGZ14] F. Montesi, C. Guidi, and G. Zavattaro. “Service-Oriented Programming with Jolie.” In: *Web Services Foundations*. Ed. by A. Bouguettaya, Q. Z. Sheng, and F. Daniel. New York, NY: Springer New York, 2014, pp. 81–107. ISBN: 978-1-4614-7518-7. DOI: 10.1007/978-1-4614-7518-7_4. URL: https://doi.org/10.1007/978-1-4614-7518-7_4 (cit. on p. 13).

[New21] S. Newman. *Building Microservices, 2nd Edition*. O’Reilly Media, Inc., Aug. 2021 (cit. on pp. 1, 8, 46).

[Nie+19] S. Niedermaier et al. “On Observability and Monitoring of Distributed Systems – An Industry Interview Study.” In: *Service-Oriented Computing*. Ed. by S. Yangui et al. Cham: Springer International Publishing, 2019, pp. 36–52. ISBN: 978-3-030-33702-5 (cit. on pp. 9, 15).

[Rad+18] F. Rademacher et al. “Microservice Architecture and Model-Driven Development: Yet Singles, Soon Married (?)” In: *Proceedings of the 19th International Conference on Agile Software Development: Companion*. XP ’18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450364225. DOI: 10.1145/3234152.3234193 (cit. on p. 9).

[Rad+19] F. Rademacher et al. “Viewpoint-Specific Model-Driven Microservice Development with Interlinked Modeling Languages.” In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. Apr. 2019, pp. 57–5709. DOI: 10.1109/SOSE.2019.00018 (cit. on p. 13).

[Rad+21] F. Rademacher et al. “Towards Holistic Modeling of Microservice Architectures Using LEMMA.” In: *ECSA 2021 Companion Volume, Virtual (originally: Växjö, Sweden), 13-17 September, 2021*. Ed. by R. Heinrich, R. Mirandola, and D. Weyns. Vol. 2978. CEUR Workshop Proceedings. CEUR-WS.org, 2021. URL: https://ceur-ws.org/Vol-2978/mde4sa-paper2.pdf (cit. on p. 13).

[Rav] V. Ravuri. *Sample E-Commerce application using Microservices / Cloud Native Architecture (CNA)*. URL: https://github.com/venkataravuri/e-commerce-microservices-sample (visited on 04/13/2023) (cit. on p. 55).

[Ric16] M. Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, Inc., Apr. 2016. ISBN: 9781491941607 (cit. on p. 8).

[Ric19] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, 2019, pp. 1–471. ISBN: 9781617294549 (cit. on pp. 8, 42–44, 46, 47, 55).

[RRS17] P. Raj, A. Raman, and H. Subramanian. *Architectural Patterns*. Packt Publishing, 2017. ISBN: 9781787287495 (cit. on p. 8).

[RS16] C. Richardson and F. Smith. *Microservices From Design to Deployment*. NGINX, Inc., 2016 (cit. on pp. 1, 2).

[RSS18] F. Rademacher, J. Sorgalla, and S. Sachweh. "Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective." In: *IEEE Software* 35.3 (May 2018), pp. 36–43. ISSN: 1937-4194. DOI: 10.1109/MS.2018.2141028 (cit. on p. 9).

[RSZ19] F. Rademacher, S. Sachweh, and A. Zündorf. "Aspect-Oriented Modeling of Technology Heterogeneity in Microservice Architecture." In: *2019 IEEE International Conference on Software Architecture (ICSA)*. Mar. 2019, pp. 21–30. DOI: 10.1109/ICSA.2019.00011 (cit. on p. 13).

[Rup10] N. B. Ruparelia. "Software Development Lifecycle Models." In: *SIGSOFT Softw. Eng. Notes* 35.3 (May 2010), pp. 8–13. ISSN: 0163-5948. DOI: 10.1145/1764810.1764814. URL: https://doi.org/10.1145/1764810.1764814 (cit. on p. 18).

[Som10] I. Sommerville. *Software Engineering*. 9th. USA: Addison-Wesley Publishing Company, 2010, pp. 27–228. ISBN: 0137035152 (cit. on pp. 18, 33, 111).

[Sor+18] J. Sorgalla et al. "AjiL: Enabling Model-Driven Microservice Development." In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. ECSA '18. Madrid, Spain: Association for Computing Machinery, 2018. ISBN: 9781450364836. DOI: 10.1145/3241403.3241406 (cit. on p. 11).

[Sor+20] J. Sorgalla et al. "Modeling Microservice Architecture: A Comparative Experiment towards the Effectiveness of Two Approaches." In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. SAC '20. Brno, Czech Republic: Association for Computing Machinery, 2020, pp. 1506–1509. ISBN: 9781450368667. DOI: 10.1145/3341105.3374065 (cit. on p. 13).

[Sul+22] A. Suljkanović et al. "Developing Microservice-Based Applications Using the Silvera Domain-Specific Language." In: *Applied Sciences* 12.13 (2022). ISSN: 2076-3417. DOI: 10.3390/app12136679. URL: https://www.mdpi.com/2076-3417/12/13/6679 (cit. on p. 13).

[Ter+17] B. Terzić et al. "MicroBuilder: A Model-Driven Tool for the Specification of REST Microservice Architectures." In: Mar. 2017 (cit. on p. 12).

[Ter+18a] B. Terzić et al. "A Model-Driven Approach to Microservice Software Architecture Establishment." In: Sept. 2018, pp. 73–80. DOI: 10.15439/2018F370 (cit. on p. 12).

[Ter+18b] B. Terzić et al. "Development and evaluation of MicroBuilder: a Model-Driven tool for the specification of REST Microservice Software Architectures." In: *Enterprise Information Systems* 12.8-9 (2018), pp. 1034–1057. DOI: 10.1080/17517575.2018.1460766. eprint: https://doi.org/10.1080/17517575.2018.1460766. URL: https://doi.org/10.1080/17517575.2018.1460766 (cit. on pp. 12, 13).

[TLP20] D. Taibi, V. Lenarduzzi, and C. Pahl. "Microservices Anti-patterns: A Taxonomy." In: *Microservices: Science and Engineering*. Ed. by A. Bucchiarone et al. Cham: Springer International Publishing, 2020, pp. 111–128. ISBN: 978-3-030-31646-4. DOI: 10.1007/978-3-030-31646-4_5. URL: https://doi.org/10.1007/978-3-030-31646-4_5 (cit. on p. 84).

[Usm+22] M. Usman et al. "A Survey on Observability of Distributed Edge & Container-Based Microservices." In: *IEEE Access* 10 (2022), pp. 86904–86919. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3193102 (cit. on p. 15).

[WHR14] J. Whittle, J. Hutchinson, and M. Rouncefield. "The State of Practice in Model-Driven Engineering." In: *IEEE Software* 31.3 (May 2014), pp. 79–85. ISSN: 1937-4194. DOI: 10.1109/MS.2013.65 (cit. on p. 9).

[Wiz+17] P. Wizenty et al. "MAGMA: Build Management-Based Generation of Microservice Infrastructures." In: *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*. ECSA '17. Canterbury, UK: Association for Computing Machinery, 2017, pp. 61–65. ISBN: 9781450352178. DOI: 10.1145/3129790.3129821 (cit. on p. 11).

[Woh+12] C. Wohlin et al. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012. ISBN: 3642290434 (cit. on p. 87).

[ZGD19] Y. Zhang, Y. Gan, and C. Delimitrou. "μqSim: Enabling Accurate and Scalable Simulation for Interactive Microservices." In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2019, pp. 212–222. DOI: 10.1109/ISPASS.2019.00034 (cit. on p. 14).

Glossary

AML Architecture Modelling Languages

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

APM Application Performance Monitoring

CNCF Cloud Native Computing Foundation

DSL Domain-Specific Language

EMF Eclipse Modeling Framework

functional requirement Functional requirement is a statement of a service a system should provide, how a system should react to particular inputs, and how a system should behave in particular situations [Som10].

gRPC gRPC Remote Procedure Call

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

KPI Key Performance Indicator

MAPI-DM Microservices API Dependency Model

MDD Model-Driven Development

MDE Model-Driven Engineering

MSA Microservice Architecture

MVC Model-View-Controller

OTLP OpenTelemetry Protocol

REST Representational State Transfer

RPI Remote Procedure Invocation

SDK Software Development Kit

UI User Interface

UML Unified Modeling Language

URL Uniform Resource Locator

YAML YAML Ain't Markup Language